

Programming, Specification, and Interactive Theorem Proving

Towards a Unified Language based on
Equational Logic, Rewriting Logic, and
Type Theory

DISSERTATION
ZUR ERLANGUNG DES DOKTORGRADES
AM FACHBEREICH INFORMATIK
DER UNIVERSITÄT HAMBURG

vorgelegt von

Mark-Oliver Stehr

geboren in Hamburg

Hamburg 2002

Genehmigt vom Fachbereich Informatik der Universität Hamburg auf Antrag von

Prof. Dr. Rüdiger Valk (Universität Hamburg)

Prof. Dr. José Meseguer (University of Illinois at Urbana-Champaign)

Prof. Dr. Dr. h.c. Wilfried Brauer (Technische Universität München)

Hamburg, den 23. September 2002
(Datum der Disputation)

Prof. Dr. Siegfried Stiehl
(Dekan des Fachbereichs Informatik)

Preface

The observation that the development of mathematical theories and the development of information systems essentially involves the same activities, namely modeling, specification, and validation, with special attention to important aspects such as modularity, reusability and information hiding and under consideration of related notions of calculation, computation, and proof, has on many occasions challenged my mind with the question if the apparently artificial gap, which is for instance especially evident between the disciplines of mathematics and software engineering, could eventually be reduced, possibly even on the basis of a unified language. Although this thesis does by no means answer this difficult question, I have learned much in the recent years, especially during the occupation with my thesis, about several formalisms which seem to contribute to this goal in their own limited way. Two of these formalisms, type theory and rewriting logic, constitute the basis of this thesis, which is concerned with new applications and a possible integration of these approaches.

It is needless to say that this thesis would not have been possible without the great efforts by those who laid the foundation on which its contents is based. On the other hand, many parts of this thesis are also the result of a fruitful collaboration with other researchers. In fact, all the main chapters of this thesis are equipped with individual acknowledgments, because the chapters have been written in the context of different projects and with the helpful involvement of different people. Some chapters have been written during my appointment as research assistant at the University of Hamburg, and others have been written during several fellowships at SRI International in California. At this point I would like to take the opportunity to express my gratitude for the help and advice of José Meseguer and Rüdiger Valk, who both in many different ways contributed a lot to the successful completion of this thesis. However, the work on my thesis would not have been so enjoyable without so many great friends and colleagues, of which I would like to thank especially, and each for very different reasons, Anahita Hassanzadeh, Berndt Farwer, Carolyn Talcott, Cesar Muñoz, Christiano Braga, Daniel Moldt, Francisco Duran, Francoise Hamester, Grit Denker, Hassan Saidi, Kevin Hall, Manfred Kudlek, Manuel Clavel, Marco Molteni, Maria Sorea, Matthias Jantzen, Michael Köhler, Narciso Martí-Oliet, Olaf Kummer, Pavel Naumov, Peter Ölveczky, Rachana Raizada, Rachid Kabeche, Randy Gray, Steven Eker, Sonia Tsui, Uwe Fenske, and Wolfram Roisch. Most importantly, I am indebted to my parents Gisela and Günther, who always supported me and are simply the best parents that I can imagine.

Abstract

Inspired by a number of different applications of rewriting logic, equational logic, and type theory that we present and further advance in this thesis, we study a unified formalism based on the key aspects of these quite different lines of research. The resulting formalism, that we call the open calculus of constructions, is intended as a step towards our long-term goal of developing a unified language for programming, specification and interactive theorem proving.

We begin our work by exploring the application of rewriting logic as a semantic framework for concurrency. To this end, we give a unified treatment of different classes of Petri nets, a typical and important representative of a class of formalisms that are used for the modeling and specification of concurrent and distributed systems based on a multiset representation of a distributed state space. Specifically, we continue the line of research initiated by Meseguer and Montanari under the motto “Petri nets are monoids” by giving a rewriting semantics for different Petri nets classes. In particular, we have covered important high-level Petri net models, namely algebraic net specifications and colored Petri nets, and we have proved that the models of our representations are naturally isomorphic to the well-known Best-Devillers process semantics. Apart from their contribution to a conceptual unification in this field, the main practical advantage of our representations in rewriting logic is their executability, which allows us to use a rewriting engine such as Maude for the efficient symbolic execution of system models and for their analysis.

The next application addressed in this thesis is the use of type theory, more precisely the calculus of inductive constructions, as a logical framework and for metalogical reasoning. Specifically, we have used the COQ proof assistant in a formally rigorous development of a UNITY-style temporal logic, which generalizes the original UNITY approach in important aspects. Since all inference rules of the temporal logic are proved as theorems in the metalogic, the result of the development is a verified temporal logic library, which due to the use of labeled transition systems as a semantic basis, can be employed for a wide range of system models, Petri nets and rewriting logic specifications being particular examples. The development also includes a new application of the proposition-as-types interpretation in the context of compositional reasoning.

The use of membership equational logic or rewriting logic as a semantic and logical framework for higher-order languages, or more generally languages with binding constructs, obviously requires a first-order treatment of names and relevant operations such as substitutions. To systematically address such applications, we develop CINNI, a new calculus of names and substitutions, that takes names seriously in the sense that it does not abstract from names, and is generic in the sense that it can be instantiated to arbitrary object languages. Our calculus unifies the standard named notation and a notation based on de Bruijn indices by employing

a representation that was originally developed by Berkling for the λ -calculus. It furthermore nicely generalizes the calculus λv of explicit substitutions developed by Lescanne, and, as we show, most metatheoretic results can be generalized to the new calculus. We furthermore give a very general confluence result for the composition of CINNI with the equations or rules capturing the dynamics of the object language, and we in particular discuss how our approach can be applied to the representation of the untyped λ -calculus, Abadi and Cardelli’s object calculus, also called the ζ -calculus, and Milner’s π -calculus for communicating and mobile systems. As a real-world application of CINNI we briefly discuss a specification of an active network programming language in the rewriting-logic-based language Maude.

We more specifically address the use of membership equational logic and rewriting logic as a first-order logical framework by representing an important class of pure type systems. Pure type systems generalize a variety of different type theories, including the calculus of constructions and its well-known subsystems, and can be seen as higher-order logics via the propositions-as-types interpretation. Following a methodology based on Meseguer’s general logics in combination with rewriting logic as a concrete logical framework, we have studied representations of pure type systems at different levels of abstractions, ranging from an abstract textbook representation to a more concrete executable representation of an important subclass, which can directly serve as a type inference and type checking algorithm. The latter representation is based on a new notion of uniform pure type systems, which take names seriously thanks to the CINNI calculus and simultaneously offer a possible solution to the known problem with α -closure pointed out by Pollack. Using an example, in which we validate proofs developed with the LEGO proof assistant in an extension of the calculus of constructions with universes, we have demonstrated how our approach directly leads to an executable prototype in a rewriting logic language such as Maude.

As an application of type theory in the context of classical reasoning we study Howe’s HOL/Nuprl connection, which addresses the problem of formal interoperability between proof assistants, from the viewpoint of Meseguer’s general logics. We supplement Howe’s semantic justification by a proof-theoretic correctness argument, a piece of work which has led to proof-translation as new interesting application (explored in joint work with Naumov) that goes beyond Howe’s original HOL/Nuprl connection. From a theoretical perspective we found that the core idea of the HOL/Nuprl connection, namely the beneficial coexistence of an intensional and an extensional logic in the same formal system, does not rely on any of the advanced concepts of Nuprl, but can equally well be used in Martin-Löf’s type theory and can further be easily adopted to type theories in the line of calculus of constructions.

The final and main contribution of this thesis is the development of a formalism that we call the open calculus of constructions (OCC). It is based on the sur-

prisingly powerful interaction between its two key features, namely dependent types, in the spirit of Martin-Löf's type theory and the calculus of constructions, and the computational system of rewriting logic and its underlying membership equational logic, which is based on conditional rewriting modulo equations. The applications of membership equational logic, rewriting logic, and type theory, studied in this thesis have not only inspired the development of this unifying formalism, but they become applications of OCC itself and benefit from its use in an essential way. On the theoretical side, we introduce OCC by presenting a classical set-theoretic semantics and a formal system for which we prove soundness and consistency as a logic. The formal system is used to define derivable judgements together with their operational semantics, and is based on the ideas that we developed earlier in the context of uniform pure type systems. The model-theoretic semantics that we develop in this thesis is a very intuitive semantics with proof-irrelevance for impredicative universes, but unlike existing approaches it is more direct and can be given independently of the formal system. Using an experimental prototype of OCC, that we implemented in Maude following the approach to the specification of type theories mentioned before in combination with reflective techniques, we have developed a large collection of examples, many of which are closely related to the applications discussed earlier in this thesis. These examples do not only convey the pragmatics of OCC, but they simultaneously provide a proof-of-concept for our approach. Among the topics covered by our examples we find executable equational/behavioral specifications, programming with dependent types, symbolic execution of system models, formalization of algebraic and categorical concepts, inductive/coinductive theorem proving, and theorem proving modulo equational theories.

Zusammenfassung

Ausgehend von verschiedenen Anwendungen von Termersetzungsllogik (rewriting logic), Gleichungslogik (equational logic), und Typtheorie, die wir in dieser Arbeit untersuchen und weiterentwickeln, studieren wir einen einheitlichen Formalismus, der auf den Hauptcharakteristika dieser recht unterschiedlichen Forschungsrichtungen basiert. Der resultierende Formalismus, den wir als offenes Kalkül der Konstruktionen (open calculus of constructions) bezeichnen, ist als erster Schritt in Richtung unseres langfristigen Ziels, der Entwicklung einer einheitlichen Sprache zur Programmierung, Spezifikation, und interaktivem Theorembeweisen, zu verstehen.

Unsere Arbeit beginnt mit der Untersuchung der Anwendbarkeit von Termersetzungsllogik als semantisches Rahmenwerk (semantic framework) für Nebenläufigkeit. Hierzu geben wir eine einheitliche Behandlung verschiedener Petrinetz-Modelle, eines typischen und wichtigen Repräsentanten einer Klasse von Formalismen, die zur Modellierung und Spezifikation von nebenläufigen und verteilten Systemen benutzt werden und auf einer Multimengen-Repräsentation des verteilten Zustandsraumes basieren. Speziell führen wir die Forschungsrichtung fort, die von Meseguer und Montanari unter dem Motto “Petrinetze sind Monoide” initiiert wurde, indem wir eine Termersetzungsemantik für verschiedene Petrinetz-Klassen angeben. Insbesondere decken wir wichtige höhere Petrinetz-Modelle, genauer algebraische Netzspezifikationen und gefärbte Petrinetze ab, und wir beweisen, daß die Modelle unserer Repräsentationen eine natürliche Isomorphie zur bekannten Prozess-Semantik von Best und Devillers aufweisen. Zusätzlich zu ihrem Beitrag zu einer konzeptuellen Vereinheitlichung ist der wichtigste praktische Vorteil unserer Repräsentationen in Termersetzungsllogik ihre Ausführbarkeit, die es uns erlaubt, Termersetzungsmaschinen wie z.B. Maude zur effizienten symbolischen Ausführung von Systemmodellen sowie zu deren Analyse zu nutzen.

Die nächste Anwendung, die in dieser Arbeit behandelt wird, betrifft die Verwendung der Typtheorie, genauer des induktiven Kalküls der Konstruktionen (inductive calculus of constructions) als logisches Rahmenwerk (logical framework) und für metalogische Beweise. Speziell haben wir den COQ-Beweisassistenten in einer rigoros formalen Entwicklung einer Temporallogik im UNITY-Stil eingesetzt, die den ursprünglichen Ansatz in wichtigen Punkten verallgemeinert. Da die Inferenzregeln der Temporallogik als Theoreme in der Metalogik bewiesen wurden, ist das Resultat dieser Entwicklung eine verifizierte Temporallogik-Bibliothek, die dank der Verwendung von allgemeinen, beschrifteten Transitionssystemen als semantische Basis für ein weites Spektrum von Systemmodellen (Petrinetze und Termersetzungsllogik-Spezifikationen wären konkrete Beispiele) eingesetzt werden kann. Unsere Entwicklung enthält u.a. eine neuartige Anwendung der Interpretation von logischen Formeln als Typen (propositions-as-types interpretation) im Kontext der kompositionalen Verifikation.

Die Verwendung von Membership-Gleichungslogik (membership equational logic) oder Termersetzungsllogik als semantisches und logisches Rahmenwerk für Sprachen höherer Ordnung, oder allgemeiner für Sprachen mit Konstrukten zur Namensbindung, benötigt offensichtlich eine Behandlung von Namen und relevanten Operationen wie Substitutionen mit Mitteln erster Ordnung. Um solche Anwendungen systematisch anzugehen, haben wir CINNI entwickelt, ein neues Kalkül der Namen und Substitutionen, das Namen in dem Sinne respektiert, daß es nicht von ihnen abstrahiert, und das ferner generisch ist in dem Sinne, daß es für beliebige Objekt-Sprachen instantiiert werden kann. Unser Kalkül vereinheitlicht die übliche Notation mit Namen und die namenlose Notation basierend auf de-Bruijn-Indizes, indem es eine Repräsentation benutzt, die ursprünglich von Berking für das λ -Kalkül entwickelt wurde. Ferner verallgemeinert CINNI das Kalkül λv der expliziten Substitutionen von Lescanne und, wie wir zeigen, lassen sich die meisten metatheoretischen Resultate auf das neue Kalkül verallgemeinern. Schließlich zeigen wir ein sehr allgemeines Konfluenz-Resultat für die Komposition von CINNI mit Gleichungen oder Regeln, die die Dynamik der Objektsprache widerspiegeln, und wir diskutieren, wie unserer Ansatz zu Repräsentationen des ungetypten λ -Kalküls, des Objekt-Kalküls von Abadi und Cardelli, auch als ζ -Kalkül bezeichnet, und Milners π -Kalküls für kommunizierende und mobile Systeme verwendet werden kann. Als eine Anwendung von CINNI aus der Praxis diskutieren wir kurz die Spezifikation einer Programmiersprache für aktive Netzwerke in der auf Termersetzungsllogik basierten Sprache Maude.

Etwas spezifischer behandeln wir die Verwendung von Membership-Gleichungslogik und Termersetzungsllogik als logisches Rahmenwerk erster Ordnung, indem wir eine wichtige Klasse der reinen Typsysteme (pure type systems) repräsentieren. Reine Typsysteme verallgemeinern eine reichhaltige Klasse von verschiedenen Typtheorien einschließlich des Kalküls der Konstruktionen und seiner bekannten Teilsysteme und können ferner als Logiken höherer Ordnung entsprechend der Interpretation von logischen Formeln als Typen (propositions-as-types interpretation) angesehen werden. Unter Verwendung eines methodischen Ansatzes, der auf Meseguers allgemeinen Logiken (general logics) in Kombination mit Termersetzungsllogik als konkretes logisches Rahmenwerk basiert, haben wir Repräsentationen von reinen Typsystemen auf verschiedenen Abstraktionsebenen studiert, angefangen von einer abstrakten lehrbuchartigen Repräsentation bis hin zu einer konkreteren und ausführbaren Repräsentation einer wichtigen Unterklasse, die in sehr direkter Weise als Typinferenz- und Typprüfungsalgorithmus genutzt werden kann. Die letztere Repräsentation basiert auf einem neuen Begriff der einheitlichen reinen Typsysteme (uniform pure type systems), die dank der Verwendung des CINNI-Kalküls Namen respektieren, und gleichzeitig eine mögliche Lösung zum bekannten Problem der α -Abgeschlossenheit darstellen, auf das Pollack hinwies. Mit Hilfe eines Beispiels, der Validation von Bewei-

sen, die mit dem LEGO-Beweisassistent in einer Erweiterung des Kalküls der Konstruktionen um Universen ausgeführt wurden, zeigen wir, wie unser Ansatz unmittelbar zu einem ausführbaren Prototyp in einer auf Termersetzungs-Logik basierenden Sprache wie Maude führt.

Als Anwendung der Typtheorie in Kontext des klassischen Beweises studieren wir Howes HOL/Nuprl-Verbindung, die sich mit dem Problem der formalen Interoperabilität zwischen Beweisassistenten aus der Sicht von Meseguers allgemeinen Logiken beschäftigt. Wir ergänzen Howes semantische Rechtfertigung durch ein beweistheoretisches Korrektheitsargument, eine Arbeit die Beweisübersetzung als neue interessante Anwendung hat (untersucht in Zusammenarbeit mit Naumov), und damit über Howes ursprüngliche HOL/Nuprl-Verbindung hinausgeht. Aus theoretischer Sicht fanden wir, daß die Kernidee der HOL/Nuprl-Verbindung, nämlich die Koexistenz einer intensionalen und einer extensionalen Logik in einem einzigen formalen System, nicht auf die speziellen Eigenschaften von Nuprl angewiesen ist, sondern ebenso in Martin-Löfs Typtheorie verwendet werden kann, und ferner leicht an Typtheorien auf der Linie des Kalküls der Konstruktionen angepasst werden kann.

Der letzte und Hauptbeitrag dieser Arbeit ist die Entwicklung eines Formalismus, den wir als offenes Kalkül der Konstruktionen (open calculus of constructions, OCC) bezeichnen. Er basiert auf der überraschend mächtigen Interaktion zwischen seinen beiden Hauptcharakteristika, nämlich abhängigen Typen im Sinne von Martin-Löfs Typtheorie und des Kalküls der Konstruktionen, und dem Berechnungssystem der Termersetzungslogik und seiner unterliegenden Membership-Gleichungslogik, das auf bedingter Termersetzung modulo Gleichungen basiert. Die Anwendungen von Membership-Gleichungslogik, Termersetzungs-Logik und Typtheorie, die wir in dieser Arbeit studierten, waren nicht nur eine wichtige Inspiration für die Entwicklung dieses einheitlichen Formalismus, sondern werden selbst zu Anwendungen von OCC und profitieren wesentlich von seinem Einsatz. Auf der theoretischen Seite führen wir OCC durch Angabe einer klassischen, mengentheoretischen Semantik und eines formalen Systems ein, für das wir Korrektheit und logische Konsistenz beweisen. Das formale System wird benutzt, um die ableitbaren Urteile und ihre operationale Semantik zu definieren und basiert auf den Ideen, die wir zuvor im Kontext der einheitlichen, reinen Typsysteme entwickelt haben. Die modelltheoretische Semantik, die wir in dieser Arbeit entwickeln, ist eine sehr intuitive Semantik mit Beweis-Irrelevanz (proof-irrelevance semantics) für imprädikative Universen, aber anders als in existierenden Ansätzen ist sie direkter und kann unabhängig vom formalen System angegeben werden. Unter Verwendung eines experimentellen Prototyps von OCC, den wir mit Hilfe des obigen Ansatzes zur Spezifikation von Typtheorien in Kombination mit reflektiven Techniken in Maude entwickelt haben, wurde eine umfangreiche Sammlung von Beispielen erstellt, von denen viele eng mit den vorher diskutierten Anwendungen zusammenhängen. Die Bei-

spiele vermitteln nicht nur die Pragmatik von OCC sondern zeigen gleichzeitig die Realisierbarkeit und den Nutzen seiner Konzepte. Unter den Themen, die beispielsweise behandelt werden, finden sich ausführbare, gleichungsbasierte und verhaltensorientierte Spezifikationen, Programmierung mit abhängigen Typen, symbolische Ausführung von Systemmodellen, Formalisierung von algebraischen und kategorientheoretischen Begriffen, induktives/coinduktives Theorembeweisen und Beweisen modulo Gleichungstheorien.

Contents

1 Introduction	19
1.1 Overview of the Thesis	21
2 Preliminaries	31
2.1 Notation and Basic Concepts	32
2.2 Transition Systems	33
2.3 Entailment Systems	34
2.4 From Equational Logic to Rewriting Logic	39
2.4.1 Many-Sorted Algebra	40
2.4.2 Structural vs. Computational Equations	46
2.4.3 Order-Sorted Algebra	49
2.4.4 Membership Equational Logic	52
2.4.5 Rewriting Logic	64
2.4.6 The Maude Rewriting Engine	75
2.5 From Higher-Order Logics to Logical Type Theories	78
2.5.1 The Calculus of Constructions	84
2.5.2 Propositions as Types and Proofs as Objects	86
2.5.3 The Generalized Calculus of Constructions	88
2.5.4 The Calculus of Inductive Constructions	89
2.5.5 The COQ Proof Development System	92
3 Rewriting Logic as a Semantic Framework:	
Representing High-Level Petri Nets	95
3.1 Place/Transition Nets	97
3.1.1 A Toy Example	103
3.1.2 Rewriting Semantics in the General Case	105

3.1.3	Petri Nets with Test Arcs	111
3.2	High-Level Petri Nets	113
3.2.1	Colored Nets and Colored Net Specifications	114
3.2.2	Algebraic Net Specifications	116
3.2.3	A Case Study	120
3.2.4	Rewriting Semantics in the General Case	125
3.2.5	Execution of Algebraic Net Specifications	134
3.2.6	Abstract Net Execution	137
3.3	Generalizations of Our Approach	142
3.3.1	Higher-Order Programming Languages	142
3.3.2	Higher-Order Logic and Types	142
3.3.3	Objects and Active Tokens	143
3.3.4	Structure of the State Space	144
3.4	Final Remarks	145
3.5	Acknowledgements	146
4	Meta-Logical Reasoning in the Calculus of Inductive Constructions: A Formalized Generalization of UNITY	149
4.1	A Sound and Complete Fragment of UNITY	152
4.1.1	The Programming Language	152
4.1.2	The Temporal Logic	153
4.2	A UNITY-Style Logic for Labeled Transition Systems	156
4.2.1	State Predicates and Functions	157
4.2.2	Basic Concepts	157
4.2.3	Safety Assertions	158
4.2.4	Liveness Assertions	160
4.3	A Verified Temporal Logic Library	162
4.3.1	Mathematical Prelude	164
4.3.2	An Abstract Frame	166
4.3.3	State Predicates	168
4.3.4	Safety Assertions	170
4.3.5	Liveness Assertions	179
4.3.6	System Composition	193
4.3.7	A State Space with Typed Variables	195

- 4.3.8 Operations on States and Variables 197
- 4.3.9 State Predicates and Variables 200
- 4.3.10 Elimination Theorem 203
- 4.4 A Toy Example in Program Verification 204
 - 4.4.1 A Concrete Frame 206
 - 4.4.2 Partial Correctness 209
 - 4.4.3 Total Correctness 211
- 4.5 Compositionality via Intensionality 213
 - 4.5.1 Extending the Example 218
- 4.6 Related Work 220
- 4.7 Final Remarks 222
- 4.8 Acknowledgements 224

**5 First-Order Representations of Higher-Order Formalisms:
A Calculus of Names and Substitutions 225**

- 5.1 Indexed Names and Named Indices 227
- 5.2 Explicit Substitutions 230
- 5.3 Metatheoretic Properties of CINNI 233
 - 5.3.1 Strong Normalization 234
 - 5.3.2 Equational Properties 237
 - 5.3.3 Preservation of Confluence 245
- 5.4 Applications 251
 - 5.4.1 Higher-Order Functions: The Lambda-Calculus 251
 - 5.4.2 Object-Orientation: The Sigma-Calculus 254
 - 5.4.3 Mobile Processes: The Pi-Calculus 256
 - 5.4.4 Further Applications 260
- 5.5 Three Orthogonal Research Directions 261
- 5.6 Final Remarks 265
- 5.7 Acknowledgements. 266

**6 Rewriting Logic as a Logical Framework:
Representing Pure Type Systems 269**

- 6.1 Overview and Main Results 272
- 6.2 The Metalogical View of PTSs 274
 - 6.2.1 PTSs in Membership Equational Logic 276

6.2.2	Taking Names Seriously	280
6.2.3	Uniform Pure Type Systems	282
6.2.4	A Conservative Optimization	286
6.3	The Meta-Operational View of PTSs	288
6.3.1	Uniform PTSs in Membership Equational Logic	290
6.3.2	Uniform PTSs in Rewriting Logic	291
6.4	Final Remarks	301
6.5	Acknowledgements	302
7	Classical Logic via Type Theory:	
	A Proof-Theoretic Approach to the HOL/Nuprl Connection	305
7.1	Preliminaries	308
7.2	The Logic of HOL	309
7.2.1	Syntactic Categories	309
7.2.2	Deduction and Entailment	312
7.2.3	Theories	313
7.3	The Type Theory of Nuprl	314
7.3.1	Syntactic Categories	315
7.3.2	Derivability and Entailment	317
7.3.3	Classical Extension	318
7.3.4	Theories	319
7.4	Theory Translation	320
7.5	Correctness of the Translation	325
7.6	Nuprl Theory Interpretations	329
7.6.1	Interpreting the Logical Theory	330
7.7	Practical Work	331
7.7.1	An Executable Formal Specification	331
7.7.2	Towards a Proof Translator	333
7.8	Final Remarks	334
7.9	Acknowledgements	335
8	Towards a Unified Language:	
	The Open Calculus of Constructions	337
8.1	Presentation of the Calculus	342
8.1.1	Syntax	342

<i>CONTENTS</i>	17
8.1.2 Semantics	348
8.1.3 Formal System	355
8.1.4 Soundness and Consistency	370
8.2 Examples in Specification and Programming	373
8.2.1 Executable Equational Specifications	373
8.2.2 Dependent Types and Universes	396
8.2.3 Executable Behavioral Specifications	409
8.2.4 Representation of Object Languages	413
8.2.5 Elimination and Coelimination Principles	418
8.2.6 Higher-Order Logic and Equality	425
8.2.7 Algebras and Categories	431
8.2.8 Executable Rewrite Specifications	446
8.3 Examples in Interactive Theorem Proving	459
8.3.1 Goal-Oriented Refinement	460
8.3.2 Reasoning with Equality	462
8.3.3 Inductive Theorem Proving	464
8.3.4 Coinductive Theorem Proving	469
8.3.5 Theorem Proving Modulo	472
8.4 Final Remarks	481
8.5 Acknowledgements	487
9 Conclusions and Future Work	491
A Pure Type Systems in Rewriting Logic	515
A.1 The Executable Specification in Maude	515
A.2 An Application: Validation of Proofs	521
B Specifying a Programming Language for Active Networks	527
B.1 The Executable Specification in Maude	528
Bibliography	547

Chapter 1

Introduction

Applications of computer science are manifold and produce a constant demand for new formalisms or extensions of existing ones which have reached their boundaries. Today we witness a huge variety of formalisms which are used for programming, specification, modeling, validation, verification, and reasoning. In addition to their use, we can classify them along several dimensions. Some formalisms are purely textual, other are based on diagrams and could be called visual formalisms. Some formalisms have a solid mathematical foundation and have possibly found their way into practice, others are well-established in practice but have not found a suitable foundation yet. Some formalisms are concerned with particular physical realizations, e.g. software or hardware, other formalisms intentionally abstract from such issues. Similarly, some formalisms are designed to deal explicitly with issues like concurrency, distribution, time, mobility, security, while others assume a higher level of abstraction. Some formalisms are very general, i.e. based on common concepts such as functions, while others favor richer paradigms such as object orientation or agent orientation.

The long-term research objective which provides the general setting for this thesis is the development of programming, specification and verification formalisms and tools which are sufficiently general to cope with the diversity of paradigms that are in use today. We try to contribute to this goal in a systematic way: First, we explore new applications of existing approaches, thereby widening their scope of applicability. Second, we isolate limitations of these approaches and look for possible solutions in terms of a common generalization.

In this thesis we have selected two promising formalisms, namely rewriting logic and type theory. Both formalisms are highly general in the sense that they can be used as *metaformalisms* to represent many other formalisms, that we call *object-formalisms*, in natural ways. On the other hand, their features are nearly complementary, so that a lot could be gained from a unification of these two lines of research.

Rewriting logic [Mes92] contains *equational logic* as a sublogic and is used for specification and prototyping in this thesis. The main quality of rewriting logic that we exploit in this thesis is its syntactic and semantic simplicity, which makes it suitable to explain other formalisms via its use as a logical and semantic framework [Mes96, MOM94, MOM96, MMO95, Mes98, BCM00, Mes92]. Rewriting logic is a first-order formalism that is simple enough to serve as a satisfactory explanation of more complex formalisms such as (higher-order) programming languages and (higher-order) type theories. In fact, rewriting logic (and its equational sublogic) is weak enough to admit initial and free (term) models, such that we can regard its specifications essentially as equational inductive definitions of finitary nature. In addition, rewriting logic has an operational semantics based on a very flexible notion of computation that allows us to write executable specifications. Such specifications can be used for prototyping, analysis and even as actual implementations, thanks to the availability of efficient rewriting engines

such as Maude [CDE⁺99a, CDE⁺00b] and ELAN [BKK⁺96, BCD⁺98, BKK⁺98]. The second formalism that we are concerned with is *type theory* [ML84, Tho91, PSN90, Luo94, CH88], which in the broad sense refers to a rich class of formalisms with very different intentions. We are in particular interested in type theories in connection with *higher-order logic*, which is used for specification and formal reasoning in this thesis. The main quality of higher-order logic that is important for us in the context of this thesis is that, thanks to its treatment of functions and predicates as first-class objects, it allows highly generic descriptions. As a consequence, higher-order logic is expressive enough to internalize generic reasoning principles, such as induction schemes. Hence, it can not only serve as an expressive specification language, but it also provides a natural framework for reasoning about specifications. The type theories that we mainly use in this thesis are expressive *logical type theories*, which admit the interpretation of higher-order logic inside the type theory. This so-called propositions-as-types interpretation unifies the disciplines of typed functional programming and interactive theorem proving in a way which has important advantages for the design of logics and their implementation. Interactive theorem proving in higher-order logics and in particular in logical type theories is well-supported by several proof development systems that are in use today. Systems that we employ in the context of this thesis are HOL [GM93b], Nuprl [CAB⁺86], LEGO [Pol94], and COQ [BBC⁺99].

1.1 Overview of the Thesis

In Chapter 2 we recall general concepts such as transition systems and entailment systems which are used in various places of the thesis. Furthermore, this chapter gives an informal introduction to equational logic, rewriting logic, higher-order logic, and logical type theory. The logics which are most relevant for this thesis, namely membership equational logic [BJM00], the corresponding version of rewriting logic [Mes92], and the calculus of constructions [CH88] with possible extensions, are presented in some detail. Particular tools that we use heavily in this thesis, namely the Maude rewriting engine and the COQ proof development system, are also discussed.

Each of the subsequent Chapters 3 – 8 presents new applications of rewriting logic or type theory in a typical domain. The experience obtained from these applications has a direct impact on the design and use of the *open calculus of constructions* introduced in Chapter 8, which unifies the capabilities of equational logic, rewriting logic and type theory in a single calculus. In spite of this general objective, each of the Chapters 3 – 8 is interesting in its own right and can be read independently, since it continues a previous line of research, provides solutions to concrete problems, and suggests new formalisms or methods that are of theoretical and practical interest. In the following we give a brief overview of

each chapter in the body of this thesis, with particular emphasis on outlining the connections between these chapters.

Chapter 3

Rewriting Logic as a Semantic Framework: Representing High-Level Petri Nets

The use of rewriting logic as a semantic framework [Mes92, Mes96] is particularly attractive due to its simplicity in syntax and semantics and due to its generality which allows a natural representation of a wide variety of different system models. Here we fill a gap in the picture by a representation of different classes of Petri nets, one of the most popular models for concurrency. Our work can be seen as a continuation of a line of research initiated by Meseguer and Montanari under the motto “Petri Nets are Monoids” [MM90], but in contrast to earlier work we use rewriting logic as a semantic framework and cover new classes of Petri nets.

Place/transition nets [Pet62], the basic model of Petri nets where tokens are indistinguishable, have been a major source of inspiration for the development of rewriting logic [Mes92]. Another source has been linear logic [Gir87], which as a resource-sensitive logic can be seen as a logical generalization of place/transition nets [MOM91b]. Today Petri net research is concerned with a variety of extensions of the basic model to accommodate practical needs to deal with issues such as data and time, as well as its integration with many of the existing specification and programming paradigms. Unfortunately, this development has led to a degree of diversification and fragmentation which is unsatisfactory from theoretical and practical viewpoints.

The main objective of this chapter is to show how rewriting logic can contribute to a unified view of Petri nets by focussing on their essential features, namely locality and monotonicity, which are not surprisingly the basic principles of rewriting logic. To this end, we show how place/transition nets [Pet62] and algebraic net specifications, generalizing [Rei91], can be represented using rewriting logic in a logically and operationally satisfactory way. The research reported in this chapter generalizes the results of [MM90] and is based on our earlier work [Stec] on the representation of algebraic net specifications. It has been further continued in [SMÖ01b, SMÖ01a] to cover Petri nets with test arcs (that we also cover in this chapter) and timed Petri nets.

We furthermore argue that the general form of algebraic net specifications based on membership equational logic with its notion of executability is sufficiently expressive to represent colored Petri nets [Jen92] over (possibly higher-order) programming languages. This can, for instance, be achieved using the systematic approach to the representation of higher-order programming languages in membership equational logic based on the CINNI calculus which is developed in Chapter 5. Another noteworthy point is that our representation of algebraic net spec-

ifications is not limited to membership equational logic, but can be naturally extended to colored net specifications over a higher-order logic such as the open calculus of constructions which is presented in Chapter 8.

In addition to the conceptual unification, our representations in rewriting logic provide a practical solution for the analysis and execution of a variety of Petri net models. A rewrite engine such as Maude can be directly used for strategy-guided analysis, where strategies can range from partial state space exploration to complete model checking. Furthermore, our representation can be used as a basis for formal verification, and hence constitutes an interesting application of the open calculus of constructions that will be discussed in Chapter 8.

In summary, our representations of Petri nets not only demonstrate the flexibility of rewriting logic as a semantic framework, but they also capture a very typical representative of the class of concurrency models which can be represented by (variants of) multiset rewriting. Conversely, the representations in rewriting logic suggest a number of extensions of Petri net models such as nets where the state space is not a pure multiset, or nets where tokens exhibit individual activity. In other words, the connection between rewriting logic and Petri nets is truly bidirectional, and we expect that our work is just the beginning of a line of research which takes advantage of this close connection.

Chapter 4

Metalogical Reasoning in the Calculus of Inductive Constructions: A Formalized Generalization of UNITY

Many formalisms in computer science can be presented in terms of equational inductive definitions, which in many cases can be expressed in membership equational logic or in rewriting logic. A certain class of inductive definitions [CPM90] is supported by the calculus of inductive constructions [BBC⁺99], an extension of the calculus of constructions which has inductive types as a primitive concept. In this chapter we explore the capabilities of this logical type theory as a higher-order logical framework and for metalogical reasoning, i.e. inductive reasoning about the formal systems represented, an application that we also address with the open calculus of constructions in Chapter 8.

Our object logic is a UNITY-style temporal logic for labeled transition systems, which we embed into the calculus of inductive constructions in a shallow way that avoids unnecessary syntactic overhead and simultaneously establishes a very tight, and therefore practically useful, connection between the object logic and the metalogic. Our formal development does not only subsume the original UNITY approach [CM88] to program verification and the more recent approach of New UNITY [Mis94, Mis95], and our earlier formalization in [Ste98], but goes beyond it in several essential aspects, such as the generality of the program/system model, the notion of fairness, and the issue of compositionality. The last aspect does not

only exploit the strong inductive reasoning capabilities of the calculus of inductive constructions, but it also uses the propositions-as-types interpretation and the associated proofs-as-objects interpretation in an essential way, which relates our results to classical work on interference-free proofs for parallel programs.

Our formal development has been conducted using the COQ proof development system [BBC⁺99], and the result is a verified temporal logic library for COQ that we made publicly available. This library can serve as an extensible toolbox for system and program verification. It can be instantiated for various system models, such as concurrent programs, (high-level) Petri nets, and more generally to all formalisms that can be specified as labeled transition systems.

On the other hand, we have learned from the development conducted in this chapter about fundamental limitations of type theories in the line of the calculus of constructions that are addressed by the unified formalism proposed in Chapter 8. Furthermore, since rewriting logic can be regarded as a specification language for labeled transition systems, a natural application of our temporal library is reasoning about rewriting logic specifications. In Chapter 8 we use our representation of high-level Petri nets of Chapter 3 to give an idea about how this can be done in our proposed unification of equational logic, rewriting logic and type theory, the advantage being that the execution of the program or system model and its verification can take place in a single language with additional benefits of partial automation in interactive theorem proving.

Chapter 5

First-Order Representations of Higher-Order Formalisms: A Calculus of Names and Substitutions

In this chapter we develop a systematic approach to the representation of higher-order languages in a first-order framework such as membership equational logic or rewriting logic. It is well understood how the syntax of first-order languages can be naturally specified as an algebraic data type and how their semantics can be specified using equations and/or rules in rewriting logic (see e.g. [MMO95, MOM96, GM96a]). For the representation of languages with an operational semantics, such as programming languages, we can often use the executable sub-language of rewriting logic to obtain executable specifications.

The main difficulty with the representation of higher-order languages, however, is to capture the binding structure of their syntax and to define a suitable notion of substitution which respects this structure. As witnessed by the large body of research connected with this subject, the concepts of binding and substitution are by no means trivial. This experience is also expressed by Abelson and Sussman [AS86]: “Despite the fact that substitution is a ‘straightforward idea’, it turns out to be surprisingly complicated to give a rigorous mathematical definition of the substitution process ... Indeed, there is a long history of erroneous definitions

of substitution in the literature of logic and programming semantics.”

In this chapter (see also [Ste00a]) we develop a calculus of names and explicit substitutions that makes use of a term representation with explicit names proposed by Berkling in the context of the λ -calculus [Ber76, BF82] and generalizes an existing substitution calculus for the λ -calculus by Lescanne [Les94] in an essential way. We refer to our calculus as *CINNI*, which stands for *Calculus of Indexed Names and Named Indices* and is motivated by its unusual term representation. In contrast to most calculi studied in the literature, CINNI is a completely generic calculus of explicit substitutions in the sense that it can be instantiated to the syntax of arbitrary object languages. In the theoretical part of the chapter we prove a number of metatheoretic properties of CINNI such as strong normalization and preservation of confluence under certain extensions, which are important to capture the computation rules of the object languages represented.

Furthermore, using Maude we give a number of applications of CINNI, namely the representation of the untyped λ -calculus and the representation of Abadi and Cardelli’s object calculus [AC96], also called the ζ -calculus, which can be regarded as a core calculus for object-oriented programming just as the λ -calculus is a core calculus for functional programming. We furthermore discuss how a CINNI-based representation of Milner’s π -calculus [Mil99] for communicating and mobile systems can be obtained. A real-world application of CINNI, that is based on joint work with Carolyn Talcott, is briefly presented in Appendix B, namely the specification of PLAN, an active network programming language [HKM⁺98b], which has imperative features and a notion of remote function execution. It is noteworthy that all these representations are executable and hence provide prototype execution environments for their languages. In the last application, the specification is part of a specification of an active internetwork architecture, which supports dynamically-invoked concurrent execution environments at different network locations.

Further substantial applications of CINNI in the context of logical type theories are given by our representations of pure type systems and by the open calculus of constructions, formalisms which are presented in the following two chapters. In fact, these applications constituted the original motivation for the development of the CINNI calculus.

Chapter 6

Rewriting Logic as a Logical Framework: Representing Pure Type Systems

In Chapters 3 and 5 we have further investigated the use of rewriting logic as a semantic framework [Mes92, Mes96, MOM96] for concurrent systems and programs of very different kinds. In this chapter we use rewriting logic as a logical

framework [MOM94, MOM96] to represent an important family of logical type theories, which are higher-order logics by virtue of the propositions-as-types interpretation [Geu93].

The type theories that we treat in this chapter belong to the class of pure type systems [Ber88, Ter89], a well-established scheme for type theories with simple and dependent function types, which includes important systems such as Church's simple type theory [Chu40], the systems F [Gir72, Rey74] and $F\omega$ [Gir72], the logical framework LF [HHP87], and the calculus of constructions [CH88].

In the body of this chapter, which is a continuation of our previous work [Ste99, SM99], we give a number of related first-order representations of pure type systems in membership equational logic and rewriting logic. Our representations of pure type systems range from high-level specifications, which abstract from naming issues, to specifications which take names seriously and are efficiently executable. Furthermore, the executable specifications are related to the high-level specifications in a very direct way, which makes soundness an almost trivial property.

In order to obtain suitable first-order representations we further develop the theory of pure type systems by introducing so-called *uniform pure type systems*, which use the CINNI calculus to solve the problem of closure under α -conversion in a very elegant way. This problem, which is fundamental for the implementation of type theories with dependent types, was first pointed out by Pollack [Pol93] but to our knowledge has not found a satisfactory solution so far.

To relate different versions of pure type systems among themselves and with their representations in membership equational logic or rewriting logic we uniformly use the concepts of entailment system, and correspondence, that we defined as a generalization of a map between entailment systems. In other words, we use and further develop the general methodology in [Mes89a, MOM94], namely the use of an abstract logical metatheory, namely the theory of general logics, together with rewriting logic as a particular logical framework.

The representation of pure type systems in this chapter constitutes the first systematic case study on the use of rewriting logic as a logical framework for higher-order type theories. On the practical side we have obtained an executable Maude specification which can be instantiated to obtain an important class of pure type systems. In Appendix A we instantiate our specification to the calculus of constructions extended by a universe hierarchy and demonstrate its use in the context of proof validation, using proofs that have been generated by the LEGO proof assistant [Pol94]. Another application of the techniques presented in this chapter is the experimental prototype of the open calculus of constructions in rewriting logic, which has been developed in the context of Chapter 8, and has a formal system that is based on the CINNI calculus in complete analogy to the use of

CINNI in uniform pure type systems.

Chapter 7

Classical Logic via Type Theory:

A Proof-Theoretic Approach to the HOL/Nuprl Connection

In spite of the constructive nature of most logical type theories, classical logic is the prevalent logic in mathematics and computer science, and it is in particular the logic used in this thesis. The UNITY methodology, for instance, is based on classical logic, and our formal development in Chapter 4 uses the calculus of inductive constructions together with classical axioms. Hence, we considered it important to explore the possibilities for classical reasoning in type theories, and we have found that much can be learned from the HOL/Nuprl connection in this respect.

The HOL/Nuprl connection [How96a, How98a] has originally been proposed and implemented by Doug Howe as a practical means of addressing the problem of formal interoperability between two well-known theorem provers of very different kinds, namely HOL, which is based on a classical higher-order logic, and Nuprl, which is based on an intuitionistic type theory. The core of the HOL/Nuprl connection is a translation between HOL theories and Nuprl theories, where the target logic is actually a variant of Nuprl extended by a classical axiom. Howe's main theoretical contribution was the development of a hybrid computational/set-theoretic semantics [How97] for this classical variant of Nuprl.

In this chapter we reinvestigate the HOL/Nuprl connection from a proof-theoretic point of view. We explain the main part of the HOL/Nuprl connection as a map between the entailment systems of HOL and the classical variant of Nuprl. Our soundness result nicely complements Howe's semantic justification for the HOL/Nuprl connection and opens new applications such as the translation of proofs, which has been further investigated in joint work with Naumov [NSM01].

From a more general perspective we consider the HOL/Nuprl connection as a case study in the context of formal interoperability between higher-order logic proof assistants, where we further explore the general methodology [Mes89a, MOM94] based on the use of an abstract logical metatheory and a particular logical framework. In fact, an executable formal specification which has a functionality similar to Howe's original translator has been specified using Maude. In this case study of Maude as a formal metatool [CDE⁺99b], membership equational logic is employed as a metalogic that is used to give a formal and executable specification of the logic translator.

Another line of practical work which is beyond the original HOL/Nuprl connection emerged from an application of our proof-theoretic approach. Due to its constructive nature, the computational content of our soundness proof can

be regarded as an algorithm that translates proofs. Based on this observation, Naumov has implemented a proof-translator on top of the Nuprl system that we describe in [NSM01] and briefly discuss in this chapter, since it constitutes an interesting practical application of our results.

Apart from the general aspects of logic translation and the use of membership equational logic as an executable metalogic there is a more specific issue, which makes the HOL/Nuprl connection interesting in the context of this thesis, namely that we can learn from it how an extensional and an intensional logic can coexist in a type theory with dependent types such as Nuprl. Since the method applies to the open calculus of constructions (Chapter 8) as well, it shows how classical reasoning is possible in purely predicative instances of OCC and especially without giving up the potential intensionality of its propositional universes.

Chapter 8

Towards a Unified Language: The Open Calculus of Constructions

Altogether, the applications in the previous chapters suggest that rewriting logic, including its membership equational sublogic, and logical type theories, with their internal higher-order logics, have features that are nearly complementary, so that much could be gained from a unification of these two lines of research.

As an example, rewriting logic can serve as a framework logic [MOM94, Mes96, MOM96], and higher-order logic can be used to formulate induction principles and to reason about the formalism that has been embedded into the framework. Furthermore, rewriting logic could benefit from an expressive type system with dependent types and universes, which in particular provides type operators and explicit polymorphism. Conversely, type theory could benefit from equational logic and rewriting logic. For instance, the lack of a notion of executable specification, providing a notion of abstract execution, is a clear weakness of logical type theories, which in turn leads to severe difficulties with principles of modularity and information hiding. Furthermore, the notion of computation in type theories is rather rigid, compared with the powerful notion of computation based on conditional rewriting modulo the axioms of an equational theory as in rewriting logic.

In this chapter we develop a unified formalism that we call the *open calculus of constructions*. The calculus is parameterized by a flexible universe hierarchy, which admits impredicative universes in the style of the calculus of constructions and predicative universes in the style of Martin-Löf's type theory. Similar to the calculus of inductive constructions, it is inspired by the extended calculus of constructions [Luo94], but has more general computational capabilities. Specifically, the open calculus of constructions incorporates membership equational logic and the corresponding version of rewriting logic as computational sublanguages, and

as a consequence supports conditional assertions, equations, and rules together with an operational semantics based on conditional rewriting modulo equational theories.

A unification of membership equational logic and the calculus of constructions has already been proposed as a long-term goal in [Jou98], but the motivations are different. The formalism proposed in [BJO99], the algebraic calculus of constructions, can be seen as an important subsystem of our very liberal approach, namely a subsystem where the strong normalization property is enforced by syntactic means. Our approach can be seen as a higher-order generalization of executable specifications in membership equational logic, and strong normalization is often a desirable but not a mandatory property. In this sense our formalism is less restrictive than the calculus of constructions and extensions of it such as the calculus of inductive constructions and the calculus of algebraic constructions which maintain strong normalization as a property of key importance, on which the standard consistency proof of the type theory as a logical system is ultimately based. The consistency of the open calculus of constructions on the other hand is justified using a classical set-theoretic semantics and does not rely on operational properties.

On the practical side, we have used rewriting logic, and hence a sublanguage of the new formalism itself, as an executable logical framework to obtain an experimental implementation of the new formalism. The result is a Maude prototype of the open calculus of constructions, which serves as an execution environment for specifications in the unified formalism and as a higher-order logic proof development system. In summary, the open calculus of constructions offers a general higher-order framework for specification, programming, and interactive theorem proving, which, due to the flexibility of its underlying equational/rewriting logic, is widely applicable in various domains. Several examples of applications that are connected with earlier chapters of this thesis have been developed using the prototype and are also presented in this chapter.

Chapter 2

Preliminaries

2.1 Notation and Basic Concepts

In this thesis we make informal use of set theory and employ the standard notations for sets and some additional notation that we explain below. On top of set theory, we use a modest amount of category theory in this thesis for which we mainly follow [BW90].

Given a set S we denote the set of sets over S , i.e. its *powerset*, by $\mathcal{PS}(S)$. *Functions* are defined as binary *relations* in the standard way, and as usual we write $S \rightarrow T$ for the set-theoretic *function space*, which is the set of all *total functions* from S to T . Similarly, we write $S \rightsquigarrow T$ to denote the set of all *partial functions* from S to T . We use $\text{Ran}(f)$ and $\text{Dom}(f)$ to refer to the *domain* and the *range* of a function f . We also use the notation $\lambda x \in S. t(x)$ borrowed from λ -calculus to denote the total function f with domain S defined by $f(x) = t(x)$ for all $x \in S$. We furthermore define the set-theoretic *dependent function space* $\Pi x \in S. T(x)$ to denote the set of all functions f with domain S with the property that $f(x) \in T(x)$ for all $x \in S$. Finally, we will use the set-theoretic *dependent sum* $\Sigma x \in S. T(x)$ to denote the set of pairs (x, y) satisfying $x \in S$ and $y \in T(x)$.

A *sequence* over a set S is a function from \mathbb{N} to S , in which case we have an *infinite sequence*, or a function from an interval $\{1, \dots, n\}$ of \mathbb{N} to S , in which case we have a *finite sequence*, which is also called a *list* of length n . We denote by $\mathcal{SEQ}(S)$ and $\mathcal{LST}(S)$ the set of sequences and lists over S , respectively. The set of *indices* of a sequence s is denoted by $\text{Ind}(s) = \text{Dom}(s)$, and the *length* of a list l is denoted by $|l|$. If l is a list of length n we sometimes write l^n instead of l to make the length explicit. For $i \in \text{Ind}(l)$ the i -th element of l is denoted by l_i . *Concatenation* of lists l and l' is written as l, l' . A list l of length n is written as $[l_1, \dots, l_n]$, a notation that reduces to $[]$ for the *empty list*. We sometimes write x instead of the *singleton list* $[x]$, so that we have $[l_1, \dots, l_n] = [l_1], \dots, [l_n] = l_1, \dots, l_n$. For convenience, we occasionally identify tuples (l_1, \dots, l_n) and lists $[l_1, \dots, l_n]$. Finally, we use the *prefix ordering* as a standard partial order on sequences.

A *multiset* over a set S is a function from S to \mathbb{N} . A multiset M over S is *finite* iff its *support* $\text{Supp}(M) = \{x \in S \mid M(x) > 0\}$ is finite. We denote by $\mathcal{MS}(S)$ and $\mathcal{FMS}(S)$ the set of multisets and finite multisets, respectively, over S , and overloading the notation for sets, we write \emptyset_S , often omitting S , for the empty multiset over S , $\{x_1, \dots, x_n\}$ for the multiset which contains precisely the elements x_1, \dots, x_n with possible repetitions, \oplus for *multiset union*, $-$ for *multiset difference*, \in for *multiset membership* and \subseteq for *multiset inclusion*.

Furthermore, we implicitly lift a function $f : X \rightarrow Y$ to sets, multisets, and sequences in the natural homomorphic way, giving rise to functions $f : \mathcal{PS}(X) \rightarrow \mathcal{PS}(Y)$, $f : \mathcal{MS}(X) \rightarrow \mathcal{MS}(Y)$, and $f : \mathcal{SEQ}(X) \rightarrow \mathcal{SEQ}(Y)$.

2.2 Transition Systems

In this thesis we are concerned with system specification formalisms such as functional, imperative, object-oriented, and concurrent programming languages, process algebras, Petri nets, rewriting logic, and temporal logic. The systems that we wish to specify exhibit a variety of different aspects such as (non)determinism, (non)termination, reactivity, concurrency, distribution, and mobility.

Labeled transition systems describe such systems in terms of *states* and possible *transitions* between states which are labeled by *actions*. Both states and actions provide possible means of interaction and communication between the system and its environment. Labeled transition systems provide a very general and abstract semantic model of such *reactive and computational systems*, and can be naturally extended to richer semantic models such as different classes of transition categories as explained at the end of this section. Labelled transition systems can be used to model (open) reactive systems which are observable at any state and interact with the environment, but also (closed) computational systems, such as functional programs, which are typically observed after reaching a final state.

A *labeled transition system* \mathcal{TS} consists of a set $St_{\mathcal{TS}}$ of *states*, a set $Act_{\mathcal{TS}}$ of *actions*, and a *transition relation* $\xrightarrow{e}_{\mathcal{TS}} \subseteq St_{\mathcal{TS}} \times St_{\mathcal{TS}}$ for each $e \in Act_{\mathcal{TS}}$. Whenever $s \xrightarrow{e}_{\mathcal{TS}} s'$ we say that e is *enabled* in s and that s' is a *successor state* of s (under e). A state s is *final* iff no action is enabled in s . A labeled transition system \mathcal{TS} is *deterministic* iff for each $e \in Act_{\mathcal{TS}}$ and $s \in St$ there is at most one $s' \in St_{\mathcal{TS}}$ with $s \xrightarrow{e}_{\mathcal{TS}} s'$; otherwise \mathcal{TS} is *nondeterministic*.

Given a labeled transition system \mathcal{TS} , an *execution* is a finite or infinite alternating sequence of states and actions that is either of the form $s_1, e_1, s_2, e_2, \dots, s_{n-1}, e_{n-1}, s_n$ or $s_1, e_1, s_2, e_2, \dots$ such that $s_i \xrightarrow{e_i}_{\mathcal{TS}} s_{i+1}$ for all indices $i, i+1$ of s . Given such an execution we say that s_1 is its *origin*, and s_n is its *destination*, also called its *result*. A state s is *weakly terminating* iff at least one maximal execution with origin s is finite, and it is (*strongly*) *terminating* iff all maximal executions with origin s are finite. A transition system is *weakly/strongly terminating* iff all its states are *weakly/strongly terminating*.

This notion of execution is the most liberal notion of execution that can be defined for labeled transition systems. In concrete applications, the set of admissible executions often has to satisfy additional properties, such as certain fairness requirements, and/or is constrained, e.g. due to interaction with the environment or by virtue of a particular execution strategy.

Usually, we do not distinguish between transition systems that are isomorphic, where our notion of morphism is the following. Given labeled transition systems \mathcal{TS} and \mathcal{TS}' , a morphism $h : \mathcal{TS} \rightarrow \mathcal{TS}'$ consists of functions $h_{St} : St_{\mathcal{TS}} \rightarrow St_{\mathcal{TS}'}$ and $h_{Act} : Act_{\mathcal{TS}} \rightarrow Act_{\mathcal{TS}'}$ such that $s \xrightarrow{e}_{\mathcal{TS}} s'$ implies $h_{St}(s) \xrightarrow{h_{Act}(e)}_{\mathcal{TS}'} h_{St}(s')$. Labeled transition systems together with their morphisms form a category **LTSys**

and transition systems form a subcategory \mathbf{TSys} by the obvious inclusion, which equips all transitions with the same label, say \bullet .

Labeled transition systems are a quite abstract model of reactive and computational systems, since the states and the actions are viewed as sets without internal structure. This abstract point of view makes them fairly general, covering a range of possibilities from actions that are atomic events occurring in a global state space to actions that are structured processes taking place in a distributed state space. Mathematically, it is often useful to complete the set of actions by considering identity actions, which do not affect the state, and actions which are obtained by sequential composition of other actions. Such closure properties are naturally expressed by *transition categories*, i.e. categories where the objects are interpreted as states and the arrows are interpreted as transitions.

In a more specific setting of distributed concurrent systems the states and the actions usually have additional structure corresponding to parallel composition operations. As a typical example they may form a commutative monoid which is often used to model a distributed state space, e.g. in process algebras, Petri nets and concurrent object systems. This is made explicit in *enriched transition categories*, i.e. transition categories with additional algebraic structure (beyond identities and sequential composition) on the objects and arrows. Such enriched transition categories are used in the model-theoretic semantics of rewriting logic (cf. Section 2.4.5 and see [Mes92] for a general definition). In this thesis, we furthermore use different kinds of enriched transition categories as reference models for classes of Petri nets (cf. Chapter 3).

2.3 Entailment Systems

Throughout this thesis we are concerned with a variety of formal systems which can be viewed as *entailment systems*, a notion defined in [Mes89a] as a main component of general logics. A *general logic* consists of an entailment system that covers the *syntactic aspect of derivability* (from a set of axioms), given by a relation \vdash , and an *institution* [GB92, GB94] that covers the *semantic aspect of validity* (in a given model), expressed by a relation \models . An essential property of a general logic is soundness, i.e., every sentence that is derivable in the formal system is semantically valid.

The following definition of an entailment system is fairly general, since it presupposes only a category of signatures \mathbf{Sign} and a uniform method to generate sentences over a signature Σ from \mathbf{Sign} . Also it only makes minimal assumptions about the entailment relation \vdash . The definition is taken from [Mes89a], but we are employing a slight modification, which has already been used in the context of institutions [GB86], by requiring a functor $\mathbf{Sen} : \mathbf{Sign} \rightarrow \mathbf{Cat}$ instead of $\mathbf{Sen} : \mathbf{Sign} \rightarrow \mathbf{Set}$. This generalization is useful in the case where the sentences are not

just sets but carry a non-trivial categorical structure expressed by morphisms between sentences. We will exploit this generality in Chapter 7 of this thesis, where sentences are sequents that can be related to each other by means of morphisms. Below and in other parts of this thesis we employ the convention that given a category \mathbf{C} its set of objects is also denoted by \mathbf{C} .

A (binary) entailment system $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ consists of

1. a category \mathbf{Sign} of signatures,
2. a functor $\mathbf{Sen} : \mathbf{Sign} \rightarrow \mathbf{Cat}$,
associating a category of *sentences* $\mathbf{Sen}(\Sigma)$ with each signature Σ ,
3. a family of binary *entailment relations* $(\vdash_\Sigma)_{\Sigma \in \mathbf{Sign}}$ with $\vdash_\Sigma \subseteq \mathcal{PS}(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma)$ such that the following conditions are satisfied:
 - (a) *Reflexivity*: for any $\phi \in \mathbf{Sen}(\Sigma)$, $\{\phi\} \vdash_\Sigma \phi$;
 - (b) *Monotonicity*: if $\Gamma \vdash_\Sigma \phi$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash_\Sigma \phi$.
 - (c) *Transitivity*: if $\Gamma \vdash_\Sigma \phi_i$, for all $i \in I$, and $\Gamma \cup \{\phi_i \mid i \in I\} \vdash_\Sigma \phi$ then $\Gamma \vdash_\Sigma \phi$.
 - (d) *Translation*: if $\Gamma \vdash_\Sigma \phi$ then $H(\Gamma) \vdash_{\Sigma'} H(\phi)$
for any signature morphism $H : \Sigma \rightarrow \Sigma'$ in \mathbf{Sign} .

Given such an entailment system we define the deductive closure of a set of sentences Γ over Σ by $\Gamma^\bullet = \{\phi \mid \Gamma \vdash_\Sigma \phi\}$.

Let $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ be an entailment system. A *theory* $\mathcal{E} = (\Sigma, \Gamma)$ consists of a signature Σ together with a set $\Gamma \subseteq \mathbf{Sen}(\Sigma)$, called the set of *axioms* of \mathcal{E} . A signature morphism $H : \Sigma \rightarrow \Sigma'$ is said to be a *theory morphism* $H : (\Sigma, \Gamma) \rightarrow (\Sigma', \Gamma')$ iff $\Gamma' \vdash_{\Sigma'} \mathbf{Sen}(H)(\phi)$ for all $\phi \in \Gamma$. H is called *axiom-preserving* iff $\mathbf{Sen}(H)(\Gamma) \subseteq \Gamma'$. The class of theories together with theory morphisms constitutes a category denoted by \mathbf{Th} . The subcategory of \mathbf{Th} with the same objects but morphisms restricted to axiom-preserving morphisms is denoted by \mathbf{Th}_0 .

The family of entailment relations $(\vdash_\Sigma)_{\Sigma \in \mathbf{Sign}}$ is lifted from signatures to theories giving a family $(\vdash_{(\Sigma, \Gamma)})_{(\Sigma, \Gamma) \in \mathbf{Th}_0}$ with $\vdash_{(\Sigma, \Gamma)} \subseteq \mathcal{PS}(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma)$ defined by $\Gamma' \vdash_{(\Sigma, \Gamma)} \phi$ iff $\Gamma \cup \Gamma' \vdash_\Sigma \phi$.

Unary Entailment Systems

Often the notion of entailment system is more general than necessary. For instance, in sequent style presentations of logics and type theories the sentences are of the form $\Gamma \vdash \phi$, where \vdash is a constructor for sentences rather than a relation. In such sequent style presentations the notion of relative derivability from axioms (often called hypotheses or context) is internalized, and hence it

is often sufficient to consider derivability without (external) axioms. Thus as a special case of of entailment systems we introduce *unary* entailment systems as follows.

A *unary entailment system* $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ consists of

1. a category \mathbf{Sign} of signatures,
2. a functor $\mathbf{Sen} : \mathbf{Sign} \rightarrow \mathbf{Cat}$,
associating a set of *sentences* $\mathbf{Sen}(\Sigma)$ to each signature Σ ,
3. a family of unary *entailment predicates* $(\vdash_{\Sigma})_{\Sigma \in \mathbf{Sign}}$
with $\vdash_{\Sigma} \subseteq \mathcal{PS}(\mathbf{Sen}(\Sigma))$, such that the *translation condition*

$$\vdash_{\Sigma} \phi \text{ implies } \vdash_{\Sigma'} H(\phi)$$

holds for any signature morphism $H : \Sigma \rightarrow \Sigma'$ in \mathbf{Sign} .

With each binary entailment system $(\mathbf{Sign}, \mathbf{Sen}, \vdash')$ we can associate a unary entailment system $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ by requiring $\vdash \phi$ iff $\emptyset \vdash' \phi$. Conversely, each unary entailment system $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ generates a binary entailment system $(\mathbf{Sign}, \mathbf{Sen}, \vdash')$, namely the smallest one that satisfies $\emptyset \vdash' \phi$ whenever $\vdash \phi$ holds. In this case we have $\Gamma \vdash' \phi$ iff $\phi \in \Gamma$ or $\vdash \phi$. This correspondence justifies the practice of viewing a unary entailment system as a binary entailment system and vice versa.

Maps of Entailment Systems

Subsequently, we present the definition of maps of entailments systems closely following [Mes89a]. The main application of maps of entailment systems is to capture a notion of *translation between sentences* of two formal systems in a way that preserves derivability in the direction of the mapping.

Let $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ and $(\mathbf{Sign}', \mathbf{Sen}', \vdash')$ be entailment systems. We require that a map of sentences should map a signature of \mathbf{Sign} to a signature of \mathbf{Sign}' and that signature morphisms are preserved. This is expressed by a functor $\Phi : \mathbf{Sign} \rightarrow \mathbf{Sign}'$. Moreover, a map of sentences should map sentences over \mathbf{Sign} to sentences over \mathbf{Sign}' in a way that is compatible with the mapping of signatures. Hence, in addition to Φ , another component of a map of sentences is a natural transformation $\alpha : \mathbf{Sen} \rightarrow \mathbf{Sen}' \circ \Phi$, that associates with each sentence ϕ over Σ a sentence $\alpha_{\Sigma}(\phi)$ over $\Phi(\Sigma)$.

A *map of sentences* (Φ, α) between entailment systems $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ and $(\mathbf{Sign}', \mathbf{Sen}', \vdash')$ consists of a functor $\Phi : \mathbf{Sign} \rightarrow \mathbf{Sign}'$ and a natural transformation $\alpha : \mathbf{Sen} \rightarrow \mathbf{Sen}' \circ \Phi$. Given such a map of sentences (Φ, α) , we assume that α has been extended to sets of sentences in the obvious way.

Maps of sentences only relate entailment systems at the level of syntax. We now extend this definition to maps of entailment systems that preserve the entailment relation, i.e. the logical structure available at this level of abstraction. In practice, it often turns out that in order to preserve the logical structure the information expressed by a signature of one theory requires the use of axioms in another theory. To capture such cases we assume that the functor $\Phi : \mathbf{Sign} \rightarrow \mathbf{Sign}'$ is the signature part of a functor $\Phi : \mathbf{Sign} \rightarrow \mathbf{Th}'_0$ that maps each signature Σ to a theory (Σ', Γ') .

Next we observe that the functor $\Phi : \mathbf{Sign} \rightarrow \mathbf{Th}'_0$ can be extended to a functor $\Phi : \mathbf{Th}_0 \rightarrow \mathbf{Th}'_0$, called the α -extension to theories, by mapping a theory (Σ, Γ) to the theory (Σ', Γ') with $(\Sigma', \emptyset') = \Phi(\Sigma)$ and $\Gamma' = \emptyset' \cup \alpha_\Sigma(\Gamma)$. Such a functor $\Phi : \mathbf{Th}_0 \rightarrow \mathbf{Th}'_0$ is called an α -simple functor, i.e., it is the α -extension to theories of a functor from signatures to theories. There are cases where axioms of the target logic are not used in the translation of a signature. In such a case $\Phi : \mathbf{Th}_0 \rightarrow \mathbf{Th}'_0$ is uniquely determined by $\Phi : \mathbf{Sign} \rightarrow \mathbf{Sign}'$ alone and we say that Φ is α -plain, i.e., it is the α -extension of a functor $\Phi : \mathbf{Sign} \rightarrow \mathbf{Th}'_0$ that sends each signature Σ to a theory (Σ', \emptyset) .

We extend the functors $\mathbf{Sen} : \mathbf{Sign} \rightarrow \mathbf{Cat}$ and $\mathbf{Sen}' : \mathbf{Sign}' \rightarrow \mathbf{Cat}$ to functors $\mathbf{Sen} : \mathbf{Th}_0 \rightarrow \mathbf{Cat}$ and $\mathbf{Sen}' : \mathbf{Th}'_0 \rightarrow \mathbf{Cat}$ by just ignoring the axioms in the theories. Correspondingly, we extend the natural transformation $\alpha : \mathbf{Sen} \rightarrow \mathbf{Sen}' \circ \Phi$ with $\mathbf{Sen} : \mathbf{Sign} \rightarrow \mathbf{Cat}$ and $\mathbf{Sen}' : \mathbf{Sign}' \rightarrow \mathbf{Cat}$ to a natural transformation between $\mathbf{Sen} : \mathbf{Th}_0 \rightarrow \mathbf{Cat}$ and $\mathbf{Sen}' \circ \Phi : \mathbf{Th}_0 \rightarrow \mathbf{Cat}$ by setting $\alpha_{(\Sigma, \Gamma)} = \alpha_\Sigma$.

Now a map of entailment systems could be defined as a map of sentences (Φ, α) with $\Gamma' = \emptyset' \cup \alpha_\Sigma(\Gamma)$ whenever $\Phi(\Sigma, \emptyset) = (\Sigma', \emptyset')$ and $\Phi(\Sigma, \Gamma) = (\Sigma', \Gamma')$ such that α preserves entailment. The actual definition in [Mes89a], however, uses α -sensitivity, a less restrictive requirement defined next.

A functor $\Phi : \mathbf{Th}_0 \rightarrow \mathbf{Th}'_0$ is called α -sensible iff

1. There is a functor $\Phi^\diamond : \mathbf{Sign} \rightarrow \mathbf{Sign}'$ such that $Sign' \circ \Phi = \Phi^\diamond \circ Sign$, where $Sign : \mathbf{Th}_0 \rightarrow \mathbf{Sign}$ and $Sign' : \mathbf{Th}'_0 \rightarrow \mathbf{Sign}'$ are the obvious forgetful functors.
2. $\Gamma'^\bullet = (\emptyset' \cup \alpha_\Sigma(\Gamma))^\bullet$ whenever $\Phi(\Sigma, \emptyset) = (\Sigma', \emptyset')$ and $\Phi(\Sigma, \Gamma) = (\Sigma', \Gamma')$.

$(\Phi, \alpha) : (\mathbf{Sign}, \mathbf{Sen}, \vdash) \rightarrow (\mathbf{Sign}', \mathbf{Sen}', \vdash')$ is a *map of entailment systems* iff the following conditions are satisfied:

1. $\alpha : \mathbf{Sen} \rightarrow \mathbf{Sen}' \circ \Phi$ is a natural transformation, where $\mathbf{Sen} : \mathbf{Th}_0 \rightarrow \mathbf{Cat}$, $\Phi : \mathbf{Th}_0 \rightarrow \mathbf{Th}'_0$, and $\mathbf{Sen}' : \mathbf{Th}'_0 \rightarrow \mathbf{Cat}$.
2. $\Phi : \mathbf{Th}_0 \rightarrow \mathbf{Th}'_0$ is α -sensible,

3. (Φ, α) is *sound*, i.e.,
 $\Gamma \vdash_{\Sigma} \phi$ implies $\alpha_{\Sigma}(\Gamma) \vdash'_{\Phi(\Sigma)} \alpha_{\Sigma}(\phi)$ for each signature $\Sigma \in \mathbf{Sign}$.

Note that $\vdash'_{\Phi(\Sigma)}$ refers to the entailment relation relative to the theory $\Phi(\Sigma)$, i.e., we use the entailment relation lifted from signatures to theories here.

We say that (Φ, α) is *conservative* iff in addition it is *complete*, i.e.

$$\alpha_{\Sigma}(\Gamma) \vdash'_{\Phi(\Sigma)} \alpha_{\Sigma}(\phi) \text{ implies } \Gamma \vdash_{\Sigma} \phi \text{ for every signature } \Sigma \in \mathbf{Sign}.$$

Furthermore, (Φ, α) is *plain* iff Φ is α -plain and (Φ, α) is *simple* iff Φ is α -simple.

Entailment systems together with maps of entailment systems form a category denoted by **Ent**.

Correspondences

In the previous section we introduced maps between sentences to capture a notion of *translation between formal systems*, which satisfies the associated soundness property that derivability is preserved in the direction of the mapping. Below we introduce the notion of correspondence between sentences of different entailment systems. In contrast, to the above maps the main application of correspondences is to capture a notion of *refinement between formal systems*, where the soundness property is defined as the preservation of derivability in the opposite direction of the correspondence.

Often a correspondence takes the form of a function, but as we will see there are interesting cases where the more general concept is useful. This is the case when there is not a single canonical way to translate a given sentence into a sentence of the target logic, a situation that we encounter in Chapter 6. We first introduced such correspondences in the context of [SM99] to meet a specific purpose, but we think they are important enough to be included in the vocabulary of general logics. On the other hand, in this thesis we do not need the notion of correspondence in full generality but only for unary entailment systems over a fixed signature, i.e., we do not consider signature translation in the following. Hence, we specialize unary entailment systems $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ to unary entailment systems (\mathbf{Sen}, \vdash) where **Sen** is a set.

Let (\mathbf{Sen}, \vdash) and (\mathbf{Sen}', \vdash') be unary entailment systems. A *correspondence of sentences* between (\mathbf{Sen}, \vdash) and (\mathbf{Sen}', \vdash') is a relation $\curvearrowright \subseteq \mathbf{Sen} \times \mathbf{Sen}'$. A correspondence \curvearrowright is called *total* iff for each $\phi \in \mathbf{Sen}$ there is a ϕ' such that $\phi \curvearrowright \phi'$. Furthermore, a correspondence \curvearrowright is said to be *sound* iff $\vdash' \phi'$ implies $\vdash \phi$ for all $\phi \curvearrowright \phi'$, and \curvearrowright is said to be *complete* iff $\vdash \phi$ implies $\vdash' \phi'$ for all $\phi \curvearrowright \phi'$. Unary entailment systems together with correspondences of sentences form a category denoted by **CEnt**.

A correspondence of sentences $\curvearrowright \subseteq \mathbf{Sen} \times \mathbf{Sen}'$ can take the form of a function

$\alpha : \mathbf{Sen}' \longrightarrow \mathbf{Sen}$ in the opposite direction, in which case we have a map of sentences. In our simplified setting, a map of entailment systems $\alpha : \mathbf{Sen}' \longrightarrow \mathbf{Sen}$, gives rise to a sound correspondence $\alpha^{-1} : \mathbf{Sen} \longrightarrow \mathbf{Sen}'$ of sentences, and if α is furthermore a conservative map of entailment systems then α^{-1} is precisely a sound and complete correspondence of sentences. Alternatively, and in fact the more common case if we are concerned with the notion of refinement or implementation, is that a correspondence \curvearrowright takes the form of a function $\alpha : \mathbf{Sen} \longrightarrow \mathbf{Sen}'$ in the same direction. In summary, correspondences do not favor any particular direction. The only reason for using an asymmetric symbol \curvearrowright is to remind us of the direction of the implication in the soundness property.

2.4 From Equational Logic to Rewriting Logic

This section introduces equational logics as they are used in several algebraic specification languages, and more specifically introduces membership equational logic and the corresponding version of rewriting logic that we use in this thesis.

Algebraic specification languages [Wir90, Bid91, Pad88] exist in many flavours. In the broad sense of the term this category includes languages such as Clear [BG77, BG80], OBJ3 [GWM⁺92], CafeOBJ [DF98], Maude [CDE⁺99a], CASL [CoF01], ASL [SW83], ACT [CEW93], Larch/LSL [GH93], Extended ML [ST91], Spectral [KBS91], and SPECTRUM [BFG⁺93]. All these specification languages share the common feature that their specifications declare *sorts* together with *functions* operating on these sorts, and *axiomatically* specify their *abstract properties* using an underlying *logic*, which is typically a variant of Horn clause logic or first-order logic with equality. In most cases the algebraic specification language enriches the underlying logic by additional features, e.g. to express certain constraints, such as initiality or freeness, which cannot be formulated inside the logic, or to deal with issues such as modularity and parameterization.

Clear [BG77, BG80] is an early algebraic specification language that is based on many-sorted, purely equational logic. More recent languages such as OBJ3 [GWM⁺92], CafeOBJ [DF98], and Maude [CDE⁺99a, CDE⁺00b] are based on more general equational logics. Here and in the following we use the term *equational logic* in a generic way to refer to logics which can be seen as sublogics of first-order Horn clause logic with equality [Pad88, GM86]. In fact, we restrict our attention in this section to algebraic specification languages based on *equational logics*, and we refer to such languages as *equational specification languages*.

The model theory of equational specification languages is well-developed [MG85, EM85, Wec92, GB92] and appealing due to its elegant treatment in terms of category theory [EM85] generalizing universal algebra [Wec92]. Further significant developments which were originally motivated by applications in the context of equational specification languages but are in fact applicable to a wide range

of logics are the abstract model theory in terms of institutions [GB92] and the abstract framework of general logics [Mes89a] that we referred to in Section 2.3.

In contrast to the typical use of more expressive logics such as first-order or higher-order logics, equational specification languages are not only employed for purely logical specification, but they also allow us to formulate functional specifications that are logically *and* operationally meaningful in a single language. Indeed, under suitable restrictions an equational specification can be equipped with a natural operational semantics in terms of term rewriting systems [HO80, DJ90, Klo92, Pla95], a link between logic and computation which forms the basis for the equational programming paradigm [O'D85, Pad88, DP88, GM86, GM87]. Although it is beyond the subject of this thesis, it is worth to point out that equational programming admits a natural unification with logic programming by simply generalizing the operational semantics from rewriting to narrowing [GM86].

On the other hand, when we are not interested in executability but rather in a high-level specification, there is no need to constrain ourselves to an executable fragment of equational logic. In fact, many algebraic specification languages use a variant of full first-order logic as underlying logic. We defer the issue of more expressive specification languages to Chapter 8, where we combine the equational specification style with a higher-order logic based on an expressive type theory.

We begin this section with a gentle introduction to equational specifications by informally discussing central concepts such as sorts and subsorts, structural and computational equations, membership assertions, and conditional axioms. This section is furthermore intended to briefly introduce the OBJ3/Maude specification style and to motivate the specification style used in this thesis which is different from the former in a few subtle aspects. Finally, we give a more formal definition of membership equational logic and its semantics. In a similar way, we present rewriting logic over membership equational logic, again together with its semantics. Rewriting logic can be seen as both a generalization and a refinement of equational logic which is suitable for specifying reactive computational systems in a logically and operationally meaningful way.

2.4.1 Many-Sorted Algebra

Many-sorted equational logic [GM85], also called *many-sorted algebra*, is a many-sorted, purely equational logic. A *many-sorted equational theory* is given by a *many-sorted signature* Ω , that declares *sorts* Srt and *operator symbols* Op , and a set of *axioms* E which are universally quantified equations. Operator symbols are either *constant symbols* or *function symbols* depending on their *arity*.

As an example consider the following many-sorted equational theory of monoids. e denotes the identity element and $_*_$ is the binary monoid operation. The

equational axioms express that e is a left and a right identity element and that $_*_$ is associative.

```

sort S .
op e : -> S .
op *_ : S S -> S .
var x y z : S .
eq e * x = x .
eq x * e = x .
eq x * (y * z) = (x * y) * z .

```

The syntax we use is typical for algebraic specification languages. It is actually the syntax of Maude [CDE⁺99a, CDE⁺00b], which generalizes that of OBJ3 [GWM⁺92]. The keyword `sort` introduces the sorts used in the specification, and `op` declares constant and function symbols together with their arity (the sorts of their arguments, if any) and coarity (the sort of their result). The keyword `var` declares sorted variables and `eq` introduces a new equational axiom over the variables that have been declared. We have chosen $_*_$ as the name of the monoidal operator to demonstrate the use of mixfix syntax supported by OBJ3 and Maude, which allows us to write $(x * y)$ instead of $_*_(x,y)$.

The *model-theoretic semantics* of a many-sorted equational theory is a category of many-sorted algebras. A many-sorted algebra for a given signature interprets sorts as sets and operator symbols as set-theoretic elements or functions depending on their arity. The model-theoretic semantics of an equational theory with a signature Ω and axioms E is the category of algebras for Ω that satisfy the axioms in E . This semantics is also called *loose semantics*, since no further restrictions are imposed beyond those given by the axioms. In the example above the loose semantics is the category of all monoids.

As another example, consider the following theory that declares a sort `Nat` together with constructors `0` and `suc`.

```

sort Nat .

op 0 : -> Nat .
op suc : Nat -> Nat .

op 1 : -> Nat .
eq 1 = suc(0) .
op 2 : -> Nat .
eq 2 = suc(1) .
...

```

Often we are not interested in the full category of algebras which are models of the theory but only in a subcategory of algebras such as the category consisting of initial algebras, also called the *initial semantics* [MG85]. Since all initial algebras are isomorphic we may think of this category also as a single initial algebra. A natural candidate to represent this category is the quotient of the term algebra in which each sort is interpreted as the set of *terms* (without variables) of this sort and where two elements of a sort are *identified* iff they can be proved to be equal using equations E which are part of the theory. The initial algebra formalizes the idea that: (1) a sort only contains elements that can be built using the constructors for this sort, and (2) elements which are not required to be equal by the theory are distinct in the semantics. These two properties are also known under the slogan “no junk” and “no confusion”, respectively [EM85]. As an example, to ensure that the theory of `Nat` above is a faithful axiomatization of the natural numbers we require that it is interpreted under the initial semantics; otherwise we would obtain models with undesired nonstandard elements or undesired identifications.

Now consider the theory below, that declares constant symbols `a`, `b`, `c`, `d`, `e` of sort `Id` intended to represent identifiers. Again, we require that this theory is interpreted initially, which ensures that the elements `a`, `b`, `c`, `d`, `e` are distinct and that they are the only elements of `Id`. Alternatively, we might choose the loose semantics if we were to think of `Id` as an open sort of unknown extension and had chosen arbitrary elements `a`, `b`, `c`, `d`, `e` of this sort.

```

sorts Id .
ops a b c d e : -> Id .

```

Now we extend the theory of `Id` as shown below to obtain a theory for pairs of identifiers. This is a typical example of an instantiated parameterized theory. More precisely, our current theory is extended by instantiating an abstract parameterized theory¹ of pairs with an actual parameter given by the trivial theory which consists of a single sort `Id` in this case. In order to faithfully reflect the standard notion of product we have to require that for each fixed interpretation of `Id` the theory `Pair-Id` is interpreted initially. Furthermore, morphisms between different interpretations of the parameter theory should extend to corresponding interpretations of the instantiated theory in a unique way. Both of these properties are expressed by the requirement that the models of the `Pair-Id` theory

¹In most specification languages and especially in Maude, the logic is extended by language constructs to express abstract parameterized modules. In this introduction, however, we only deal with flat specifications, i.e. specification which can be derived from structured specifications by instantiation of possible parameters and by abstracting from their structure. Indeed, our goal of our treatment in Chapter 8 is to express concepts such as modules and parameterization inside the logic, thereby avoiding specific extensions for this purpose.

are *free*, we also say *freely generated*, over those of the trivial parameter theory of `Id`.

```

sorts Pair-Id .
op pair : Id Id -> Pair-Id .

```

The notions of initiality and, more generally, freeness will be made precise later in the context of membership equational logic. We consider initiality and freeness requirements as additional sentences, so-called *initiality and freeness constraints*, which restrict the category of models of the underlying theory. An equational theory which is equipped with such constraints is called an *equational specification*. We emphasize the distinction between theories and more general specifications, because theories do always have models, e.g. the initial one, whereas specifications can be contradictory, e.g. the initial specification of `Nat` together with the subsequent addition of an equation `suc(0) = 0`. The essential point is that equational theories are formulated in an equational logic, which as a positive logic is weak enough to always admit initial and free models, whereas equational specifications use strictly more expressive means, namely constraints, thereby leaving the realm of equational logic. Due positive nature of its logic, equational theories equipped with an initiality constraint can be regarded as possible formalization of a notion of equational inductive definitions, and more generally, parameterized equational theories equipped with a corresponding freeness constraint for the parameter formalize the notion of parameterized equational inductive definitions. Indeed, the close connection between initiality and induction principles has already been pointed out in [MG85], and it generalizes to the more expressive equational logics introduced later in this section.

Now we can further extend our specification of pairs, for instance by adding the standard projections:

```

var x y : Id .

op fst : Pair-Id -> Id .
eq fst(pair(x,y)) = x .

op snd : Pair-Id -> Id .
eq snd(pair(x,y)) = y .

```

Another interesting feature of equational specification languages is that their equational nature suggests a simple notion of computation. Indeed, in addition to its model-theoretic semantics, a many-sorted equational specification can be equipped with an *operational semantics* by viewing each equation $L = R$ as a *reduction rule*. To define the operational semantics we introduce the reduction

relation \rightarrow_E as a binary relation on terms. We write $C[\theta(L)] \rightarrow_E C[\theta(R)]$ if the specification contains an equation $L = R$, there is a substitution θ for its variables, and there is a context C , i.e. a term with a hole. Here $C[M]$ denotes the context with the hole replaced by M . In this case we also say that the equation $L = R$ can be *applied* to a term $C[\theta(L)]$ yielding $C[\theta(R)]$ as a result. If $M \rightarrow_E M'$ we say that M *reduces to M' in one step*. We also use the reflexive and transitive closure $\xrightarrow{*}_E$ of \rightarrow_E , and if $M \xrightarrow{*}_E M'$ we say that M *reduces to M'* . Equations of the simple form $L = R$ satisfying the restriction that the variables of R are contained in the set of variables of L are *executable*, in the sense that, given a term M , we can decide if the equation can be applied, and if this is the case we can compute the set of all results obtained by applying the equation to M (possibly at different subterms).

The idea behind an operational reduction semantics is that each application of an equation transforms a term into another term that is simpler in a certain sense. Therefore, for an equational specification to be computationally useful we usually expect that if a term cannot be further reduced we have computed its unique *normal form* which represents the value of the given term. A sufficient condition for the uniqueness of normal forms is that the reduction relation \rightarrow_E is *confluent*, i.e., if a term M can be reduced to different terms M' and M'' , then both of them can be further reduced to the same term M''' . Another property that is often of interest is *termination* of \rightarrow_E , i.e. the property that infinite computations do not exist. Clearly, confluence and termination together ensure that each term has a unique normal form. In this thesis, we will mostly work with confluent equational specifications, although we will argue in Chapter 8 that confluence is already a quite strong property that could be replaced by weaker notions. We do not require any termination properties, however, since we are also interested in partially terminating specifications, where normal forms exist only for a subset of all terms. A typical example would be an equational specification of the untyped λ -calculus (cf. Chapter 5).

The next example is an executable and terminating many-sorted equational specification of lists over the sort of identifiers already introduced. The specification extends the previous specification of `Id` and a specification of `Bool` (see below). We should add a constraint stating that the theory of `List-Id` with operators `nil` and `cons` is free over the trivial parameter theory of `Id`, to express that we are concerned with an instantiation of a parameterized theory.

```

sort List-Id .

var l' : List-Id .

op nil : -> List-Id .
op cons : Id List-Id -> List-Id .

```

```

op in : Id List-Id -> Bool .
eq in(x,nil) = false .
eq in(x,cons(x',l')) =
  if x == x' then true else in(x,l') fi .

```

Above we presuppose the following specification of a sort `Bool`, and we should add an initiality constraint for the subtheory of `Bool` with constants `false` and `true`.

```

sort Bool .

op false : -> Bool .
op true  : -> Bool .

var A,B : Bool .

op if_then_else_fi : Bool Bool Bool -> Bool .

eq if true then A else B = A .
eq if false then A else B = B .

```

Additionally, we presuppose a specification of a boolean equality function `_==_` for `Id`, together with equations stating that the constants of sort `Id` are distinct, making explicit a logical consequence of the initiality constraint in the specification of `Id`.

```

op _==_ : Id Id -> Bool .

eq a == a = true .
eq a == b = false .
eq a == c = false .
...

```

Maude and OBJ3 provide a similar equality operator `_==_` for all sorts of the specification in a built-in fashion, based on the assumption that the specification is confluent. Given a term $M == N$, the engine fully reduces M and N to say M' and N' and checks if M' and N' are equal. The term $M == N$ evaluates to `true` if this is the case, and to `false` otherwise. In this thesis we prefer to use an explicit definition such as the one given above for two reasons: First of all, we do not want to base soundness of the operational semantics on a confluence assumption, since confluence is difficult to verify in our general setting of

membership equational logic. The second reason is that even for confluent and terminating specifications the operational semantics of `==` given above only coincides with the model-theoretic semantics for initial models, but not in the general case. For instance, the fact that $M == N$ evaluates to `false` using the above built-in mechanism means that the interpretations of M and N are distinct in the initial semantics, but in general there may be models where the interpretations of M and N are equal.

It has been shown in [BT80] that every total computable function can be represented by a confluent and terminating term rewriting system. In spite of its operational universality many-sorted algebra requires ad hoc encoding of important concepts, such as conditional axioms, predicates and partial functions, which makes it insufficient in practice. Similarly, the logical expressiveness of a pure equational logic is confined to equationally definable classes of models, which is not sufficient for most applications. These limitations have prompted several extensions, such as structural equations, conditional equations, subsorts, and memberships; concepts that will be informally introduced in the following sections. It is important to keep in mind that all these extensions preserve the two main characteristics of an equational logic, namely, that it is weak enough to admit initial and free models and that it can be naturally equipped with an operational semantics.

2.4.2 Structural vs. Computational Equations

The motivation for introducing structural equations comes from the fact that there are cases where equations cannot be naturally regarded as reduction rules, because such rules would be nonterminating, or where it is operationally not desirable to favor a particular direction of computation, because each possible choice would artificially break a natural symmetry. From a model-theoretic point of view *structural equations* are not distinguished from other equations. Operationally, however, structural equations are not regarded as reduction rules but they are already satisfied by the term representation. To distinguish structural equations from equations intended as reductions rules, we refer to the latter as *computational equations*.

Typical examples where structural equations are useful are algebraic representations of finite data types such as sets and multisets, where equality is coarser than syntactic equality and there is no canonical representation. An example is the following specification of finite sets. Extending the previous specification of `Id`, we declare a new sort `FS-Id` intended to represent finite sets of identifiers. In contrast to the specification of lists, we use a commutative constructor `union` that directly reflects the unordered nature of sets. Since we are concerned with an instance of a parameterized specification, we should add a constraint stating that the subtheory of `FS-Id` with `empty`, `single`, `union` and the corresponding

equations is free over the trivial parameter theory `Id`.

```

sort FS-Id .

var x x' : Id .
var s s' s'' : FS-Id .

op empty : -> FS-Id .
op single : Id -> FS-Id .
op union : FS-Id FS-Id -> FS-Id [assoc comm] .

eq union(s,s) = s .
eq union(empty,s) = s .

op in : Id FS-Id -> Bool .
eq in(x,empty) = false .
eq in(x,union(single(x'),s')) =
  if x == x' then true else in(x,s') fi .

```

The operator attributes in square brackets specify that `union` is an operator with associativity and commutativity as structural equations. The two computational equations introduced by `eq` express idempotence of `union` and the fact that `empty` is its identity element, respectively. The attributes `assoc` and `comm` of `union` are tantamount to imposing the following structural equations:²

```

seq union(union(s,s'),s'') = union(s,union(s',s'')) .
seq union(s,s') = union(s',s) .

```

Although these equations are equivalent to ordinary equations from the model-theoretic point of view, we would have nonterminating computations in the operational semantics, if these equations were used as reduction rules. Instead we identify terms which are equal by virtue of the structural equations and define the reduction relation on top of these. This technique is known as *reduction modulo a congruence* [BD89, BN98], where in our case the congruence is specified by the structural equations. To reflect the difference in their operational semantics, the set E of axioms of the specification is partitioned into E^S , containing the structural equations introduced by `seq` (or by operator attributes as a shorthand), and E^C , containing the computational equations introduced by `eq`.

For suitable sets of structural axioms the operational semantics can be implemented by using an internal term representation which satisfies the structural

²In the syntax of OBJ3 and Maude, structural equations are always introduced by the operator attributes specified in square brackets. In this thesis we sometimes prefer to state them explicitly using the keyword `seq`.

equations and a matching algorithm which operates modulo these structural equations. Since such an algorithm exists for the combination of structural equations in our example, namely associativity and commutativity, the computational equations of our specification are indeed executable.

Another typical combination of structural equations is used in the following specification of finite multisets, which play a central role in the representation of concurrent systems (cf. Chapter 3). Using the fact that finite multisets are free commutative monoids we obtain the following specification of a *finite multiset sort* `FMS-Id` over `Id`. The specification extends the specification of `Id` and we should add a constraint stating that the subtheory of `FMS-Id` with operators `empty`, `single`, `union` and the corresponding equations is free over the trivial parameter theory `Id`.

```

sort FMS-Id .

op empty : -> FMS-Id .
op single : Id -> FMS-Id .
op union : FMS-Id FMS-Id -> FMS-Id [assoc comm id: empty] .

op in : Id FMS-Id -> Bool .
eq in(x,empty) = false .
eq in(x,union(single(x'),s')) =
  if x == x' then true else in(x,s') fi .

```

The operator attributes in square brackets specify that `union` is an associative, commutative operator with `empty` as a left and right identity element, i.e., the operator attributes `assoc`, `comm`, and `id: empty` are equivalent to the following structural equations:

```

seq union(union(s,s'),s'') = union(s,union(s',s''))
seq union(s,s') = union(s',s)
seq union(empty,s) = s
seq union(s,empty) = s

```

Again, a matching algorithm for this combination of equations exists such that the specification is indeed executable. In fact, the previous specifications of `FS-Id` and `FMS-Id` can both be executed in Maude which supports all combinations of associativity, commutativity, and left/right identity laws as structural equations.

To conclude this brief introduction to the use of structural equations we would like to point out that there is some flexibility in the choice whether an equation should be considered to be a structural equation or a computational equation. For instance, the identity law is specified by a computational equation in the

set specification, but to demonstrate this flexibility it is specified by a structural equation in the multiset specification. The advantage of structural equations is that the matching capabilities can be exploited, often leading to a concise specification style. On the other hand, structural equations should be used with care, since, especially in the presence of identity laws, they can lead to unexpected nonterminating computations.

2.4.3 Order-Sorted Algebra

Order-sorted equational logic, also called *order-sorted algebra*, is a generalization of many-sorted equational logic. There are a number of variations on the theme of order-sorted equational logic (see [GD94] for a survey, and [Mes98] for further discussion and comparison of different alternatives). Here we briefly describe the most common approach [GM92]. Order-sorted equational specifications in this sense are supported by the language OBJ3. A similar form of order-sorted equational logic appears as a sublanguage of membership equational logic that is supported by Maude.

Order-sorted equational specifications allow us to specify *subsort axioms* which are set-theoretically interpreted as inclusions. Subsort axioms induce a preorder, i.e. a reflexive and transitive relation, on the set of sorts. Below we have a simple theory that, when equipped with an initiality constraint, is a specification of the natural numbers with successor and predecessor functions:

```

sort Nat NzNat .

subsort NzNat < Nat .

op 0 : -> Nat .
op suc : Nat -> NzNat .
op pred : NzNat -> Nat .

var n : Nat .
eq pred(suc(n)) = n .

```

As another typical example of an order-sorted specification we extend the previous specification to a specification of natural numbers and integers as follows, assuming that we add a suitable initiality constraint.

```

sort Int .

subsort Nat < Int .

```

```

op suc : Int -> Int .
op pred : Int -> Int .

var i : Int .
eq pred(suc(i)) = i .
eq suc(pred(i)) = i .

```

Order-sorted specifications allow *subsort overloading*. Operators such as `pred` in the previous example are *interpreted as a single set-theoretic function*, but there are different views on this function, as expressed by the operator declarations:

```

op pred : NzNat -> Nat .
op pred : Int -> Int .

```

Another form of overloading supported by order-sorted specifications is *ad hoc overloading*, where operators with the *same name* can have independent specifications and different interpretations. Polymorphic equality is an example of ad hoc overloading, if equality is specified for each sort independently.³

```

op _==_ : Bool Bool -> Bool .
op _==_ : Id Id -> Bool .
op _==_ : Int Int -> Bool .

```

Ad hoc overloading can also be used to specify the standard polymorphic operator `if_then_else-fi`:⁴

```

op if_then_else-fi : Bool Bool Bool -> Bool .
op if_then_else-fi : Bool Id Id -> Id .
op if_then_else-fi : Bool Int Int -> Int .

```

The examples above raise the question of how to distinguish ad hoc overloading from subsort overloading in view of the fact that their semantics is quite different. The solution of order-sorted algebra is to take the subsort order \leq (written $<$ in Maude syntax) induced by the subsort declarations into consideration. The subsort order is naturally extended to tuples of sorts. Now the following condition is imposed on the algebra: Given two operators $f : \bar{s} \rightarrow s$ and $f' : \bar{s}' \rightarrow s'$, if the arities of two operators are comparable, i.e., $\bar{s} \leq \bar{s}'$ or $\bar{s}' \leq \bar{s}$, we are concerned

³In some languages it is possible to define a single polymorphic equality uniformly over all types. In this case we would speak of parametric polymorphism rather than of ad hoc polymorphism (cf. [Bur00]).

⁴In this case, the use of parametric polymorphism would be more appropriate. Indeed, in Chapter 8 we show how it can be specified in the open calculus of constructions as a single polymorphic function in a uniform way.

with *subsort overloading* and the functions interpreting these operators should coincide on their common domain. Otherwise the arities of two operator symbols are incomparable w.r.t. the subsort order, and we are concerned with *ad hoc overloading* with no restrictions imposed on the interpretation of these operators.

It is noteworthy that order-sorted theories have initial and, more generally, free models, but only if a number of technical conditions are imposed upon the specification [GM92]. This difficulty disappears in other formulations of order-sorted algebra [GD94] and in the richer framework of membership equational logic [Mes98] that is introduced next. In connection with this issue, the notions of ad hoc overloading and subsort overloading are simplified, leading to subtle differences in the conditions given above. To eliminate these differences we impose the following mild requirement on order-sorted equational theories throughout this thesis: For every two operators $f : \bar{s} \rightarrow s$ and $f' : \bar{s}' \rightarrow s'$, either \bar{s} and \bar{s}' are comparable (giving rise to subsort-overloading), or \bar{s} and \bar{s}' are clearly unrelated, i.e. not in the same connected component of the subsort order (giving rise to ad hoc overloading).

We conclude this section about order-sorted algebra with another example of the use of subsorts that is typical of the OBJ3 and Maude specification style. Reconsider the specification of FS-Id. Instead of an explicit injection operator such as `single`, it is often possible to interpret injection as set-theoretic inclusion without changing the category of models essentially. In the specification of FS-Id this fact could be exploited by replacing the declarations

```
op empty : -> FS-Id .
op single : Id -> FS-Id .
op union : FS-Id FS-Id -> FS-Id [assoc comm] .
```

by

```
op empty : -> FS-Id .
subsort Id < FS-Id .
op __ : FS-Id FS-Id -> FS-Id [assoc comm] .
```

Here we have also renamed `union` to `__` using the OBJ3 style mixfix syntax. The advantage of identifying `x` and `single(x)`, is the more convenient notation which nicely supplements the mixfix syntax. For instance, we can now write `(a b c)` instead of `union(union(single(a),single(b)),single(c))`, which is clearly more readable. A disadvantage, however, is that the possibilities catching sort errors by static type checking are diminished. Since we aim at a strong typing discipline (especially in Chapter 8), we will often avoid this identification in favor of a more explicit notation.

Equational specifications can be generalized to specifications with conditional equations where conditions are conjunctions of equations. In fact, the order-

sorted equational specifications of OBJ3 admit conditional equations of this kind. However, we defer the discussion of conditional axioms to the next section, where we introduce *membership equational logic* [BJM97], a richer specification language that naturally subsumes many-sorted and order-sorted algebra.

2.4.4 Membership Equational Logic

Membership equational logic (MEL) [BJM97, BJM00, Mes98] can be regarded as a many-sorted first-order Horn clause logic with equality and unary predicates. In accordance with the terminology introduced in the given references we call the types of the logic not sorts (as in the previous subsection) but *kinds*, and we refer to the unary predicates as *sorts*. The atomic sentences are *equalities* $M = N$ for terms M and N of the same kind, and *memberships* $M : s$ for a term M and a sort s , both of the same kind. Sentences of MEL are universally quantified Horn clauses on the atoms. In general, the kind of a term can be syntactically determined, whereas the sort of a term is not necessarily unique and may depend on equational and membership axioms. Semantically, a kind is interpreted as a set, and each sort of this kind corresponds to a subset.

A typical specification in membership equational logic is the following specification of paths [CDE⁺99a] in a concrete graph together with operations for concatenation and for determining their length. As in [CDE⁺99a] we assume an initiality constraint for the entire theory.

```

sorts Edge Path Path? Node .
subsorts Edge < Path < Path? .

ops n1 n2 ... : -> Node .
ops e1 e2 ... : -> Edge .

ops source target : Path -> Node .

eq source(e1) = n1 . eq target(e1) = n2 .
eq source(e2) = n1 . eq target(e2) = n3 .
...

var E : Edge . var P : Path . var Q : Path? .

op concat : Path? Path? -> Path? [assoc] .
cmb concat(E, P) : Path
  if target(E) = source(P) .

ceq source(concat(E, P)) = source(E) if concat(E, P) : Path .

```

```
ceq target(concat(P, E)) = target(E) if concat(P, E) : Path .
```

MEL is at least as expressive as order-sorted equational logic, since a subsort axiom such as

```
subsort Edge < Path .
```

is semantically equivalent to a conditional membership axiom of the form

```
cmb Q : Path if Q : Edge .
```

Furthermore, in MEL we can define functions using conditional equations such as:

```
op length : Path -> Nat .
eq length(E) = 1 .
ceq length(concat(E, P)) = suc(length(P))
   if target(E) = source(P) .
```

Operationally, equational axioms (with optional conditions) in membership equational logic are again interpreted either as *computational equations*, i.e. as reduction rules (those introduced by `eq` and `ceq`), or as *structural equations* (like associativity for `concat` in our example) and the technique of reduction modulo structural equations is used to execute membership equational specifications. Since computational equations can be conditional, determining their applicability can involve not only matching modulo the structural equations, but can also require the verification of its conditions. Equational conditions $M = N$ are verified by reducing M and N until both sides are equal (modulo the structural equations). Membership conditions $M : \mathbf{s}$ are verified by a combination of reduction using computational equations and exhaustive goal-oriented proof search using membership axioms, that we also refer to as *membership assertions*.

Relationship to Horn Clause Logic with Equality

MEL is a special *many-sorted Horn clause logic with equality* (*MSHCLEQ*) [GM87]. More precisely, each sort of MEL can be regarded as a unary predicate, giving rise to a map of logics $\mathbf{MEL} \rightarrow \mathbf{MSHCLEQ}$ in the sense of [Mes89a], which is essentially a sublogic inclusion. Conversely, MSHCLEQ can be mapped into MEL as follows: We assume a sort `Prop` of propositions, i.e. formulae, together with a subsort `True` of true propositions. For each predicate $P : \bar{s}_1, \dots, \bar{s}_n$ in MSHCLEQ we introduce an operator $P : \bar{s}_1, \dots, \bar{s}_n \rightarrow \mathbf{Prop}$. Then each atomic formula $P(M_1, \dots, M_n)$ is translated into a membership $P(M_1, \dots, M_n) : \mathbf{True}$.

It is easy to show that again this gives rise to a map of logics **MSHCLEQ** \rightarrow **MEL**.⁵ In summary, we see that MEL and MSHCLEQ are just slight syntactic variations of each other, MEL being slightly simpler, since it has only unary predicates (sets) rather than n -ary predicates (relations) as a primitive concept. As a simple example of the use of such a propositional sort we can see below a positive specification of equality of identifiers and membership in a finite set of identifiers.

```

sort Prop True .
subsort True < Prop .

op  equal : Id Id -> Prop .
mb  equal(x,x) : True .

op  in : Id FS-Id -> Prop .
cmb in(x,union(single(x'),s')) : True if equal(x,x') : True .
cmb in(x,union(single(x'),s')) : True if in(x,s') : True .

```

The advantage of this representation of predicates is that it does not make any assumptions about the nature of **Prop** and it does not enforce any equations for predicates. Hence, it is not necessary to interpret **Prop** as the set of booleans, but a more intensional interpretation of **Prop** is possible as well. This approach is quite different from the typical OBJ3 specification style, which is classically biased, since predicates are represented as boolean-valued functions. Membership equational logic removes this imbalance and supports equations and memberships on an equal footing.

As suggested in [Gog] it is possible to establish a map from MSHCLEQ into the weaker *many-sorted conditional equational logic (MSCEQL)*, which has a notion of equality but no further predicates. This is done as above, but we interpret each atomic formula $P(M_1, \dots, M_n)$ as an equation $P(M_1, \dots, M_n) = \mathbf{true}$ instead of $P(M_1, \dots, M_n) : \mathbf{True}$. Composing this with the map **MEL** \rightarrow **MSHCLEQ** mentioned above we obtain a map **MEL** \rightarrow **MSCEQL**. A disadvantage, in our view, is that this map is still classically biased, because it introduces unnecessary identifications between elements of **Prop**.

In the remainder of this section we define membership equational theories and membership equational specifications together with their model-theoretic and operational semantics.

⁵Different from an alternative map of this kind studied in [Mes98], predicates receive a first-class status in our representation, which is more suitable for a generalization to higher-order logics and logical type theories (cf. Chapter 8).

Syntax

A *membership equational signature* Ω consists of a set of *kinds* Knd_Ω , a set of sorts Srt_Ω , a function $knd_\Omega : Srt_\Omega \rightarrow Knd_\Omega$ that associates to each sort its kind, and a family $(Op_\Omega^{\bar{k},k})_{\bar{k} \in Knd_\Omega^*, k \in Knd_\Omega}$ of *operator symbols* such that the following *overloading restriction*⁶ holds: If $Op_\Omega^{\bar{k},k} \cap Op_\Omega^{\bar{k}',k'} \neq \emptyset$ then $\bar{k} = \bar{k}'$ implies $k = k'$. Instead of $o \in Op_\Omega^{\bar{k},k}$ we simply write $o : \bar{k} \rightarrow k$. If \bar{k} is empty we write $o : \rightarrow k$, and o is called a *constant symbol*, otherwise o is called a *function symbol*.

Given membership equational signatures Ω and Ω' , a *membership equational signature morphism* $H : \Omega \rightarrow \Omega'$ consists of functions $H_{Knd} : Knd_\Omega \rightarrow Knd_{\Omega'}$, $H_{Srt} : Srt_\Omega \rightarrow Srt_{\Omega'}$ and $H_{Op} : Op_\Omega \rightarrow Op_{\Omega'}$ such that (1) $H_{Knd}(knd_\Omega(s)) = knd_{\Omega'}(H_{Srt}(s))$ for each sort $s \in Srt_\Omega$, and (2) $f : \bar{k} \rightarrow k$ in Ω implies $H_{Op}(f) : H_{Knd}(\bar{k}) \rightarrow H_{Knd}(k)$ in Ω' . We usually omit the indices of H if there is no danger of confusion. Membership equational signatures together with their morphisms form a category **MESign**.

A *kinded variable set* is a family $(X_k)_{k \in Knd}$ of pairwise disjoint sets which are also disjoint from the operator symbols in Op_Ω . For convenience, we generally assume that variables are chosen from a fixed totally ordered set, so that each finite kinded variable set X has a *canonical enumeration*, i.e. a list $\bar{x} = \bar{x}_1, \dots, \bar{x}_n$ such that $X = \{\bar{x}_1, \dots, \bar{x}_n\}$ where all x_i are distinct. Indeed, in this case we often identify X with $\bar{x} : \bar{k} = x_1 : k_1, \dots, x_n : k_n$ where $x_i \in X_{k_i}$, and we also refer to $\bar{x} : \bar{k}$ as its *canonical enumeration*.

Given a kinded variable set X , the kinded set of Ω -terms over X , written $Trm_\Omega(X) = (Trm_\Omega(X)_k)_{k \in Knd}$, is inductively defined as follows: (1) each variable $x \in X_k$ is in $Trm_\Omega(X)_k$; (2) each constant symbol c with $c : \rightarrow k$ is in $Trm_\Omega(X)_k$ for $k \in Knd$; (3) each *function application* of the form $f(M_1, \dots, M_n)$ is in $Trm_\Omega(X)_k$ for $f : \bar{k} \rightarrow k$ with $\bar{k} = k_1, \dots, k_n$ and $M_i \in Trm_\Omega(X)_{k_i}$. If X is the empty variable set, then the terms over X are just called *terms* (or *ground terms* to emphasize that they do not contain variables), and we write Trm_Ω and $Trm_{\Omega,k}$ instead of $Trm_\Omega(X)$ and $Trm_\Omega(X)_k$, respectively. In the context of a membership equational signature Ω we typically use variables M and N to range over terms in Trm_Ω .

A *kinded substitution* for X is a kinded function $\theta = (\theta_k)_{k \in Knd}$ associating to each $x \in X_k$ an element $\theta_k(x) \in Trm_{\Omega,k}$. It is extended to terms over X as follows: (1) $\theta_k(c) = c$ for $c : \rightarrow k$; and (2) $\theta_k(f(M_1, \dots, M_n)) = f(\theta_{k_1}(M_1), \dots, \theta_{k_n}(M_n))$ for $f : \bar{k} \rightarrow k$ and $M_i \in Trm_\Omega(X)_{k_i}$ where $\bar{k} = k_1, \dots, k_n$. Instead of $\theta_k(M)$ for $M \in Trm_\Omega(X)_k$ we also use the notation $\theta(M)$.

We define *atomic Ω -formulae over X* as either: (1) Ω -memberships over X of the form $M : s$ for $M \in Trm_\Omega(X)_{knd(s)}$, or (2) Ω -equations over X of the form $M =$

⁶This overloading restriction has been further relaxed in the Maude language.

N for $M, N \in \text{Trm}_\Omega(X)_k$ for some kind k . Furthermore, Ω -conditions over X are conjunctions of the form $\phi_1 \wedge \dots \wedge \phi_n$, where ϕ_1, \dots, ϕ_n are atomic formulae over X . Given an Ω -condition $\phi_1 \wedge \dots \wedge \phi_n$ over X , an Ω -axiom can be either: (1) a *membership axiom* of the form $\forall X . M : s$ **if** $\phi_1 \wedge \dots \wedge \phi_n$, where $M : s$ is an Ω -membership over X , or (2) an *equational axiom* of the form $\forall X . M = N$ **if** $\phi_1 \wedge \dots \wedge \phi_n$, where $M = N$ is an Ω -equation over X . We usually omit the quantifier if X is empty. A membership axiom of the form $\forall x : k . x : s'$ **if** $x : s$ is also called a *subsort axiom* and expresses that s is a subsort of s' .

A *membership equational theory (MET)* \mathcal{E} consists of a signature $\Omega_{\mathcal{E}}$ and a set of $\Omega_{\mathcal{E}}$ -axioms $E_{\mathcal{E}}$.

Model-theoretic Semantics

The model-theoretic semantics of membership equational logic is a standard algebraic one [BJM97, BJM00, Mes98]. *Models* of a membership equational theory are suitable algebras satisfying the axioms. The details are given below.

Let Ω be a signature. An Ω -algebra A consists of a *kind interpretation* $\llbracket k \rrbracket_A$ for each $k \in \text{Knd}$, a *sort interpretation* $\llbracket s \rrbracket_A \subseteq \llbracket k \rrbracket_A$ for each $s \in \text{kind}^{-1}(k)$, an *operator interpretation* $\llbracket o_{\bar{k},k} \rrbracket_A$ for each $o : \bar{k} \rightarrow k$ such that $\llbracket c_k \rrbracket_A \in \llbracket k \rrbracket_A$ for $c : \rightarrow k$ and $\llbracket f_{\bar{k},k} \rrbracket_A \in \llbracket k \rrbracket_A \rightarrow \llbracket k \rrbracket_A$ for $f : \bar{k} \rightarrow k$ where $\llbracket \bar{k} \rrbracket_A = \llbracket k_1 \rrbracket_A \times \dots \times \llbracket k_n \rrbracket_A$ if $\bar{k} = k_1, \dots, k_n$. For better readability we often write $\llbracket c \rrbracket_A$ and $\llbracket f \rrbracket_A$ instead of $\llbracket c_k \rrbracket_A$ and $\llbracket f_{\bar{k},k} \rrbracket_A$, assuming that the subscripts are clear from the context. To simplify some constructions we assume in this thesis, without loss of generality, that $\llbracket k \rrbracket_A \cap \llbracket k' \rrbracket_A = \emptyset$ for all kinds $k \neq k'$.

Let A, B be Ω -algebras. An Ω -morphism, written $h : A \rightarrow B$, is a kinded function $h = (h_k)_{k \in \text{Knd}}$ such that $h_k : \llbracket k \rrbracket_A \rightarrow \llbracket k \rrbracket_B$ for all $k \in \text{Knd}$ and the following conditions hold: (1) $h_k(\llbracket s \rrbracket_A) \subseteq \llbracket s \rrbracket_B$ for $s \in \text{kind}^{-1}(k)$; (2) $h_k(\llbracket c_k \rrbracket_A) = \llbracket c_k \rrbracket_B$ for $c : \rightarrow k$; and (3) $h_k(\llbracket f_{\bar{k},k} \rrbracket_A(a_1, \dots, a_n)) = \llbracket f_{\bar{k},k} \rrbracket_B(h_{k_1}(a_1), \dots, h_{k_n}(a_n))$ for $f : \bar{k} \rightarrow k$ with $\bar{k} = k_1, \dots, k_n$ and $a_i \in \llbracket k_i \rrbracket_A$. Ω -algebras together with Ω -morphisms constitute a category $\mathbf{Mod}(\Omega)$.

Let A be an Ω -algebra. A *kinded assignment* $\beta : X \rightarrow A$ is a kinded function $\beta = (\beta_k)_{k \in \text{Knd}}$ associating to each $x \in X_k$ an element $\beta_k(x) \in \llbracket k \rrbracket_A$. It is extended to terms over X as follows: (1) $\beta_k(c) = \llbracket c_k \rrbracket_A$ for $c : \rightarrow k$; and (2) $\beta_k(f(M_1, \dots, M_n)) = \llbracket f_{\bar{k},k} \rrbracket_A(\beta_{k_1}(M_1), \dots, \beta_{k_n}(M_n))$ for $f : \bar{k} \rightarrow k$ and $M_i \in \text{Trm}_\Omega(X)_{k_i}$ where $\bar{k} = k_1, \dots, k_n$. Instead of $\beta_k(M)$ for $M \in \text{Trm}_\Omega(X)_k$ we also use the notation $\beta(M)$ or $\llbracket M \rrbracket_{A,\beta}$.

Let A be an Ω -algebra, let $\beta : X \rightarrow A$ be a kinded assignment, and let $M, N \in \text{Trm}_\Omega(X)_k$. We define validity of formulae starting with atomic formulae: an Ω -membership $M : s$ over X is *valid* under β iff $\llbracket M \rrbracket_{A,\beta} \in \llbracket s \rrbracket_A$; and an Ω -equation $M = N$ over X is *valid* under β iff $\llbracket M \rrbracket_{A,\beta} = \llbracket N \rrbracket_{A,\beta}$. We write $A, \beta \models \phi$ iff the atomic formula ϕ is valid under β . Furthermore, an Ω -condition $\phi_1 \wedge \dots \wedge \phi_n$

over X is *valid* under β iff $A, \beta \models \phi_i$ for each $i \in \{1, \dots, n\}$, in which case we also write $A, \beta \models \phi_1 \wedge \dots \wedge \phi_n$. An Ω -axiom $\forall X . \phi$ **if** $\phi_1 \wedge \dots \wedge \phi_n$ is *valid* iff for each kinded assignment $\beta : X \rightarrow A$ we have $A, \beta \models \phi$ whenever $A, \beta \models \phi_1 \wedge \dots \wedge \phi_n$. We also write $A \models \forall X . \phi$ **if** $\phi_1 \wedge \dots \wedge \phi_n$ in this case. Given a set E of Ω -axioms, we write $A \models E$ iff $A \models \psi$ for each $\psi \in E$. Given a MET \mathcal{E} , we say that A is an \mathcal{E} -algebra iff $A \models E_{\mathcal{E}}$. We say that ψ is *valid* in \mathcal{E} (written $\mathcal{E} \models \psi$) iff $A \models \psi$ for each \mathcal{E} -algebra A , and given a set E of Ω -axioms, we say that E is *valid* in \mathcal{E} (written $\mathcal{E} \models E$) iff $\mathcal{E} \models \psi$ for each $\psi \in E$.

Given METs \mathcal{E} and \mathcal{E}' , a *MET morphism* $H : \mathcal{E} \rightarrow \mathcal{E}'$ is a membership equational signature morphism $H : \Omega_{\mathcal{E}} \rightarrow \Omega_{\mathcal{E}'}$ such that $\mathcal{E}' \models H(E_{\mathcal{E}})$, where H is lifted to terms and axioms in the natural homomorphic way. We say that \mathcal{E} is a *subtheory* of \mathcal{E}' , written $\mathcal{E} \hookrightarrow \mathcal{E}'$, iff there is a MET morphism $J : \mathcal{E} \hookrightarrow \mathcal{E}'$ that is an inclusion.

METs together with their morphisms form a category **MET**, and given a MET \mathcal{E} the class of \mathcal{E} -algebras together with their Ω -morphisms constitutes a full subcategory of **Mod**(Ω) denoted by **Mod**(\mathcal{E}). **Mod**(\mathcal{E}) is called the *model-theoretic semantics* of \mathcal{E} . Each MET morphism $H : \mathcal{E} \rightarrow \mathcal{E}'$ induces an obvious forgetful functor **Mod**(H) : **Mod**(\mathcal{E}') \rightarrow **Mod**(\mathcal{E}) that we also write as \mathbf{U}_H . In fact, we have a contravariant functor **Mod** : **MET** \rightarrow **Cat**^{op}. Given an inclusion $I : \mathcal{E} \hookrightarrow \mathcal{E}'$ and an \mathcal{E}' -algebra A , we also write $A|_{\mathcal{E}}$ instead of $\mathbf{U}_I(A)$.

Theorem 2.4.1 (Initial and Free Models) METs have initial and free models [BJM97, BJM00, Mes98]. In fact, given a MET \mathcal{E}' there exists an initial \mathcal{E}' -algebra, written $\mathbf{I}(\mathcal{E}')$. More generally, given a MET morphism $H : \mathcal{E} \rightarrow \mathcal{E}'$ between METs \mathcal{E} and \mathcal{E}' there exists a free functor $\mathbf{F}_H : \mathbf{Mod}(\mathcal{E}) \rightarrow \mathbf{Mod}(\mathcal{E}')$, i.e. a functor that is left adjoint to \mathbf{U}_H . In the following we write η_H and ϵ_H for the unit and counit, respectively, of this adjunction, i.e., we have natural transformations $\eta_H(A) : A \rightarrow \mathbf{U}_H(\mathbf{F}_H(A))$ for \mathcal{E} -algebras A and $\epsilon_H(A') : \mathbf{F}_H(\mathbf{U}_H(A')) \rightarrow A'$ for \mathcal{E}' -algebras A' .

Deductive System

The logical and computational aspects of MEL are best defined by two related formal systems, which we call the deductive system and the computational system of MEL. The first system is concerned with the view of MEL as a logic, in which valid sentences can be proved by deduction. The computational system is concerned with the view of MEL as an executable specification language. Here the details are important for us, because it is the basis of the operational semantics of MEL, which we specifically use in several parts of this thesis, and furthermore because it is the starting point for our generalization of MEL to the typed higher-order case, which is the subject of Chapter 8.

We now present the *deductive system of MEL*. Our presentation is similar to

[BJM97, BJM00], but we do not consider variable declarations, because formally they can be seen as additional constants in the signature. In the following we use Ω to range over MEL signatures and E to range over sets of Ω -axioms so that Ω and E define a MET \mathcal{E} . As sentences the deductive system uses the following judgements. (*Deductive*) MEL judgements are *membership judgements* $\Omega, E \vdash M : s$ and *equality judgements* $\Omega, E \vdash M = N$. The rules of the deductive system given below, where we use θ to range over kinded substitutions for X .

$$\frac{}{\Omega, E \vdash M = M} \quad (\text{EqRefl})$$

$$\frac{\Omega, E \vdash M = M'}{\Omega, E \vdash M' = M} \quad (\text{EqSym})$$

$$\frac{\Omega, E \vdash M = M' \quad \Omega, E \vdash M' = M''}{\Omega, E \vdash M = M''} \quad (\text{EqTrans})$$

$$\frac{\Omega, E \vdash M_i = M'_i \text{ for } i \in \{1, \dots, n\}}{\Omega, E \vdash f(M_1, \dots, M_n) = f(M'_1, \dots, M'_n)} \quad (\text{EqApp})$$

$$\frac{\forall X. \phi \text{ if } \phi_1 \wedge \dots \wedge \phi_m \in E \quad \Omega, E \vdash \theta(\phi_i) \text{ for } i \in \{1, \dots, m\}}{\Omega, E \vdash \theta(\phi)} \quad (\text{Axiom})$$

$$\frac{\Omega, E \vdash M' : s \quad \Omega, E \vdash M = M'}{\Omega, E \vdash M : s} \quad (\text{MbEq})$$

As explained before, MEL is a special case of MSHCLEQ, that is many-sorted Horn clause logic with equality. Indeed, the deductive system above is a special case of the sound and complete system for many-sorted Horn clause logic presented in [GM87], so that also the deductive system of MEL is sound and complete w.r.t. its model-theoretic semantics. The proof of completeness relies on a standard initial model construction [BJM97, BJM00].

Computational System

The *operational semantics of MEL* is a partial specification of an algorithm to execute METs. The algorithm is specified by means of a formal system, that we call the *computational system of MEL*, together with certain requirements regarding the application or implementation of the inference rules. The computational system given subsequently is equivalent to the rules of deduction for sort-decreasing conditional rewrite/membership systems presented in [BJM97, BJM00] except for the fact that we have explicit judgements for structural equality in the sys-

tem below. In particular, it is a refinement of the *deductive system of MEL*, so that the operational semantics is sound by construction.

Whenever we are interested in the operational semantics of MEL, which is the case especially in the following, we consider METs \mathcal{E} with a signature Ω and axioms $E = E^S \cup E^C \cup E^A$ where *structural axioms* E^S , *computational axioms* E^C , and *assertional axioms* E^A can be distinguished and have the following properties: Structural axioms in E^S are unconditional equations, computational axioms in E^C are conditional equations, and assertional axioms in E^A are conditional membership axioms.

As sentences the computational system of MEL uses a number of different judgements that we introduce first. (*Computational*) *MEL judgements* are either *assertional membership judgements* $\Omega, E \vdash M : s$, *assertional equality judgements* $\Omega, E \vdash M = N$, *structural equality judgements* $\Omega, E \vdash M \equiv_S N$, or *computational equality judgements* $\Omega, E \vdash M \xrightarrow{*}_E N$.

Subsequently, we present the rules of the computational system together with the requirements for the operational semantics. To emphasize the operational goal-oriented reading we have grouped these rules according to the form of their conclusion. Again we use θ to range over kinded substitutions for X .

Structural equality judgements are closed under the following rules:

$$\frac{}{\Omega, E \vdash M \equiv_S M} \quad (\text{StrRefl})$$

$$\frac{\Omega, E \vdash M \equiv_S N}{\Omega, E \vdash N \equiv_S M} \quad (\text{StrSym})$$

$$\frac{\Omega, E \vdash M \equiv_S M' \quad \Omega, E \vdash M' \equiv_S M''}{\Omega, E \vdash M \equiv_S M''} \quad (\text{StrTrans})$$

$$\frac{\Omega, E \vdash M_i \equiv_S M'_i \text{ for } i \in \{1, \dots, n\}}{\Omega, E \vdash f(M_1, \dots, M_n) \equiv_S f(M'_1, \dots, M'_n)} \quad (\text{StrApp})$$

$$\frac{\forall X . M = M' \in E^S}{\Omega, E \vdash \theta(M) \equiv_S \theta(M')} \quad (\text{Str})$$

Operationally, the structural equality judgements are realized by a suitable term representation in which M and N are indistinguishable in all computational and assertional judgements iff $\Omega, E \vdash M \equiv_S N$. The rule **Str** expresses that structural equality judgements are generated by the structural equations in E^S . Structural equality can be seen as an instantaneous reversible transformation, which justifies rules for reflexivity, symmetry, and transitivity, and furthermore, since this transformation can take place in arbitrary subterms, we have the context closure

rule **StrApp**.

To simplify the presentation of the computational system we now introduce the general convention that in the context of a given MET with signature Ω and axioms E *structurally equal* terms, i.e. those terms for which $\Omega, E \vdash M \equiv_S M'$, are identified in all computational and assertional judgements.⁷

Computational equality judgements are closed under the following rules:

$$\frac{}{\Omega, E \vdash M \xrightarrow{*}_E M} \quad (\text{RedRefl})$$

$$\frac{\Omega, E \vdash M \xrightarrow{*}_E M' \quad \Omega, E \vdash M' \xrightarrow{*}_E M''}{\Omega, E \vdash M \xrightarrow{*}_E M''} \quad (\text{RedTrans})$$

$$\frac{\Omega, E \vdash M_i \xrightarrow{*}_E M'_i \text{ for } i \in \{1, \dots, n\}}{\Omega, E \vdash f(M_1, \dots, M_n) \xrightarrow{*}_E f(M'_1, \dots, M'_n)} \quad (\text{RedApp})$$

$$\frac{\forall X . M = M' \text{ if } \phi_1 \wedge \dots \wedge \phi_m \in E^C \quad \Omega, E \vdash \theta(\phi_i) \text{ for } i \in \{1, \dots, m\}}{\Omega, E \vdash \theta(M) \xrightarrow{*}_E \theta(M')} \quad (\text{Red})$$

The computational equality judgements are reflexive, transitive, and enjoy the context closure property, but they are not symmetric, because they are intended to capture the directed notion of computation based on reduction. According to **Red** they are generated by the computational equations in E^C , and by our earlier convention they are defined modulo structural equality, which in view of **RedRefl** especially means that structural equality is subsumed by computational equality.

Operationally, the computational equality judgement $\Omega, E \vdash M \xrightarrow{*}_E N$ means that M is given and N is computed by a reduction using the computational equations in E^C modulo structural equality. From the viewpoint of the computational system this means that the algorithm should use reduction modulo structural equality to compute solutions to incomplete goals of the form $\Omega, E \vdash M \xrightarrow{*}_E ?$. Here, we define *reduction* as *rewriting* using an arbitrary strategy *without backtracking*, i.e. alternative reductions are not considered after a reduction step has taken place. The operational semantics does not fix the method that is used to find applicable instances of axioms. We also require that trivial non-termination caused by a repetition of *trivial reductions*, i.e. reductions that do not involve an application of **Red**, is avoided by the strategy.⁸

⁷A more precise formulation would make use of equivalence classes w.r.t. structural equality as in the formal system of rewriting logic presented in [Mes92].

⁸This and other requirements of the operational semantics could be made explicit in the formal system itself, but for the sake of a simpler and more natural formal system we prefer to state them externally.

Assertional judgements are closed under the following rules:

$$\frac{\forall X . M : s \text{ if } \phi_1 \wedge \dots \wedge \phi_m \in E^A \quad \Omega, E \vdash \theta(\phi_i) \text{ for } i \in \{1 \dots m\}}{\Omega, E \vdash \theta(M) : s} \quad (\text{Ass})$$

$$\frac{\Omega, E \vdash M' : s \quad \Omega, E \vdash M \xrightarrow{*}_E M'}{\Omega, E \vdash M : s} \quad (\text{MbRed})$$

$$\frac{}{\Omega, E \vdash M = M} \quad (\text{EqRefl})$$

$$\frac{\Omega, E \vdash M' = N' \quad \Omega, E \vdash M \xrightarrow{*}_E M' \quad \Omega, E \vdash N \xrightarrow{*}_E N'}{\Omega, E \vdash M = N} \quad (\text{EqRed})$$

The operational meaning of assertional judgements, which can be either assertional membership judgements $\Omega, E \vdash M : s$ or assertional equality judgements $\Omega, E \vdash M = N$, is that either M and s or M and N , respectively, are given and then the judgement is verified in a goal-oriented fashion. The rule **Ass** applies a membership axiom from E^A to solve a membership goal, and **MbRed** allows simplification of the goal using computational equality. Similarly, **EqRefl** uses reflexivity to solve an equational goal, and **EqRed** again allows equational simplification of the goal. By our convention that structurally equal terms are identified, the process of solving and reducing goals takes places modulo the structural equations.

Operationally, we require that the rules for assertional judgements are implemented in a goal-oriented fashion by an exhaustive strategy *with backtracking* for all rules with the exception of **MbRed** and **EqRed**, which use reduction and hence should not involve backtracking. This means in particular that in the case of **Ass** all possibilities to apply membership assertions admitted by the computational system are explored, provided this is not prevented by nonterminating conditions. To avoid trivial nontermination, we furthermore require that the reduction in **MbRed** and at least one of the reductions in **EqRed** is nontrivial in the sense defined earlier.

We finally require that all subgoals that arise from conditional axioms in **Red** or **Ass** are verified in the order in which the conditions appear in the axiom. This last requirement ensures that certain conjuncts of a condition are only verified after other conjuncts are known to be valid, a feature that is often needed in theories that are only partially terminating to ensure that the verification of conditions terminates.

The operational semantics abstracts from most implementation details and hence admits many possible algorithms with various possibilities for optimization. In

summary, it can be described as a combination of conditional reduction and exhaustive goal-oriented proof search which both take place modulo the structural equality. It is noteworthy that we are concerned with a truly bidirectional connection between these two mechanisms, because reduction can give rise to new goals which are solved by goal-oriented search, and goal-oriented search may itself rely on reduction of the goals.

Since the operational semantics of MEL is explained as an application of the rules of its underlying computational system, it is obviously sound, and since this computational system is a refinement of the deductive system of MEL we immediately obtain soundness of the operational semantics w.r.t. the deductive system. In the following we formulate two operational conditions, namely sort-decreasingness and confluence, which together imply completeness of the operational semantics w.r.t. the computational system⁹ for solving assertional goals, provided that completeness is not prevented by nontermination.¹⁰ We say that a MET \mathcal{E} is *sort-decreasing*¹¹ iff $\mathcal{E} \vdash M : s$ and $\mathcal{E} \vdash M \xrightarrow{*}_E M'$ then $\mathcal{E} \vdash M' : s$. We say that \mathcal{E} is (*equationally*) *confluent* iff $\mathcal{E} \vdash M \xrightarrow{*}_E M'$ and $\mathcal{E} \vdash M \xrightarrow{*}_E M''$ imply that there is a term M''' such that $\mathcal{E} \vdash M' \xrightarrow{*}_E M'''$ and $\mathcal{E} \vdash M'' \xrightarrow{*}_E M'''$. Sort-decreasingness ensures that terms can be reduced using an arbitrary strategy without losing sort information, and confluence ensures that reduction can take place using an arbitrary strategy without losing possible normal forms.

Since we have presented the operational semantics in terms of a formal system, *computations* in a MET are defined as the subset of goal-oriented derivations in the formal system that are admitted by the operational semantics. In particular, we say that a given reduction goal $\mathcal{E} \vdash M \xrightarrow{*}_E ?$ or a given assertional goal $\mathcal{E} \vdash M : s$ or $\mathcal{E} \vdash M = N$ is *terminating* iff all computations starting from this goal are terminating. It is important to be aware of the fact that there are two potential sources of nontermination, namely nonterminating reductions and nonterminating proof search, which are not independent as we saw. In contrast to [BJM97, BJM00] which is mainly concerned with so-called reductive theories, we do not impose termination as a general requirement, because we will also use theories in this thesis that are only partially terminating.

⁹A proof of completeness w.r.t. the deductive system under the assumption of reductivity, a syntactic condition for termination, can be found in [BJM00].

¹⁰Completeness for computational judgements holds in the sense that all normal forms in the computational system are actually computed in the operational semantics. Completeness can furthermore be violated due to nontermination, because we have avoided any restrictions on the strategy for the application of assertional axioms. In fact, for efficiency reasons the Maude engine uses depth-first search, which can cause incompleteness in nonterminating theories. It is noteworthy that this is the same kind of incompleteness that is present in the operational semantics of Prolog.

¹¹This definition of sort-decreasingness is simpler and slightly stronger than the corresponding notion in [BJM97, BJM00]. Furthermore, we do not consider an explicit subsort ordering, because subsort axioms can be directly expressed in MEL as explained earlier.

The operational semantics of MEL has been given as the partial specification of an algorithm. Obviously, we cannot expect that such an algorithm exists in the most general case of METs, because already the structural equality may be undecidable and not implementable. Therefore, we give in the following a number of *sufficient conditions* for executability, which are in fact also conditions that are required by the Maude implementation. A MET \mathcal{E} is said to be *executable* iff the following conditions are satisfied: (1) the structural equality of \mathcal{E} can be implemented by a suitable term representation so that an algorithm for matching modulo structural equality exists (more precisely, this has to be a terminating algorithm that finds all matches of subterms in a given term for a given pattern modulo structural equality); and (2) the set of computational equations and assertional axioms of \mathcal{E} is finite and they all satisfy the variable restriction¹² defined below so that all potentially applicable instances of these axioms can be found by the matching algorithm. We say that a computational axiom in E^C satisfies the *variable restriction* iff all variables occurring in the right hand side of the equation or in the condition also appear in the left hand side. Similarly, we say that an assertional axiom in E^A satisfies the *variable restriction* iff all variables occurring in the condition also appear in the conclusion.

Constraints

In contrast to an entirely loose or entirely initial semantics of membership equational theories, in practice a mixed specification style is used, where certain subtheories are intended to be equipped with initial interpretations or certain subtheories are interpreted free over their parameter theories. To make such restrictions on the models explicit in the specification we enrich a membership equational theory by initiality and freeness constraints [DM99, GB92], and refer to these enriched theories as *membership equational specifications (MESs)*.

From a model-theoretic point of view, constraints are axioms that are treated in full analogy to membership or equational axioms, i.e. as sentences that have to be valid in all models. Hence, the models of a MES are algebras which satisfy all the given initiality and freeness constraints. Given a MES model, a model of a subtheory is obtained by its associated forgetful functor \mathbf{U}_K for K the corresponding subtheory inclusion. In particular, this means that a model induces a unique interpretation for each subtheory, which is the justification for the condition on ϵ below. The notion of constraint we use here is a special case of the notion proposed in [GB92], where initiality constraints are seen as a special case of freeness constraints.

¹²In its most recent version, Maude imposes an even weaker variable restriction for executability due to the admissibility of conditions with *matching equations* [CDE⁺00a]. These are equations of the form $P := M$ for a pattern P and a term M , and operationally the variables in P are instantiated by matching against a reduced version of M .

Let $J : \mathcal{T}'' \hookrightarrow \mathcal{T}'$ and $I : \mathcal{T}' \hookrightarrow \mathcal{T}$ be MET inclusions. A *constraint* for \mathcal{T} can take one of the following two forms: (1) \mathcal{T}' is **initial** or (2) \mathcal{T}' is **free over** \mathcal{T}'' . A *membership equational specification (MES)* \mathcal{E} is a MET $\mathcal{T}_{\mathcal{E}}$ together with a set $C_{\mathcal{E}}$ of constraints for $\mathcal{T}_{\mathcal{E}}$.

Let A be a \mathcal{T} -algebra. We define *validity* of a constraint as follows: (1) the constraint \mathcal{T}' is **initial** is *valid* iff the unique morphism from $\mathbf{I}(\mathcal{T}')$ to $A|_{\mathcal{T}'}$ is an isomorphism, and (2) the constraint \mathcal{T}' is **free over** \mathcal{T}'' is *valid* iff $\epsilon_J(A|_{\mathcal{T}'}) : \mathbf{F}_J(A|_{\mathcal{T}''}) \rightarrow A|_{\mathcal{T}'}$ is an isomorphism. Given a MES \mathcal{E} with an underlying MET $\mathcal{T}_{\mathcal{E}}$, an \mathcal{E} -*algebra* is a $\mathcal{T}_{\mathcal{E}}$ -algebra A such that each constraint in $C_{\mathcal{E}}$ is valid in A . In analogy to the definition of validity in METs, we say that ψ is *valid* in \mathcal{E} (written $\mathcal{E} \models \psi$) iff $A \models \psi$ for each \mathcal{E} -algebra A , and given a set E of Ω -axioms, we say that E is *valid* in \mathcal{E} (written $\mathcal{E} \models E$) iff $\mathcal{E} \models \psi$ for each $\psi \in E$.

In analogy to METs we furthermore define: Given MESs \mathcal{E} and \mathcal{E}' , a *MES morphism* $H : \mathcal{E} \rightarrow \mathcal{E}'$ is a morphism $H : \mathcal{E} \rightarrow \mathcal{E}'$ such that $\mathcal{E}' \models H(C_{\mathcal{E}})$, i.e., the constraints $H(C_{\mathcal{E}})$ are valid in all \mathcal{E}' -algebras, where H is lifted to constraints in the natural way. \mathcal{E} is a *subspecification* of \mathcal{E}' , written $\mathcal{E} \hookrightarrow \mathcal{E}'$, iff there is a MES morphism $J : \mathcal{E} \hookrightarrow \mathcal{E}'$ that is an inclusion. MESs together with their morphisms form a category **MES** and the category of \mathcal{E} -algebras $\mathbf{Mod}(\mathcal{E})$ is the full subcategory of $\mathbf{Mod}(\mathcal{T}_{\mathcal{E}})$ that contains only \mathcal{E} -algebras. Again, $\mathbf{Mod}(\mathcal{E})$ is called the *model-theoretic semantics* of \mathcal{E} . Each MES morphism $H : \mathcal{E} \rightarrow \mathcal{E}'$ induces an obvious forgetful functor $\mathbf{Mod}(H) : \mathbf{Mod}(\mathcal{E}') \rightarrow \mathbf{Mod}(\mathcal{E})$ that we also write as \mathbf{U}_H . Again, we have a contravariant functor $\mathbf{Mod} : \mathbf{MES} \rightarrow \mathbf{Cat}^{\text{op}}$ that generalizes $\mathbf{Mod} : \mathbf{MET} \rightarrow \mathbf{Cat}^{\text{op}}$. Given an inclusion $I : \mathcal{E}' \hookrightarrow \mathcal{E}$ and an \mathcal{E} -algebra A , we also write $A|_{\mathcal{E}'}$ instead of $\mathbf{U}_I(A)$.

2.4.5 Rewriting Logic

Equational specification languages do not have explicit notions of state and change. Such notions are however desirable to model reactive computational systems with important aspects such as concurrency, nondeterminism and non-termination. Although equational logic can be equipped with an operational semantics, the operational aspects are not visible in the abstract model-theoretic semantics of equational logic, since by definition the interpretations of the left hand side and the right hand side of an equation are identified. Furthermore, the completeness of the operational semantics relies on certain confluence and termination requirements, which constrain the generality of the reduction relation in a way that makes it incompatible with the general dynamics of reactive systems that are typically nondeterministic and nonterminating. As already pointed out in Section 2.2, we adopt the fairly general view that a reactive computational system can be modeled as a (structured) state space together with (structured) transitions between states. In order to capture state changes in an equational framework we employ *rewrite theories*, which from different perspectives can be

seen as a refinement or as a generalization of equational theories. Rewriting logic is a generalization in the sense that it contains an equational logic as a sublogic, but it is a refinement in the sense that its central notion of a directed rewrite emerged from the notion of equality by giving up the symmetry law, in complete analogy to linear logic [Gir87], which arises by giving up some structural laws of propositional logic. The underlying theory of rewriting logic has been developed in [Mes92] for the simple case of single-sorted pure equational logic and is briefly reviewed next, before we generalize it to MEL as an underlying logic.

A *simple rewrite theory* consists of a signature Ω , a set of unconditional equations E , called *structural axioms* or *structural equations*, a set of labels Lab , and a set of labeled *rewrite axioms* R , also called *rewrite rules*, of the form¹³ $\forall X . l : L \rightarrow R$. The signature Ω together with E constitutes a single-sorted equational theory. The equational axioms in E are called structural equations (relative to the rewrite rules), since together with the signature they specify the structure of objects that are transformed by the rewrite rules. The algebraic models of rewrite theories are enriched transition categories, which are obtained by equipping the algebras which are models of the underlying equational theory with a suitable transition relation according to the given rewrite rules. More precisely, a *rewrite axiom* of the form $l : L \rightarrow R$ states that a part of a system's state which matches L , can be *rewritten*¹⁴ in one step, i.e. subjected to a local state transition, by replacing this part by the corresponding instance of R . The sentences of rewriting logic are so-called *rewrites* of the form $P : M \rightarrow N$. Such a sentence is valid iff M *rewrites* to N in a general sense explained below, and this rewrite is witnessed by the *proof* P . Proofs are generated from identity proofs (denoted by id_M), (well-formed) sequential composition (denoted by $;$), parallel composition according to the operations in Ω , and basic proofs (denoted by $l(Q_1, \dots, Q_n)$, where l is the label of a rule and Q_1, \dots, Q_n are proofs of potential concurrent rewrites on the objects transformed by the rule). Proofs in rewriting logic are furthermore equipped with an equality which reflects the notion of concurrent rewrite computation in the most abstract axiomatic way.

As an example of a simple rewrite theory consider the following theory of a binary nondeterministic choice operator. Again, we employ the syntax of Maude, where a rewrite rule is introduced by the keyword `r1` (or `cr1` for conditional rules) followed by a label in square brackets.

```
sort Id .
```

¹³In general, rewriting logic admits conditional rewrite axioms where the conditions are conjunctions of rewrites. In this thesis, however, we will not use rewrites in conditions. Indeed, in rewriting logic over MEL we will only allow conditions containing equations and memberships.

¹⁴We are using the terms matching and rewriting here in a model-dependent semantic sense, which correspond to the usual syntactic notion of matching and rewriting modulo E for initial term models.

```

ops a b c d : -> Id .

op _?_ : Id Id -> Id .

rl [left]   : x ? y => x .
rl [right]  : x ? y => y .

```

Some rewrites derivable in this theory are:

```

ida : a → a
idb : b → b
idc : c → c
idd : d → d
left(ida, idb) : (a ? b) → a
right(idc, idd) : (c ? d) → d
right(ida, idd) : (a ? d) → d
(left(ida, idb) ? right(idc, idd)) : ((a ? b) ? (c ? d)) → (a ? d)
(left(ida, idb) ? right(idc, idd)) ; right(ida, idd) :
    ((a ? b) ? (c ? d)) → d

```

So far we have given a simplified overview of rewriting logic over a single-sorted equational logic. From a more general point of view, rewriting logic can be regarded as being parameterized by its underlying equational logic, which can be single-sorted, many-sorted, order-sorted and so on. In the design of the Maude language [CDE⁺99a, CDE⁺00b], *membership equational logic* has been chosen as the underlying equational logic. Also in this thesis, we usually use rewriting logic to refer to a version of rewriting logic over membership equational logic, which we abbreviate as **RWL_{MEL}** or just **RWL** and which we introduce subsequently in some detail.

Syntax

Below we give the definition of rewrite theories, denoted by \mathcal{R} , based on an underlying MET $\mathcal{E}_{\mathcal{R}}$ with a distinguished *data subtheory* $\mathcal{E}_{\mathcal{R}}^D$. The data subtheory specifies the *data space*, whereas the remaining part of $\mathcal{E}_{\mathcal{R}}$ specifies the *state space* by introducing the *rewrite kinds*, i.e. kinds whose elements correspond to states and therefore can be rewritten by rewrite rules. In the context of this thesis, the state space is always specified in a purely equational way, on top of the data space theory, which can make use of membership equational logic in its full generality. In addition to the *static part*, i.e. the data and state space theory in membership equational logic, a rewrite theory has a *dynamic part* given by labeled rewrite rules which specify all possible transitions that can take place between states.

A *rewrite theory (RWT)* \mathcal{R} consists of a MET $\mathcal{E}_{\mathcal{R}}$ with a distinguished *data subtheory* $\mathcal{E}_{\mathcal{R}}^D$, a set of *labels* $Lab_{\mathcal{R}}$, and a set of *rewrite axioms* $R_{\mathcal{R}}$, also called *rewrite rules*, of the form $\forall X . l : M \rightarrow N$ if $\phi_1 \wedge \dots \wedge \phi_n$ where $l \in Lab_{\mathcal{R}}$, $\phi_1 \wedge \dots \wedge \phi_n$ is a $\mathcal{E}_{\mathcal{R}}$ -condition over X , and $M, N \in Trm_{\mathcal{R}}(X)_k$ in $\mathcal{E}_{\mathcal{R}}$ for a rewrite kind k . If $R_{\mathcal{R}}$ contains such an axiom we also write $l : \bar{k} \rightarrow k$ assuming $X = \bar{x} : \bar{k}$. We generally assume that each label in $Lab_{\mathcal{R}}$ is used in exactly one rewrite rule from $R_{\mathcal{R}}$. *Rewrite kinds* are the kinds introduced in $\mathcal{E}_{\mathcal{R}}$, i.e. those kinds which are not already contained in $\mathcal{E}_{\mathcal{R}}^D$. In the context of a RWT \mathcal{R} we usually use l to range over $Lab_{\mathcal{R}}$.

The kinded set of (*raw*) *proof terms* $Prf_{\mathcal{R},k}$ is inductively defined as follows: (1) each term $M \in Trm_{\mathcal{R},k}$ is in $Prf_{\mathcal{R},k}$; (2) the *identity proof* id_M is in $Prf_{\mathcal{R},k}$ for each $M \in Trm_{\mathcal{R},k}$; (3) the *sequential proof* $P; Q$ is in $Prf_{\mathcal{R},k}$ for all $P, Q \in Prf_{\mathcal{R},k}$; (4) the *parallel proof* $f(P_1, \dots, P_n)$ is in $Prf_{\mathcal{R},k}$ for $f : \bar{k} \rightarrow k$ with $\bar{k} = k_1, \dots, k_n$ and $P_i \in Prf_{\mathcal{R},k_i}$; (5) the *basic proof* $l(P_1, \dots, P_n)$ is in $Prf_{\mathcal{R}}$ for $l : \bar{k} \rightarrow k$ with $\bar{k} = k_1, \dots, k_n$ and $P_i \in Prf_{\mathcal{R},k_i}$. In the context of a RWT \mathcal{R} we typically use variables P, Q , and R to range over proof terms in $Prf_{\mathcal{R}}$.

A *kinded proof substitution* for X is a kinded function $\rho = (\rho_k)_{k \in Kind}$ associating to each $x \in X_k$ an element $\rho_k(x) \in Prf_{\Omega,k}$. It is extended to terms over X as follows: (1) $\rho_k(c) = id_c$ for $c : \rightarrow k$; and (2) $\rho_k(f(M_1, \dots, M_n)) = f(\rho_{k_1}(M_1), \dots, \rho_{k_n}(M_n))$ for $f : \bar{k} \rightarrow k$ with $\bar{k} = k_1, \dots, k_n$ and $M_i \in Trm_{\Omega}(X)_{k_i}$. Instead of $\rho_k(M)$ for $M \in Trm_{\Omega}(X)_k$ we also use the notation $\rho(M)$.

Given two RWTs \mathcal{R} and \mathcal{R}' , a *RWT morphism* $H : \mathcal{R} \rightarrow \mathcal{R}'$ consists of a MET morphism $H_{\mathcal{E}} : \mathcal{E}_{\mathcal{R}} \rightarrow \mathcal{E}_{\mathcal{R}'}$ and a function $H_{Lab} : Lab_{\mathcal{R}} \rightarrow Lab_{\mathcal{R}'}$ such that $H_{\mathcal{E}}$ has a restriction $H_D : \mathcal{E}_{\mathcal{R}}^D \rightarrow \mathcal{E}_{\mathcal{R}'}^D$ to the data subtheory and for each rewrite axiom $r \in R_{\mathcal{R}}$ there is a rewrite axiom in $R_{\mathcal{R}'}$ that is structurally equal to $H(r)$ up to a renaming of the variables, where H is lifted to rewrite axioms in the obvious homomorphic way. RWTs together with their morphisms form a category that is denoted by **RWT**.

Recall that METs have been enriched by constraints resulting in membership equational specifications (MESs). In complete analogy we generalize the definition of a RWT to a *rewrite specification (RWS)* \mathcal{R} , which has an *underlying MES* $\mathcal{E}_{\mathcal{R}}$ instead of an underlying MET and a distinguished *data subspecification* $\mathcal{E}_{\mathcal{R}}^D$ of $\mathcal{E}_{\mathcal{R}}$ instead of a data subtheory. Generalizing the notion of RWT morphisms to *RWS morphisms* correspondingly, RWSs form a category that is denoted by **RWS**.

Model-theoretic Semantics

The model-theoretic semantics of an arbitrary rewrite theory is given by a category of enriched transition categories and can be informally summarized as follows. At the end of this section, we give a more precise definition of the model-

theoretic semantics of REL by means of a mapping from RWL into MEL.

A model of a rewrite theory/specification \mathcal{R} is a model A of the underlying MET/MES $\mathcal{E}_{\mathcal{R}}$ that is enriched as follows: For each kind k the set $\llbracket k \rrbracket_A$ is equipped with the structure of an enriched transition category. The corresponding rewrites $l : M \rightarrow M'$ in the rewrite rules of \mathcal{R} are interpreted by means of a ternary predicate which captures the arrows of the corresponding transition category. This interpretation is lifted to general rewrites $P : M \rightarrow M'$, where P is a rewrite proof term, as follows. Sequential composition of proof terms is interpreted by arrow composition, and parallel composition operators are interpreted by enriching the category with an algebraic structure as it has been specified in $\mathcal{E}_{\mathcal{R}}$. In order to be a model of \mathcal{R} , the transition category has to satisfy a number of requirements, namely, functoriality w.r.t. the algebraic structure, the equations in $\mathcal{E}_{\mathcal{R}}$ lifted to arrows, and for each rewrite axiom in \mathcal{R} the so-called decomposition and exchange laws which are defined below.

The model-theoretic semantics of \mathcal{R} is the category of such models and the morphisms of this category are functors which preserve the additional algebraic structure of the transition categories. Usually we are interested in the subcategory of models which are free over the data subtheory $\mathcal{E}_{\mathcal{R}}^D$. In the important case where $\mathcal{E}_{\mathcal{R}}^D$ is interpreted initially, this corresponds to the initial model described in [Mes92].

Deductive System

As in the case of MEL, we distinguish between two formal systems, the deductive and the computational system of RWL. We first define the *deductive system of RWL* as an extension of the deductive system of MEL. Again we use Ω to range over MEL signatures, E to range over sets of axioms, and we use R to range over sets of rewrite axioms, so that together Ω , E , and R define a RWT. The additional sentences are (*deductive*) *rewrite judgements* of the form $\Omega, E, R \vdash P : M \rightarrow M'$ and (*deductive*) *equality judgements* $\Omega, E, R \vdash P = P'$ on proof terms. Furthermore, we add the following rules. In these rules we use $X = \bar{x}^n : \bar{k}^n$ to range over a kinded variable set. We use θ and θ' to range over kinded substitutions for X and ρ to range over kinded proof substitutions for X . For better readability, we write $M[\rho]$ to abbreviate $\rho(M)$, $l[\rho]$ to abbreviate $l(\rho(\bar{x}_1), \dots, \rho(\bar{x}_n))$, and $l[\theta]$ to abbreviate $l(\text{id}_{\theta(\bar{x}_1)}, \dots, \text{id}_{\theta(\bar{x}_n)})$.

$$\frac{\begin{array}{l} \forall X . l : M \rightarrow M' \text{ if } \phi_1 \wedge \dots \wedge \phi_m \in R \\ \Omega, E \vdash \theta(\phi_i) \text{ for } i \in \{1, \dots, m\} \\ \Omega, E, R \vdash \rho(\bar{x}_i) : \theta(\bar{x}_i) \rightarrow \theta'(\bar{x}_i) \end{array}}{\Omega, E, R \vdash l[\rho] : M[\theta] \rightarrow M'[\theta']} \quad (\text{Rew})$$

$$\frac{}{\Omega, E, R \vdash \text{id}_M : M \rightarrow M} \quad (\text{RewRefI})$$

$$\frac{\Omega, E, R \vdash P : M \rightarrow M' \quad \Omega, E \vdash Q : M' \rightarrow M''}{\Omega, E, R \vdash P; Q : M \rightarrow M''} \quad (\text{RewTrans})$$

$$\frac{\Omega, E, R \vdash P_i : M_i \rightarrow M'_i \text{ for } i \in \{1, \dots, n\}}{\Omega, E, R \vdash f(P_1, \dots, P_n) : f(M_1, \dots, M_n) \rightarrow f(M'_1, \dots, M'_n)} \quad (\text{RewApp})$$

$$\frac{\Omega, E, R \vdash P : M' \rightarrow M'' \quad \Omega, E, R \vdash P = P' \quad \Omega, E, R \vdash M = M' \quad \Omega, E, R \vdash M'' = M'''}{\Omega, E, R \vdash P' : M \rightarrow M'''} \quad (\text{RewModulo})$$

To state the rule **RewModulo**, the equality judgements have been extended from terms to proof terms. The corresponding equality rules are given below. **Eqn** states that equality on proof terms is generated by the equational axioms inherited from the underlying MET, but as mentioned earlier additional equations are imposed, namely the equational axioms of categories, functoriality, and the decomposition and exchange laws.

$$\frac{\forall X . M = N \text{ if } \phi_1 \wedge \dots \wedge \phi_m \in R \quad \Omega, E \vdash \theta(\phi_i) \quad \Omega, E \vdash \theta'(\phi_i) \text{ for } i \in \{1, \dots, m\} \quad \Omega, E, R \vdash M[\rho] : M[\theta] \rightarrow M[\theta'] \quad \Omega, E, R \vdash N[\rho] : N[\theta] \rightarrow N[\theta']}{\Omega, E, R \vdash M[\rho] = N[\rho]} \quad (\text{Eqn})$$

$$\frac{\Omega, E, R \vdash P : M \rightarrow M'}{\Omega, E, R \vdash \text{id}_M; P = P \quad \Omega, E, R \vdash P; \text{id}_{M'} = P} \quad (\text{Id1/Id2})$$

$$\frac{\Omega, E, R \vdash P : M \rightarrow M' \quad \Omega, E, R \vdash Q : M' \rightarrow M'' \quad \Omega, E, R \vdash R : M'' \rightarrow M'''}{\Omega, E, R \vdash (P; Q); R = P; (Q; R)} \quad (\text{Assoc})$$

$$\frac{}{\Omega, E, R \vdash f(\text{id}_{M_1}, \dots, \text{id}_{M_n}) = \text{id}_{f(M_1, \dots, M_n)}} \quad (\text{Functor1})$$

$$\frac{\Omega, E, R \vdash P_i : M_i \rightarrow M'_i \quad \Omega, E, R \vdash Q_i : M'_i \rightarrow M''_i}{\Omega, E, R \vdash f(P_1; Q_1, \dots, P_n; Q_n) = f(P_1, \dots, P_n); f(Q_1, \dots, Q_n)} \quad (\text{Functor2})$$

$$\frac{\forall X . l : N \rightarrow N' \text{ if } \phi_1 \wedge \dots \wedge \phi_m \in R \quad \Omega, E \vdash \theta(\phi_i) \text{ for } i \in \{1, \dots, m\} \quad \Omega, E, R \vdash \rho(\bar{x}_i) : \theta(\bar{x}_i) \rightarrow \theta'(\bar{x}_i) \text{ for } i \in \{1, \dots, n\}}{\Omega, E, R \vdash l[\rho] = l[\theta]; N'[\rho]} \quad (\text{Decompose})$$

$$\frac{\forall X . l : N \rightarrow N' \text{ if } \phi_1 \wedge \dots \wedge \phi_m \in R \quad \Omega, E \vdash \theta(\phi_i) \quad \Omega, E \vdash \theta'(\phi_i) \text{ for } i \in \{1, \dots, m\} \quad \Omega, E, R \vdash \rho(\bar{x}_i) : \theta(\bar{x}_i) \rightarrow \theta'(\bar{x}_i)}{\Omega, E, R \vdash l[\theta]; N'[\rho] = N[\rho]; l[\theta']} \quad (\text{Exchange})$$

Due to the properties of computational rewrite judgements, we also refer to a derivable judgement $\Omega, E, R \vdash P : M \rightarrow M'$ as a *general (concurrent) rewrite*. However, more elementary notions of rewriting can be useful, and they can be expressed by imposing certain restrictions on the application of the inference rules above. We say that a general rewrite $\Omega, E, R \vdash P : M \rightarrow M'$ is a *proper (concurrent) rewrite* if the rule **Rew** is used at least once in its derivation. It is called a *one-step parallel rewrite* if the rule **Rew** is used at least once in its derivation and **RewTrans** is not used. It is said to be a *one-step sequential rewrite* if the rule **Rew** is used exactly once in its derivation and **RewTrans** is not used. Finally, we call it a *one-step direct rewrite* if the rule **Rew** is used exactly once in its derivation and **RewTrans** and **RewApp** are not used. To reflect these notions inside the deductive system, we introduce additional judgements of the form $\Omega, E, R \vdash P : M \overset{\pm}{\rightarrow}_R M'$, $\Omega, E, R \vdash P : M \overset{=}{\rightarrow}_R M'$, $\Omega, E, R \vdash P : M \rightarrow_R M'$, and $\Omega, E, R \vdash P : M \Rightarrow_R M'$ and we assume corresponding rules (without giving them here) so that these judgements are derivable iff $\Omega, E, R \vdash P : M \rightarrow M'$ is a proper concurrent rewrite, a one-step parallel rewrite, a one-step sequential rewrite, or a one-step direct rewrite, respectively. Often rewrite proof terms are not needed, and we therefore introduce *abstract rewrite judgements* $\Omega, E, R \vdash M \rightarrow M'$. Again we assume a rule to ensure that an abstract rewrite is derivable iff $\Omega, E, R \vdash P : M \rightarrow M'$ is derivable for some P .

Computational System

The computational system of RWL is the basis for the operational semantics of RWL, which is again a partial specification of an algorithm to execute RWTs. Again, the computational system is a refinement of the deductive system so that soundness is satisfied by construction.

The *computational system of RWL* extends the computational system of MEL as follows. The additional sentences are (*computational*) *rewrite judgements* of the form $\Omega, E, R \vdash P : M \overset{*}{\rightarrow}_R M'$, and the additional rules are given below. They correspond to those of the deductive system with two exceptions: first, the rule **RewModulo** is oriented in the computational system, and second, a computational equality judgement for rewrite proofs and corresponding rules are not needed, because computations do not take place inside proofs. Again we use θ to range over kinded substitutions for X , and ρ to range over kinded proof substitutions for X .

$$\frac{\begin{array}{l} \forall X . l : M \rightarrow M' \text{ if } \phi_1 \wedge \dots \wedge \phi_m \in R \\ \Omega, E \vdash \theta(\phi_i) \text{ for } i \in \{1, \dots, m\} \\ \Omega, E, R \vdash \rho(\bar{x}_i) : \theta(\bar{x}_i) \overset{*}{\rightarrow}_R \theta'(\bar{x}_i) \end{array}}{\Omega, E, R \vdash l[\rho] : M[\theta] \overset{*}{\rightarrow}_R M'[\theta']} \quad (\text{Rew})$$

$$\begin{array}{c}
\frac{}{\Omega, E, R \vdash \text{id}_M : M \xrightarrow{*}_R M} \quad \text{(RewRefl)} \\
\\
\frac{\Omega, E, R \vdash P : M \xrightarrow{*}_R M' \quad \Omega, E \vdash Q : M' \xrightarrow{*}_R M''}{\Omega, E, R \vdash P; Q : M \xrightarrow{*}_R M''} \quad \text{(RewTrans)} \\
\\
\frac{\Omega, E, R \vdash P_i : M_i \xrightarrow{*}_R M'_i \text{ for } i \in \{1, \dots, n\}}{\Omega, E, R \vdash f(P_1, \dots, P_n) : f(M_1, \dots, M_n) \xrightarrow{*}_R f(M'_1, \dots, M'_n)} \quad \text{(RewApp)} \\
\\
\frac{\Gamma \vdash P : M' \xrightarrow{*}_R M'' \quad \Gamma \vdash M \rightarrow_E M' \quad \Gamma \vdash M'' \rightarrow_E M'''}{\Gamma \vdash P : M \xrightarrow{*}_R M'''} \quad \text{(RewModulo)}
\end{array}$$

Operationally, the computational rewrite judgement $\Omega, E, R \vdash P : M \xrightarrow{*}_R M'$ means that M is given and M' is computed from M by rewriting modulo structural and computational equality using the computational rewrite axioms in Γ , and a proof term P is constructed to witness this rewrite. From the viewpoint of the computational system this means that the algorithm should use rewriting modulo structural and computational equality to compute certain solutions to incomplete goals of the form $\Omega, E, R \vdash ? : M \xrightarrow{*}_R ?$. By our earlier convention, rewriting takes place modulo structural equality. The rule **RewModulo** expresses furthermore that rewriting takes place modulo computational equality, and we impose the usual operational requirement to avoid trivial nontermination, namely that at least one of the two reductions involved is nontrivial. We also require that trivial nontermination caused by a repetition of *trivial rewrites*, i.e. rewrites that do not involve an application of **Rew**, is avoided by the strategy. However, in contrast to reduction, we do not impose any further restrictions on the rewrite strategy, which may or may not involve backtracking, and is in general application-dependent. As before, we require that all subgoals that arise from conditional axioms in **Rew** are verified in the order in which the conditions occur in the axiom. In summary, the operational semantics of RWL is explained by applying the computational equations *and* the rewrite rules modulo structural equality. In this way we can achieve the same effect as if we were regarding all computational equations as structural equations, provided that a suitable coherence requirement between computational equations and rewrite rules is satisfied, which is given below.

As in the case of MEL, the operational semantics is explained as an application of the rules of its underlying computational system, and hence it is obviously sound. Generalizing the MEL case we now formulate a condition called coherence, which ensures completeness of the operational semantics w.r.t. its underlying computational system for rewrite goals,¹⁵ provided that completeness is not prevented

¹⁵We define completeness for rewrite goals modulo the deductive equality on rewrite proof

by nontermination or by a nonexhaustive rewrite strategy. For the following definitions we assume that \mathcal{R} is a RWT that is *sort-decreasing* and *equationally confluent*, i.e. the underlying equational theory $\mathcal{E}_{\mathcal{R}}$ has these properties. We then say that \mathcal{R} is *coherent*¹⁶ iff $\mathcal{R} \vdash M \xrightarrow{*}_{\mathcal{E}} M'$ and $\mathcal{R} \vdash P : M \xrightarrow{*}_{\mathcal{R}} N$ implies that there are terms M'', P'' and N'' such that $\mathcal{R} \vdash M' \xrightarrow{*}_{\mathcal{E}} M'', \mathcal{R} \vdash P = P'', \mathcal{R} \vdash N = N''$ and $\mathcal{R} \vdash P'' : M'' \xrightarrow{*}_{\mathcal{R}} N''$. Coherence ensures that reduction using computational equations can take place without losing possible rewrites. A stronger operational property, that may be satisfied in specific applications, is confluence for rewrites. We say that \mathcal{R} is (*rewrite-*)*confluent* iff \mathcal{R} is coherent and $\mathcal{R} \vdash P' : M \xrightarrow{*}_{\mathcal{R}} M'$ and $\mathcal{R} \vdash P'' : M \xrightarrow{*}_{\mathcal{R}} M''$ implies that there are Q', Q'' and N'' , such that $\mathcal{R} \vdash Q' : M' \xrightarrow{*}_{\mathcal{R}} N'', \mathcal{R} \vdash Q'' : M'' \xrightarrow{*}_{\mathcal{R}} N''$, and $\mathcal{R} \vdash P'; Q' = P''; Q''$. It is important to point out that the notions of coherence and confluence given here are quite strong, and can often be replaced by weaker notions, which still can be sufficient to ensure completeness in a weaker, possibly application-specific sense, and/or under certain rewrite strategies. An obvious possibility is for instance the use of abstract rewrites instead of rewrites in these definitions, which could be justified in applications where rewrite proof terms are not relevant.

As in MEL, *computations* in a rewrite theory are defined as goal-oriented derivations in the computational system that are admitted by the operational semantics. We say that a rewrite goal $\Omega, E, R \vdash ? : M \xrightarrow{*}_{\mathcal{R}} ?$ is *terminating* if all computations starting from this goal are terminating. Usually, the set of computations is restricted by a given rewrite strategy, so that we define computations and derived notions such as termination *relative to a given strategy*.

Furthermore, we extend the notion of executability from MET to RWT as follows. We say that a RWT \mathcal{R} is *weakly executable* iff the underlying MET of \mathcal{R} is executable and the set of rewrite rules $R_{\mathcal{R}}$ is finite. It is said to be *strongly executable* iff additionally the *variable restriction* is satisfied for all rewrite rules in $R_{\mathcal{R}}$, i.e., all variables occurring in the right hand side or in the condition of the rewrite rule also appear in the left hand side. In the case of strong executability an implementation can use matching modulo structural equations to find all potential instantiations for the variables of rewrite rules, whereas in the case of weak executability a rewrite strategy is needed to take care of this. Such rewrite strategies are supported by the Maude engine as we will briefly discuss below.

Mapping Rewriting Logic into Membership Equational Logic

In order to obtain a precise definition of the model-theoretic semantics of a RWT \mathcal{R} , it is convenient to define the model-theoretic semantics of \mathcal{R} by means of a

terms.

¹⁶Since we take proofs into account, our notion of coherence is stronger than equational coherence in [Vir95, Vir] and an equivalent notion in [Mes93] for an underlying confluent and terminating equational theory.

MET $\mathbf{E}(\mathcal{R})$ which has already a standard model-theoretic semantics in terms of $\mathbf{E}(\mathcal{R})$ -algebras. The *membership equational presentation* of a RWT \mathcal{R} is a MET $\mathbf{E}(\mathcal{R})$ that extends $\mathcal{E}_{\mathcal{R}}$, the underlying MET of \mathcal{R} , by the following:

1. for each kind of $\mathcal{E}_{\mathcal{R}}$ we add a new kind $[\mathbf{RawPrf}]_k$ together with new operator symbols called *identity and sequential proof constructors*

$$\begin{aligned} \text{id} &: k \rightarrow [\mathbf{RawPrf}]_k, \\ \text{;-} &: [\mathbf{RawPrf}]_k [\mathbf{RawPrf}]_k \rightarrow [\mathbf{RawPrf}]_k; \end{aligned}$$

2. a new operator symbol called *parallel proof constructor*

$$f : [\mathbf{RawPrf}]_{\bar{k}_1} \dots [\mathbf{RawPrf}]_{\bar{k}_n} \rightarrow [\mathbf{RawPrf}]_k$$

for each operator $f : \bar{k}^n \rightarrow k$ in $\mathcal{E}_{\mathcal{R}}$;

3. a new operator symbol called *basic proof constructor*

$$l : [\mathbf{RawPrf}]_{\bar{k}_1} \dots [\mathbf{RawPrf}]_{\bar{k}_n} \rightarrow [\mathbf{RawPrf}]_k$$

for each label $l : \bar{k}^n \rightarrow k$ in $R_{\mathcal{R}}$;

4. a kind $[\mathbf{Prf}]_k$ with a sort \mathbf{Prf}_k and an operator symbol

$$\text{-; -} \rightarrow \text{-} : [\mathbf{RawPrf}]_k k k \rightarrow [\mathbf{Prf}]_k;$$

5. membership axioms representing the rules of the deductive system of RWL specialized to the underlying MET $\mathcal{E}_{\mathcal{R}}$, where a rewrite

$$P : M \rightarrow N \quad \text{is represented as a membership} \quad (P : M \rightarrow N) : \mathbf{Prf}_k.$$

if $M, N \in \text{Trm}_{\mathcal{R},k}$ and $P \in \text{Prf}_{\mathcal{R},k}$.

\mathbf{E} can be extended to a functor $\mathbf{E} : \mathbf{RWT} \rightarrow \mathbf{MET}$ in the obvious way.

By means of \mathbf{E} we can lift models and the initiality and freeness results from MEL to RWL which is done next. We define \mathcal{R} -algebras as $\mathbf{E}(\mathcal{R})$ -algebras, and by composing $\mathbf{E} : \mathbf{RWT} \rightarrow \mathbf{MET}$ with the functor $\mathbf{Mod} : \mathbf{MET} \rightarrow \mathbf{Cat}^{\text{op}}$ we obtain $\mathbf{Mod} \circ \mathbf{E} : \mathbf{RWT} \rightarrow \mathbf{Cat}^{\text{op}}$ which is also denoted $\mathbf{Mod} : \mathbf{RWT} \rightarrow \mathbf{Cat}^{\text{op}}$. As usual we write \mathbf{U}_H for $\mathbf{Mod}(H)$ given an RWS morphism H . By defining $\mathbf{I}(\mathcal{R})$ as $\mathbf{I}(\mathbf{E}(\mathcal{R}))$ and $\mathbf{F}_H : \mathbf{Mod}(\mathcal{R}) \rightarrow \mathbf{Mod}(\mathcal{R}')$ as $\mathbf{F}_{\mathbf{E}(H)} : \mathbf{Mod}(\mathbf{E}(\mathcal{R})) \rightarrow \mathbf{Mod}(\mathbf{E}(\mathcal{R}'))$ we can lift Theorem 2.4.1 from MEL to RWL resulting in the following

Theorem 2.4.2 (Initial and Free Models) RWTs have initial and free models. In fact, given a RWT \mathcal{R}' there exists an initial \mathcal{R}' -algebra, written $\mathbf{I}(\mathcal{R}')$. More generally, given a RWT morphism $H : \mathcal{R} \rightarrow \mathcal{R}'$ between RWTs \mathcal{R} and \mathcal{R}' there exists a free functor $\mathbf{F}_H : \mathbf{Mod}(\mathcal{R}) \rightarrow \mathbf{Mod}(\mathcal{R}')$, i.e. a functor that is left adjoint to \mathbf{U}_H . In the following we write η_H and ϵ_H for the unit and counit, respectively, of this adjunction, i.e., we have natural transformations $\eta_H(A) : A \rightarrow \mathbf{U}_H(\mathbf{F}_H(A))$ for \mathcal{E} -algebras A and $\epsilon_H(A') : \mathbf{F}_H(\mathbf{U}_H(A')) \rightarrow A'$ for \mathcal{R}' -algebras A' .

Among all models of a RWT \mathcal{R} we are usually either interested in the initial model $\mathbf{I}(\mathcal{R})$ or in the model $\mathbf{F}_K(A)$ that is free over a model A of the data subtheory $\mathcal{E}_{\mathcal{R}}^D$, which is contained in \mathcal{R} by means of the obvious inclusion $K : \mathcal{E}_{\mathcal{R}}^D \hookrightarrow \mathbf{E}(\mathcal{R})$.

For the latter case, it is often very convenient to assume that the free functor \mathbf{F}_K has been defined in such a way that $\eta_K(A)$ becomes the identity and therefore $\mathbf{U}_K(\mathbf{F}_K(A)) = A$ for all RWTs \mathcal{R} and K as above. The protection lemma ensures that this is possible without loss of generality.

Lemma 2.4.3 (Protection Lemma) Let \mathcal{R} be a RWT and A an $\mathcal{E}_{\mathcal{R}}^D$ -algebra, and consider the obvious inclusion $K : \mathcal{E}_{\mathcal{R}}^D \hookrightarrow \mathbf{E}(\mathcal{R})$. Then $\eta_K(A) : A \rightarrow \mathbf{U}_K(\mathbf{F}_K(A))$ is an isomorphism.

Proof Sketch. The result follows from the fact that the axioms in $\mathbf{E}(\mathcal{R})$ that are not already present in the subspecification $\mathcal{E}_{\mathcal{R}}^D$ satisfy the following conditions: (1) the signature does not introduce new elements in kinds contained in $\mathcal{E}_{\mathcal{R}}^D$, (2) the membership axioms do not introduce new elements in sorts contained in $\mathcal{E}_{\mathcal{R}}^D$, and (3) the equations do not identify elements in kinds contained in $\mathcal{E}_{\mathcal{R}}^D$. \square

Concrete instances of the model-theoretic semantics of rewriting logic will be used in Sections 3.1.2 and 3.2.4 for the particular forms of underlying METs that are relevant for that chapter.

Applications

As a formal specification language for labeled transition systems, more precisely enriched transition categories, rewriting logic can serve as a uniform *semantic framework* to represent a wide variety of programming and system models. This includes classes of finite and infinite automata, process algebras, Petri nets, and many other formalisms [Mes96, MOM96, Mes92], a particularly interesting one being a logical theory of concurrent object-oriented programming [Mes93]. In Chapters 3 and 5 of this thesis, we apply membership equational logic and the corresponding version of rewriting logic to the representation of programming languages, and we develop new representations of system models, mainly in the context of Petri nets, which were highly influential in the early developments of rewriting logic.

Another application of rewriting logic is its use as a first-order logical framework [MOM96, MOM94, Mes98, MMO95] in a spirit similar to Feferman's finitary inductive definitions of [Fef88]. From the viewpoint of general logics [MOM94, Mes89a], which study maps between logics, rewriting logic appears as a particularly interesting element in the category of logics, which can be used as a logical framework, because it is conceptually simple and important logics can be naturally mapped into it. This *general logics methodology*, that is based on general logics as a *logical metatheory* in combination with rewriting logic as a concrete *executable logical framework*, is employed in Chapter 6 of this thesis. Beyond its use as a logical framework, rewriting logic can serve as an *executable metalogic*, e.g. to specify the maps between logics, an application that we will study in the context of Chapter 7 of this thesis.

2.4.6 The Maude Rewriting Engine

The Maude paradigm recognizes the diversity of languages and formalisms as an inevitable consequence of the diversity of potential applications. In contrast to many conventional programming and specification languages, Maude is not assumed to be a closed language. Instead, it should be regarded as an open-ended family of interrelated languages that can be mapped into the core language, called *Core Maude*, in a conservative way. Core Maude is a language that has a simple formal foundation, namely rewriting logic together with its membership equational sublogic. Beyond this it avoids any assumptions about specific applications or specialized paradigms, and merely relies on the experience that a substantial number of paradigms, including functional programming, logic programming, and concurrent object-oriented programming, can be naturally represented in rewriting logic.

Core Maude can be seen as a framework language encapsulating the fundamental concepts of rewriting, that can be used to build multiparadigm languages, tools and environments. Core Maude is based on membership equational logic and the corresponding version of rewriting logic, but it has a number of additional features such as user-definable mixfix syntax, relaxed conditions for operator overloading, efficient support for subsort axioms, user-definable equational reduction strategies, several common built-in sorts and functions, and built-in support for reflection, which makes it a suitable practical basis for realizing the Maude paradigm. A good example of a language in the Maude family is *Full Maude*. Full Maude is a conservative extension of Core Maude in the sense above that provides two orthogonal features, namely: (1) a notion of object-oriented modules in the sense of [Mes93], and (2) a module-algebra in the style of OBJ3 with concepts such as renaming, parameterized theories and modules, and theory interpretations (views). Following the Maude paradigm, Full Maude has been specified in Core Maude itself using its reflective capabilities [Dur99].

The Maude rewriting engine can be used to execute specifications written in Core Maude. In particular, the Maude engine implements the operational semantics of MEL and RWL¹⁷, which is made accessible either externally via the user interface or internally, i.e. inside a rewrite specification, via the reflection interface. In order to provide the efficiency that is needed for complex applications, the Maude engine internally employs state-of-the-art matching algorithms and advanced compilation and optimization techniques. To this end, the specification must be (weakly) executable in the sense defined earlier, where as structural equations all combinations of associativity, commutativity, or left/right identity laws are admitted. Subsort axioms, which provide a semantically equivalent, but more efficient way to express membership axioms of a particular form, have to satisfy a regularity condition [BJM97, BJM00], which implies that each term has a statically computable least sort w.r.t. the partial order induced by these axioms. Furthermore, we have seen that the operational semantics of MEL and RWL does fix the particular strategies that are used for reduction and rewriting. Indeed, an additional feature of the Maude engine is that it allows the user to control the application of computational equations and rewrite rules by means of two kinds of strategies:

1. First of all, the user can specify *equational reduction strategies*, also called evaluation strategies in [CDE⁺99a]. These strategies can be specified for each operator separately and impose certain constraints on the way in which *equations* can be applied in the execution semantics described above. In general, reduction strategies can be used to increase efficiency, but also to eliminate nonterminating computations, e.g. to compute lazily with infinite data structures. The standard lazy and eager reduction strategies known from functional programming languages arise as special cases.
2. A more general form of strategies are *rewrite strategies*, that can be used to specify iterative rewriting, which involves the application of specific instances of *rewrite rules* and the selection of continuation points. Each rewrite strategy explores a transition system that is a subsystem of the labelled transition system given by all derivable rewrite judgements. In practice, rewrite strategies can range from a straightforward execution strategy, that applies the rules in some order with certain constraints to finally obtain a result, to a possibly exhaustive exploration strategy that explores multiple paths in the transition system. Technically, user-definable rewrite strategies in Maude are implemented using reflection, i.e., they are themselves executable specifications that use the reflection interface to operate on the object specification that has to be controlled.

¹⁷In its current version, however, Maude does not construct rewrite proof terms.

To conclude the overview of the Maude system we would like to mention a number of further applications showing that the efficient execution by rewriting is only one use of formal specifications. For instance, *interactive theorem proving* in the context of a MEL specification is supported by the inductive theorem proving tool developed by Clavel [Cla98]. This tool is another example of a reflective architecture, since the theorem prover operates at the metalevel of the object specification, i.e. the specification we wish to prove properties about. Other tools that exploit Maude's reflective capabilities are tools developed by Durán [Dur99] that can support the user to verify confluence, termination, and coherence of certain specifications in MEL and RWL. Finally, a reflective architecture is also used in the Maude prototype of the open calculus of constructions, which is presented in Chapter 8.

2.5 From Higher-Order Logics to Logical Type Theories

Type theories are an important ingredient not only for typed programming languages but also for typed logics as they are used in various interactive mathematical proof development systems. Although it is difficult to agree on a precise general notion of what a type theory is, it is correct to say that a common characteristic of type theories is that they classify certain syntactic objects according to types on the basis of typing rules, giving rise to an inductively-defined syntactic abstraction. In this thesis we use the notion of type theory usually in a more restricted sense to refer to different kinds of typed λ -calculi. Typed λ -calculi are extensions of the simply typed λ -calculus [Chu40]. We distinguish two forms of typed λ -calculi, namely those which use a language with type annotations (Church style) and those where types are assigned to untyped terms (Curry style). A unified presentation of several typed λ -calculi in the form of the so-called λ -cube can be found in [Bar92].

Often a type theory is used as part of a logic, and we speak of a *typed logic* in this case. For instance, algebraic specification languages such as many-sorted or order-sorted equational logics (cf. Section 2.4) are typed logics, with an underlying type theory of a very simple kind. Due to the absence of function types, such type theories could be called *algebraic type theories*, to distinguish them from *typed λ -calculi* as they are used for instance in proof development systems such as HOL [GM93b], PVS [ORS92, COR⁺95], IMPS [FGT93], or Isabelle [Pau90]. The underlying type theories of such higher-order logics usually introduce a type of formulae on which the logic is based. In the case of classical logics (all of the above except for Isabelle) the type of formulae is identified with the boolean data type. In spite of the use of the type system to distinguish between formulae and terms of other data types, we can observe a clear conceptual separation between the logic and the type theory in the logics above, in the sense that the inference rules of the logic are added on top of the type theory. A typed logic of this kind that we will discuss in this thesis in more detail is HOL. In fact, a detailed formulation of the HOL logic as an entailment system in the sense of Section 2.3 will be given in Chapter 7.

There is a second class of typed logics, implemented in proof development systems such as AUTOMATH [dB80], Nuprl [CAB⁺86], LEGO [Pol94, LP92], COQ [BBC⁺99], and ALF [Mag94], where the separation between the logic and the type theory disappears thanks to the propositions-as-types interpretation, an interpretation of the logic in the type theory, also known as formulae-as-types analogy [How80]. In this case, the inference rules of the logic become special instances of the typing rules. As an example, the calculus of constructions has an intuitionistic higher-order logic according to the propositions-as-types interpretation.

Since the logic is under this interpretation internal to the type theory we will also refer to the type theories which can be used in this spirit somewhat informally as *logical type theories* (such type theories will be used in Chapters 6, 7, and 8). The essential difference, when compared with the typed logics discussed above, is that the logic is conservative over the type theory rather than being obtained by an axiomatic extension. On the other hand, we will see that such an interpretation only provides an intuitionistic minimal logic, and that additional axioms are needed to make this logic classical. A closely related feature of logical type theories is that the propositions-as-types interpretation comes together with an abstract notion of proofs¹⁸ as objects, a fact that we use in this thesis not only to exchange proofs (cf. Appendix A), but also to reason about proofs inside the logic (cf. Chapter 4).

Historically, the first type theory in the broad sense of the term is the *theory of types* introduced by Whitehead and Russell in “*Principia Mathematica*” [WR13]. In their classical work the authors propose a typed logic to avoid the paradoxes that had been detected in the early formulation of set theory. The underlying type theory is based on a notion of set rather than the notion of function, which is the prevailing primitive concept in all modern type theories.

The first typed λ -calculus was developed by Church [Chu40] in 1940. This calculus is also called *simple type theory*, because the only nonatomic types are function types of the form $A \rightarrow B$ for types A and B . Although Church’s original motivation for developing such a theory was to use the λ -calculus as a logic, this goal was not achieved at that time. Instead, Church laid the foundations for typed functional programming languages, an application of type theory which turned out to be very fruitful, as witnessed by today’s sophisticated functional programming languages such as ML and Haskell.

A considerable step forward was the introduction of so-called *dependent function types* in the context of the *AUTOMATH* project [NGV⁺94] in 1968. A dependent function type, that we write as $(X : A)B$,¹⁹ denotes a set of functions that take an argument X of type A and yield a result of type $B(X)$, i.e. a type which can depend on the actual argument. Dependent function types generalize ordinary function types $A \rightarrow B$, where B does not depend on the argument. Furthermore, dependent types enable the direct use of the λ -calculus as a logic according to the *propositions-as-types interpretation* [How80], an issue that we will discuss in more detail in Section 2.5.2. The *AUTOMATH* project was the first project

¹⁸Depending on the type theory, the amount of information contained in such abstract proofs can vary. In the calculus of constructions, for instance, it is sufficient to make proof checking decidable, but in Nuprl it is the computational content of a proof, which is in general not sufficient for efficient proof checking.

¹⁹The type theory AUT-QE unifies λ -abstraction $[X : A]M$ and Π -abstraction $(X : A)B$, an interesting aspect which leads to subtle differences between this type theory and more recent type theories with dependent types.

that developed a proof assistant used to formalize large portions of informal mathematics such as Landau’s “Grundlagen der Analysis”. Another interesting comment concerning the use of AUTOMATH type theories is that they do not favor or enforce a strictly conservative discipline for AUTOMATH developments. In fact, nonconservative extensions of the type theory, e.g. by classical axioms or axiomatically defined data types, have been liberally used. A similarly liberal standpoint is part of the pragmatics of the open calculus of constructions that we present in Chapter 8.

In the subsequent years a number of very influencing type theories have been developed by Martin-Löf. The first system he put forward in 1971 was equipped with a type of all types, which soon turned out to be logically inconsistent,²⁰ as witnessed by Girard’s paradox [Gir72, Coq86]). Not much later, Martin-Löf developed his *intuitionistic type theory* [ML74] which underwent a number of changes in subsequent years. Except for the first inconsistent system, Martin-Löf’s type theories are predicative. They have a rich variety of types, such as enumeration types, natural numbers, products, sums, dependent function types, and dependent sum types. A major step was the extension of the type theory by an infinite predicative hierarchy of universes $\mathbb{U}_0, \mathbb{U}_1, \mathbb{U}_2, \dots$ in [ML82]. Universes are types of types and hence allow us to view types as elements, i.e. as first-class citizens. *Predicativity* means that the universes are stratified and not generally closed under formation of dependent function types. More precisely, if A and B are types then $(X : A)B$ resides in a universe at least as high as those of A and B . The clear hierarchical structure of predicative theories is intended to avoid paradoxes as the one in Martin-Löf’s first system. In connection with (dependent) function types universes allow us to represent type operators, i.e. functions that operate on types, and explicitly polymorphic functions, i.e. functions that have explicit type parameters, although the latter feature is not needed in Martin-Löf’s (implicitly) polymorphic type theories, where λ -abstractions do not carry types. On top of the rich collection of types the logic is defined, i.e. introduced conservatively, by means of a propositions-as-types interpretation. In particular, conjunction and disjunction are interpreted as product and sum, respectively, and universal and existential quantifications are interpreted as dependent functions and dependent sums, respectively. With the exception of the decidable monomorphic type theory in [ML74] and a monomorphic system described in [PSN90], which is also known as Martin-Löf’s logical framework²¹ (put forward in 1986), Martin-Löf’s type theories are polymorphic, that is different types can be associated with the same term. Martin-Löf considered extensional and intensional polymorphic type theories, differing in the rules for equality judgements (both versions are pre-

²⁰In spite of its failure to serve as a logic, it has been found that a type theory with a type of all types can very well serve as an expressive programming language [Car86].

²¹A variant of Martin-Löf’s logical framework has been implemented in the proof assistant Alfa [CC99b] and in the programming language Cayenne [Aug99, Aug].

sented in [PSN90]). In short, extensional type theories allow us to use arbitrary logical equalities to prove equality and typing judgements, whereas in intensional type theories equality judgements capture a more limited intensional notion of computational equality. The semantics of Martin-Löf's polymorphic type theories can be given by an untyped programming language, where type constructors are not essentially different from other constructors, and computation rules are given in the style of structured operational semantics. A particularly interesting feature is that the type theory is open-ended [Tsu01] in the sense that the derivable judgements remain valid under quite general extensions of the underlying programming language by new base types and new computation rules. Open-endedness has been investigated in [How91, How89] and is also the basis for the Nuprl type theory [All87], which further advances Martin-Löf's ideas.

Nuprl [CAB⁺86] extends Martin-Löf's polymorphic extensional type theory in various directions. It consequently realizes Martin-Löf's idea that a type theory should be open-ended, in the sense that type constructors, inference rules, and computation rules can be added when this extension can be justified its semantics, and hence it is more appropriate to view Nuprl as an evolving family of type theories. The collection of Nuprl types includes subset types, quotient types, inductive types, and intersection types. Nuprl furthermore supports contravariant subtyping, which is justified by its consequent characterization of functions by their computational behavior. Among other extensions, partial types [CS87] and entirely new concepts such as very dependent function types [Hic96] have been studied. In addition, Nuprl realizes the idea of direct (i.e. untyped) computation rules that are admitted by Martin-Löf's semantics, but are not part of any of his type theories. The constructive semantics of Nuprl differs from Martin-Löf's semantics in the meaning of sequents, and it is also more complicated due to the admission of quotient types. A detailed description can be found in [All87]. As in Martin-Löf's polymorphic extensional type theory, type checking and type inference in the Nuprl type theory are undecidable. Hence, typing judgments are witnessed by external Nuprl proofs, which are derivations involving the inference rules of Nuprl. The Nuprl proof development system, that supports goal-oriented, tactic-based, interactive theorem proving, has proven its applicability in a considerable number of complex developments ranging from algebra and automata theory to protocol verification and optimization [Jac94, CNJU00, LvRBK01]. As a consequence of the propositions-as-types interpretation, Nuprl has an abstract notion of proofs-as-objects, which represents the computational contents of proofs inside the type theory and is therefore referred to as proofs-as-programs interpretation. Program extraction, i.e. the extraction of programs from (external) proofs, is supported by Nuprl on the basis of this interpretation.

An interesting variant of Nuprl is the classical version proposed by Howe [How96a, How98a] in the context of the HOL/Nuprl connection. By giving a hybrid computational/set-theoretic semantics he shows how classical reasoning becomes pos-

sible with only a few modifications to the Nuprl type theory. Since classical reasoning is an important mode that should be supported by a generally applicable type theory, we will extract and present the essence of the HOL/Nuprl connection in Chapter 7, where we also give a precise definition of the relevant entailment systems (in the sense of Section 2.3).

Quite different from Martin-Löf’s approach, Girard and Reynolds developed independently in the early 70’s the *second order λ -calculus*, also called *system F* [Gir72, GLT89, Rey74]. Girard’s motivation was of a proof-theoretic nature, whereas Reynolds aimed at applications in the context of programming languages. System F can be seen as Church’s simple type theory extended by the explicit polymorphism, i.e. by the possibility of defining functions with type parameters. Girard has developed another system, called *higher-order λ -calculus* or *system $F\omega$* [Gir72], that extends system F by type operators, i.e. operators that have types as arguments and produce types. In contrast to Martin-Löf’s type theory and Nuprl, Girard’s systems are impredicative and hence less restrictive concerning the formation of function types. This remark also holds for the calculus of constructions which contains F and $F\omega$ as subsystems.

In 1984 Coquand and Huet developed the *calculus of constructions (CC)*, that integrates dependent function types, explicit polymorphism, and type operators in a surprisingly simple and uniform way [CH85, Coq85, CH88]. In contrast to Martin-Löf’s approaches, CC is an impredicative type theory and the only nonatomic type is the dependent function type, written as $(X : A)B$. *Impredicativity* refers to the single impredicate universe **Prop** and means that **Prop** is closed under dependent function types over arbitrary types. In other words, if B is in **Prop** and A is any type then $(X : A)B$ exists and is still in **Prop**. CC has an impredicative higher-order logic by virtue of the propositions-as-types interpretation. As a notion of computation CC employs β -reduction. Among several extensions of CC that have been proposed in the literature (see below), an extension of CC by η -conversion has been studied in [Geu93]. Last but not least, it is worth mentioning that CC (and also the extensions mentioned below) enjoy metatheoretical properties such as type uniqueness, subject reduction, and strong normalization. Furthermore, type checking is decidable and efficient type checking and type inference algorithms exist [Hue89].

The *logical framework LF* [HHP87] is a theory of dependent types that generalizes Church’s simple type theory. It is similar to the system λP , a predicative subsystem of CC according to Barendregt’s λ -cube, except for the fact that in LF type conversion includes not only β -conversion by also η -conversion. As explained in [HHP87], LF is well-suited as a higher-order logical framework that is capable of encoding a variety of standard logics and type theories using the so-called *judgements-as-types* interpretation. We refer to LF as a *higher-order logical framework*, because it is based on the λ -calculus, in contrast to a first-order framework such as membership equational logic or rewriting logic. On

the other hand, LF is often called a *first-order λ -calculus*, because of its rather modest notion of dependent function types, which allow quantification over elements, but not over types as in the second-order or higher-order λ -calculus above. The approach to encode terms with bound variables in LF is based on *higher-order abstract syntax* [PE88], i.e., binding constructs of the object language are represented using λ -abstractions of LF. A good introduction to LF-style logical frameworks and these representation techniques can be found in [Pfe96]. LF has been implemented in the constraint logic programming language ELF [Pfe91] and its successor Twelf [PS99]. Both systems provide the infrastructure to represent and implement other logics. Twelf additionally supports metalogical theorem proving by virtue of a first-order metalogic that has been built on top of LF.

In 1986 Coquand extended CC by a cumulative hierarchy of universes [Coq86] $\mathbf{Type}_0, \mathbf{Type}_1, \mathbf{Type}_2, \dots$, as it has been used in Martin-Löf's type theory. The resulting system is called the *generalized calculus of constructions (GCC)*. As in the case of Martin-Löf's type theory, the universes \mathbf{Type}_i are predicative, but impredicativity of the bottom universe \mathbf{Prop} is maintained. Already CC admits a classical model where \mathbf{Prop} is interpreted as a two-element set. However, this classical model is not very useful in CC, since it makes the only proper universe available semantically degenerated and thus does not leave any space for interesting data types. In the context of GCC, however, interpreting \mathbf{Prop} classically can lead to a useful model, because this does not exclude a rich semantics for data types that reside in the new predicative universes \mathbf{Type}_i .

Motivated by theory abstraction as an application, Luo added strong dependent sum types, also called strong Σ -types, to GCC. The resulting type theory, the *extended calculus of constructions (ECC)* is presented in [Luo91] and [Luo94]. Another subtle but important improvement is that ECC extends the subtyping relation of GCC to a slightly richer subtyping relation to ensure the existence of principal types, which simplifies type checking considerably. This completion is also referred to as full cumulativity as opposed to the weaker cumulativity of the Martin-Löf-style universe hierarchy used in GCC. Later ECC was extended by a scheme for inductive definitions to the *unified theory of dependent types (UTT)* [Luo94].²² Since the early 90's the LEGO proof assistant [LP92] developed by Pollack provides an implementation of ECC. It has also been extended by the UTT scheme for inductive definitions and in addition by a scheme for coinductive definitions in the meantime. Thanks to the underlying type theory and the use of the propositions-as-types interpretation, the complexity of LEGO is quite moderate. Interactive theorem proving reduces to functional programming and can be conducted by goal-oriented refinement. On the other hand, the LEGO system is not equipped with powerful tactics for proof automation as we find them

²²Strictly speaking, UTT is not exactly an extension of ECC, because it is specified in Martin-Löf's logical framework, which is predicative so that coercions have to be used as a replacement for universe subtyping.

in HOL, PVS, or COQ. Nevertheless, a number of substantial developments, such as a normalization proof for system F [Alt93], have been conducted using the LEGO system. In Appendix A we use an example from the LEGO distribution to illustrate the use of our executable specification of pure type systems.

In parallel with the development of ECC and UTT, the development of another extension of GCC was initiated by Coquand and Paulin-Mohring in [CPM90]. It is called the *calculus of inductive constructions (CIC)* [BBC⁺99]. Generalizing the universe hierarchy of ECC, which has a single impredicative universe, it is equipped with two impredicative universes **Prop** and **Set** and a single predicative universe hierarchy on top of these.²³ Furthermore, it has inductive data types and a restricted form of terminating recursion. Its most recent version supports also the coinductive counterparts of these concepts [Gim94]. Special attention has been devoted to the issue of extracting programs from type-theoretic proof objects, a feature which makes the precise rules for inductive definitions rather subtle. The COQ system [BBC⁺99] that implements CIC is a tactic-based interactive proof assistant that supports a goal-oriented proof style. It has many additional features such as user-definable syntax and coercions that are very useful in practice. Its overall complexity is beyond that of LEGO, but it shares with LEGO the idea that the soundness of the system is based on a relatively small core that is essentially a type checker. COQ has been employed for many complex developments using intuitionistic or classical logic.

Since this thesis aims at a unified language based on equational logic, rewriting logic, and a logical type theory, namely the (extended) calculus of constructions, we begin with an introduction to the latter. Since inductive reasoning is an important use of the unified formalism, the open calculus of constructions presented in Chapter 8, we also introduce CIC, the most commonly used extension of CC by inductive types that is the basis of the COQ system. In fact, this type theory is particularly well-suited for inductive reasoning, which makes it especially useful as a metalogic to reason about inductively presented formal systems, an application that we explore by the formalization of the UNITY-style temporal logic in Chapter 4. In that application we use CIC not only as a metalogic to reason about the temporal logic, but also as a functional programming language and as a logic for high-level specification and verification of programs and systems.

2.5.1 The Calculus of Constructions

The *calculus of constructions (CC)* [CH85, Coq85, CH88, Coq90a] extends the simply typed λ -calculus $\lambda \rightarrow$ [Chu40] by a general form of dependent function types, that generalize explicit polymorphism, type operators, and dependent

²³An earlier version of CIC was equipped with two separate universe hierarchies with **Prop** and **Set** at the bottom, respectively.

types. The systems F [Gir72, Rey74], $F\omega$ [Gir72] and a system λP close to the logical framework LF [HHP87] are subsumed by CC. CC is an impredicative type theory in the sense that its type universe **Prop** is closed under formation of (dependent) function types. Based on β -reduction as a notion of computation, CC is strongly normalizing, and as a consequence all representable functions are total and computable. Furthermore, type checking is decidable and an algorithm for type inference exists. From a logical point of view, CC admits the interpretation of an intuitionistic higher-order logic according to the propositions-as-types interpretation which is discussed in the following section.

In what follows, we introduce the language of CC and its formal system. As usual in expressive type theories, terms and types are not a priori distinguished. *Terms* are defined by the following syntax, using M, N, A, B to range over terms and X to range over names:

$$X \mid MN \mid [X : A]M \mid (X : A)B \mid \mathbf{Prop} \mid \mathbf{Type}$$

Here, X denotes a variable, (MN) denotes a function application, $[X : A]M$ denotes a λ -abstraction, $(X : A)B$ denotes a dependent function type, **Prop** denotes an impredicative type universe, and **Type** denotes an auxiliary type universe. We should add that in $[X : A]M$ and $(X : A)B$ the variable X is bound in M and B , respectively, and we assume that α -equivalent terms, i.e., terms that are equal up to consistent renaming of their bound variables, are identified. We furthermore use the standard convention that a nondependent function type $A \rightarrow B$ abbreviates $(X : A)B$ where X does not occur in B .

The formal system of CC defined as follows (cf. [Bar92]). The sentences are *typing judgements* of the form $\Gamma \vdash M : A$, meaning that Γ is a well-typed context and M is of type A in Γ . Here a *context* is a list of declarations of the form $X : A$, which declares X to be of type A . We write $X \in \Gamma$ to express that X is declared in Γ . We write $[X := N]M$ to denote the standard (capture-free) substitution of all free occurrences of X in M by N . The variables s, s_1, s_2 range over the universes $\{\mathbf{Prop}, \mathbf{Type}\}$. The formal system of CC has the following inference rules:

$$\frac{}{\boxed{\phantom{\Gamma \vdash \text{Prop} : \text{Type}}} \vdash \mathbf{Prop} : \mathbf{Type}} \quad (\text{Ax})$$

$$\frac{\Gamma \vdash A : s}{\Gamma, X : A \vdash X : A} \quad X \notin \Gamma \quad (\text{Start})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, X : B \vdash M : A} \quad X \notin \Gamma \quad (\text{Weak})$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash B : s_2}{\Gamma \vdash (X : A)B : s_2} \quad (\text{Pi})$$

$$\frac{\Gamma, X : A \vdash M : B \quad \Gamma \vdash (X : A)B : s}{\Gamma \vdash [X : A]M : (X : A)B} \quad (\text{Lda})$$

$$\frac{\Gamma \vdash M : (X : A)B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : [X:=A]B} \quad (\text{App})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad A \equiv_{\beta} B \quad (\text{Conv})$$

In the last rule, \equiv_{β} denotes β -conversion, i.e. the equivalence induced by β -reduction \rightarrow_{β} which is the context closure of \Rightarrow_{β} defined by

$$([X : M] N) \Rightarrow_{\beta} [X:=N]M.$$

CC uses β -reduction as the notion of computation and β -conversion as the notion of type equality, which is intensional, computational and decidable for well-typed terms. Although \rightarrow_{β} is defined on all terms, well-typed or not, this more a matter of convenience, since it is in fact intended as a typed notion of computation, i.e. it is only used on well-typed terms.

It is important to note that **Type** has only the status of an auxiliary universe in CC, which enables us to form certain dependent types such as $(X : \mathbf{Prop})\mathbf{Prop}$ or $(X : A)\mathbf{Prop}$ in a well-typed context with $A : \mathbf{Prop}$, but on the other hand well-typed contexts cannot contain declarations of the form $X : \mathbf{Type}$, as we can see from the rules **Start** and **Weak**. For this reason, **Type** is not a full-featured universe and not made explicit in equivalent (earlier) presentations of CC such as the one in [Coq90a].

Formally, the type inference rules given above constitute an inductive definition of a (unary) entailment system in the sense of Section 2.3. We will use such entailment systems to state some metatheoretic results in the more general context of pure type systems which subsume CC as a special case in Chapter 6.

2.5.2 Propositions as Types and Proofs as Objects

Under the *propositions-as-types interpretation* [How80] (see also [Coq90b]), also called formulae-as-types analogy, a logical formula A is interpreted as a type $\llbracket A \rrbracket$ such that A is provable iff $\llbracket A \rrbracket$ is inhabited. In addition, a proof P is interpreted as an object $\llbracket P \rrbracket$ such that $\llbracket P \rrbracket$ is an element of $\llbracket A \rrbracket$. This second part is called *proofs-as-objects interpretation* or *proofs-as-programs interpretation*. These interpretations of a logic in a type theory have originally been used in the context of the Curry-Howard isomorphism [How80], which requires an even stronger correspondence, namely that normalization of proofs in the logic corresponds to normalization in the type theory. Curry-Howard isomorphisms have been successfully established in the setting of intuitionistic logics and corresponding type

theories, such as between a higher-order intuitionistic logic and the calculus of constructions [Geu93].

The propositions-as-types interpretation can be seen as a type-theoretic concretization of the *BHK interpretation* [Hey71], that has been used by Brouwer, Heyting, and Kolmogorov to explain the meaning of intuitionistic logic and can be summarized as follows:

1. A proof of a conjunction $A \wedge B$ is a pair containing a proof of A and a proof of B ,
2. A proof of a disjunction $A \vee B$ is a pair containing either a proof of A or a proof of B together with information about which of these choices is taken,
3. A proof of an implication $A \Rightarrow B$ is a constructive method providing a proof B given a proof of A .
4. A proof of a universal quantification $\forall x \in A . B$ is a constructive method providing a proof of $B(a)$ for each a in A .
5. A proof of an existential quantification $\exists x \in A . B$ is a pair containing an element $a \in A$ and a proof of $B(a)$.

What counts as a constructive method is not fixed in the BHK interpretation. The degree of constructivity depends on what we are willing to accept as a proof. An extreme position would be to require that the method should be a computable function, possibly within certain complexity bounds. On the other extreme we have the classical position, where we accept as methods all set-theoretic functions. In the context of typed λ -calculi that formalize a notion of function, the most obvious choice is to identify such methods with the functions of the calculus. In type theories which admit a rich spectrum of semantics, ranging from constructive to classical, we do not have to commit ourselves to any of the positions mentioned above.

In a type theory with function types an implication $A \Rightarrow B$ can be interpreted as a type $\llbracket A \rrbracket \rightarrow \llbracket B \rrbracket$. In fact, this interpretation establishes a Curry-Howard isomorphism between simple type theory and minimal propositional logic. To deal with logics with quantifiers we need more expressive types. For instance, in type theories with dependent function types, a universal quantification $\forall X \in A . B$ can be interpreted as $(X : \llbracket A \rrbracket) \llbracket B \rrbracket$. Since dependent function types generalize ordinary function types, the former are sufficient to establish a propositions-as-types interpretation for a logic based on implication and universal quantification.

In the context of intuitionistic impredicative higher-order logic, the implicative-universal fragment is sufficient to define all the remaining logical operators and

equality. This technique, which is also called *impredicative encoding*, is for instance used in the LEGO system to introduce the logic on top of the type theory. Alternatively, and this is the choice adopted in COQ and Nuprl, the remaining logical operators can be defined in terms of suitable (inductive) data types following the BHK-interpretation (see Section 2.5.5).

A conceptual benefit of the propositions-as-types and proofs-as-objects interpretation is that the activities of typed functional programming and theorem proving that appear to be unrelated at the first sight become identified in a strict formal sense: Proving a theorem A is the task of writing a program which constructs a proof, i.e. an element of A , viewing A as a type. Interactive proof assistants can exploit the propositions-as-types interpretation and implement goal-oriented theorem proving as program construction by refinement. The task of verifying that a theorem has a given proof, i.e. proof-checking, is reduced to type checking. If efficient algorithms for type checking exist, internal proof objects can completely replace the external notion of proof, which further simplifies the implementation of proof development systems based on such type theories.

In summary, the propositions-as-types interpretation reveals an analogy between logics and type systems which can be exploited to simplify the design of logics and proof development systems considerably. In our view, the economy of concepts makes type theories with dependent types a very natural formal basis for proof development systems that implement higher-order logics.

2.5.3 The Generalized Calculus of Constructions

Similar to von Neumann universes in set theory, the calculus of constructions can be equipped with an infinite cumulative universe hierarchy, as in the *generalized calculus of constructions (GCC)*, which maintains the impredicative universe \mathbf{Prop} and adds a hierarchy $\{\mathbf{Type}_i \mid i \in \mathbb{N}\}$ of predicative universes. The universe structure of GCC is characterized by a *universe containment relation* \in which is defined by

$$\mathbf{Prop} \in \mathbf{Type}_0 \in \mathbf{Type}_1 \in \mathbf{Type}_2 \dots$$

and induces a *subtyping order* \leq on universes, namely

$$\mathbf{Prop} \leq \mathbf{Type}_0 \leq \mathbf{Type}_1 \leq \mathbf{Type}_2 \dots$$

The following two inference rules express universe containment (\mathbf{Ax}) and cumulativity (\mathbf{Cum}), respectively.

$$\frac{}{\Box \vdash s_1 : s_2} \quad \text{if } s_1 \in s_2 \quad (\mathbf{Ax})$$

$$\frac{\Gamma \vdash A : s_1}{\Gamma \vdash A : s_2} \quad \text{if } s_1 \leq s_2 \quad (\mathbf{Cum})$$

The inference rules of GCC are those of CC together with the previous two rules, but with the Pi rule replaced by the following two, which distinguish dependent type formation in the *impredicative universe* **Prop** from dependent type formation in the *predicative universes* \mathbf{Type}_i . The key difference is that **Prop** is closed under formation of dependent types, but \mathbf{Type}_i does not enjoy this property.

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash B : s_2}{\Gamma \vdash (X : A)B : \mathbf{Prop}} \quad \text{if } s_2 = \mathbf{Prop} \quad (\text{ImpredPi})$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash B : s_2}{\Gamma \vdash (X : A)B : \max(s_1, s_2)} \quad \text{if } s_2 = \mathbf{Type}_i \quad (\text{PredPi})$$

In contrast to CC, which has only a single universe **Prop** apart from the auxiliary universe **Type**, the richer universe structure allows us to separate logical objects residing in **Prop** from data types which reside in the other universes. An important benefit arising from this separation is not only conceptual clarity but also the existence of reasonable classical models where **Prop** is interpreted as a two element set $\{\{\emptyset\}, \emptyset\}$ containing a nonempty set (the interpretation of **True**) and an empty set (the interpretation of **False**). This semantics is classified as a *proof-irrelevance semantics*, because it abstracts from the differences between proofs of a propositions. Obviously, the semantics also justifies the *law of the excluded middle*, and furthermore *logical extensionality*, i.e. two propositions are equal iff they are logically equivalent. Formally, such classical models are also possible in CC (see e.g. [Coq90a]), but they exclude the existence of nontrivial data types, since in CC data types that have to be introduced in the context can only live in **Prop**.

2.5.4 The Calculus of Inductive Constructions

The *calculus of inductive constructions* (CIC) [BBC⁺99, PM93] further generalizes GCC. It has been implemented in the COQ system which is briefly introduced in the subsequent section. CIC replaces the single impredicative universe **Prop** by two independent impredicative universes, called **Prop** and **Set**, such that $\mathbf{Prop}, \mathbf{Set} \in \mathbf{Type}_0$. As ECC, CIC has a fully cumulative universe hierarchy, i.e. the subtyping relation is slightly richer than that of GCC. For details concerning this issue we refer to [Luo94] and [BBC⁺99]. More importantly, CIC adds inductive definitions, constructs for terminating fixpoint definitions and case analysis and their coinductive counterparts. The universe **Set** is intended as a universe of (computational) data types, such as booleans **bool** and natural numbers **nat**, which are given by their usual inductive definitions. In spite of these quite powerful extensions, CIC maintains metatheoretic properties such as uniqueness of principle types, subject reduction, strong normalization, and decidability of type

checking. As GCC, CIC also admits a classical semantics [Wer97], and in fact it is this semantics under which we use CIC in this thesis.

The main feature of CIC that we exploit in this thesis are inductive type definitions. For such an inductive definition the user has to specify only the constructors for the data type to be defined and their types. Functions on inductive datatypes can then be defined using a syntactically restricted combination of case analysis and fixpoint operators. As an example consider the following definition of the inductive type `bool` of booleans with constructors `true` and `false`, which can be written in the syntax of COQ as:

```
Inductive bool : Set := true : bool
      | false : bool.
```

Given such an inductive definition, it is possible to define the standard elimination principles, which allow us to define (possibly dependent) functions on inductive types by higher-order primitive recursion. To be more precise, we can define a separate elimination principle for each universe, i.e. `Prop`, `Set` and `Typei`:

```
bool_ind = [P:(bool->Prop); t:(P true); f:(P false); b:bool]
  Cases b of true => t | false => f end
  : (P:(bool->Prop))(P true)->(P false)->(b:bool)(P b)
```

```
bool_rec = [P:(bool->Set); t:(P true); f:(P false); b:bool]
  Cases b of true => t | false => f end
  : (P:(bool->Set))(P true)->(P false)->(b:bool)(P b)
```

```
bool_rect = [P:(bool->Type); t:(P true); f:(P false); b:bool]
  Cases b of true => t | false => f end
  : (P:(bool->Type))(P true)->(P false)->(b:bool)(P b)
```

Notice that in the syntax of COQ a predicative universe `Typei` is just written as `Type` leaving the universe level implicit, a feature called *typical ambiguity*, so that the last definition introduces actually an infinite family of elimination principles.

Similarly, consider an inductive definition of the type `nat` of natural numbers with constructors `0` and `S`:

```
Inductive nat : Set := 0 : nat
      | S : nat->nat.
```

Again we can define the standard elimination principles as follows:

```

nat_ind = [P:(nat->Prop); z:(P 0); f:((n:nat)(P n)->(P (S n)))]
  Fix F {F [n:nat] : (P n) :=
    Cases n of 0 => z | (S m) => (f m (F m)) end}
  : (P:(nat->Prop))(P 0)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

```

```

nat_rec = [P:(nat->Set); z:(P 0); f:((n:nat)(P n)->(P (S n)))]
  Fix F {F [n:nat] : (P n) :=
    Cases n of 0 => z | (S m) => (f m (F m)) end}
  : (P:(nat->Set))(P 0)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

```

```

nat_rect = [P:(nat->Type); z:(P 0); f:((n:nat)(P n)->(P (S n)))]
  Fix F {F [n:nat] : (P n) :=
    Cases n of 0 => z | (S m) => (f m (F m)) end}
  : (P:(nat->Type))(P 0)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

```

The COQ system automatically generates such elimination principles whenever possible. It is noteworthy that the elimination principles are introduced by purely definitional means, since CIC has a built-in notion of inductive types. A different approach is taken in systems such as UTT, as implemented in LEGO. UTT provides inductive type schemes for the underlying type theory ECC. Such schemes describe how for each inductive type elimination principles are introduced as new primitives together with suitable computation rules, giving rise to a nonconservative extension of the underlying type theory. In other words, UTT can be seen as a metatheory that describes admissible extensions of ECC.²⁴

Exploiting the propositions-as-types interpretation for types in **Prop**, the elimination principle allows us to define functions on inductive types which generate proofs. In other words, the elimination principles for **Prop** correspond to *induction principles*, a fact that can be easily seen from their types under the logical reading. The elimination principles for **Set** and **Type** on the other hand, are also called *recursion principles*, since they are primarily used in connection with data types, if we follow the conventions of CIC explained above. In other words, the propositions-as-types interpretation conceptually unifies the notions of induction and (primitive) recursion, using the single concept of elimination principles. The inductive data types of CIC are sufficiently powerful to admit the definition of inductive type families and simultaneous inductive definitions of several types. Furthermore, inductive definitions of sets/relations, which can be represented as predicates, i.e. **Prop**-valued functions, arise as a special case of inductive type families. In summary, we can again see how the propositions-as-types interpretation, here in connection with inductive types, leads to potential simplifications in the design of logics and proof development systems.

²⁴To be more precise a slight generalization of ECC is needed, which allows extensions of the computational system by certain forms of computation rules, as it is implemented in LEGO.

The central motivation for the separation between `Prop` and `Set` in CIC is to support extraction of programs from proofs in such a way that only the computationally meaningful parts of the proof, i.e. elements of types in `Set`, are extracted. To maintain the separation between uninformative (logical) and informative (computational) objects the rules for inductive definitions have to distinguish carefully between types from `Prop` and types from `Set`. For more information on inductive types in CIC, in particular for a number of restrictions that are enforced to ensure logical consistency and allow the interpretation of elements of types in `Prop` as uninformative objects, we refer to [BBC⁺99, PM93].

2.5.5 The COQ Proof Development System

The proof assistant COQ [BBC⁺99] supports goal-oriented interactive theorem proving and program construction by refinement in CIC. Its core, a successor of the *constructive engine* [Hue89], is a type inference and type checking algorithm that is also used for proof checking under the propositions-as-types and proofs-as-objects interpretation. COQ can be used as an interactive system with partial automation in the sense that proofs are generated by tactics that are invoked by the user or executed as part of a proof script. Tactics are written in the functional metalanguage CAML, the implementation language of the COQ system. What makes the COQ approach quite different from the LCF approach inherited by systems such as HOL and Nuprl, is that after completion of a proof the tactics have produced an explicit proof object, that is not accepted by the system until it passes the type checker. This ensures that the soundness of the system depends only on the type checker, which is of relatively small complexity compared with the whole system.

The COQ proof assistant does not only implement CIC but also provides a number of additional features such as typical ambiguity to avoid the specification of explicit universe levels, automatic completion of terms involving holes (denoted by `?`), extensible grammars for user-definable syntax, implicit coercions guided by a coercion hierarchy, program extraction from proofs, user-friendly notation for recursive definitions, user-definable tactics in addition to the tactics that can be written in the metalanguage CAML, and generation of natural language from proofs. Furthermore, COQ has a rich mathematical library that provides a good starting point for formal developments.

The standard logical libraries of COQ are based on the following core definitions, which establish a propositions-as-types interpretation on the basis of suitable inductive types.

```
Inductive True : Prop :=
  I : True.
```

```
Inductive False : Prop := .
```

```
Definition imp :=
  [A,B:Prop] A -> B.
```

```
Definition not :=
  [A:Prop] A -> False.
```

```
Inductive and [A,B:Prop] : Prop :=
  conj : A -> B -> (and A B).
```

```
Inductive or [A,B:Prop] : Prop :=
  or_introl : A -> (or A B)
| or_intror : B -> (or A B).
```

```
Definition iff :=
  [P,Q:Prop] (and (P -> Q) (Q -> P)).
```

```
Definition all :=
  [A:Set] [P:A->Prop] (x:A) (P x).
```

```
Definition allT :=
  [A:Type] [P:A->Prop] (x:A) (P x).
```

```
Inductive ex [A:Set;P:A->Prop] : Prop :=
  ex_intro : (x:A) (P x)->(ex A P).
```

```
Inductive exT [A:Type;P:A->Prop] : Prop :=
  exT_intro : (x:A) (P x)->(exT A P).
```

In addition, we have a polymorphic version of Leibnitz equality `eq` (and `eqT`) which is introduced inductively, defining `(eq A x)` as a unary predicate which is only satisfied for `x`.

```
Inductive eq [A:Set;x:A] : A -> Prop :=
  refl_equal : (eq A x x).
```

```
Inductive eqT [A:Type;x:A] : A -> Prop :=
  refl_eqT : (eqT A x x).
```

Finally, the COQ library introduces a more readable syntax for logical operators such that `(not A)`, `(and A B)`, `(or A B)`, `(all ? P)`, `(allT ? P)`, `(ex ? P)`,

$(\text{exT } ? P)$, $(\text{eq } ? M N)$, and $(\text{eqT } ? M N)$ can be written as $\sim A$, $A \wedge B$, $A \vee B$, $(\text{All } P)$, $(\text{AllT } P)$, $(\text{Ex } P)$, $(\text{ExT } P)$, $M = N$, and $M == N$, respectively.

The theoretical aspects of CIC, mainly its capabilities to reason about inductively defined objects and its natural capability of treating proofs as objects, but also the maturity of the COQ system together with its rich collection of features and extensions, and its mathematical libraries have motivated the choice of COQ as a framework for the development of the temporal logic library presented in Chapter 4. Our experience with COQ, and with the related systems LEGO and Nuprl, also had a considerable impact on the design and implementation of the open calculus of constructions that is subject of Chapter 8.

Chapter 3

Rewriting Logic as a Semantic Framework:

Representing High-Level Petri Nets

This chapter studies the use of rewriting logic as a semantic framework. Simultaneously, it attempts to contribute to the general goal of unifying Petri net models by studying in detail the unification of key models within rewriting logic [Mes92]. Specifically, we show how place/transition nets, nets with test arcs, algebraic net specifications, and colored Petri nets can all be naturally represented within rewriting logic. Our work extends in substantial ways previous work on the rewriting logic representation of place/transition nets [Mes92], nets with test arcs [Mes96], and algebraic net specifications [Stec].

The representations in question associate a rewrite specification to each net in a given class of Petri net models in such a way that concurrent computations in the original net naturally coincide with concurrent computations in the associated rewrite specification. That is, we exhibit appropriate bijections between Petri net computations and rewriting logic computations, viewed as equivalence classes of proofs, that is, as elements of the free model associated to the corresponding rewrite specification [Mes92].

Furthermore, for certain classes of nets, namely place/transition nets and a general form of algebraic net specifications, which subsume the well-known class of colored Petri nets, we show that the representation maps into rewriting logic are *functorial*; that is, that they map in a functorial way net morphisms to rewrite specification morphisms. In addition, such functorial representations can be further extended to the level of *semantic models*, yielding *semantic equivalence theorems* (in the form of natural isomorphisms of functors) between well-known semantic models for the given class of Petri nets and the free models of the corresponding rewrite theories or, more precisely, models obtained from such free models by forgetting some structure.

As we further explain in the body of the chapter, this work, including the above-mentioned functorial semantics and the semantic equivalences, generalizes in some ways, and complements in others, a substantial body of work initiated by José Meseguer and Ugo Montanari under the motto “Petri nets are monoids” [MM90, MOM91a, MOM91b, MMS96, DMM96, MMS92, MMS94, MMS97, BMMS98, BMMS99], in which categorical models are naturally associated as semantic models to Petri nets, and are shown to be equivalent to well-known “true concurrency” models. Our work is also related to linear logic representations of Petri nets [MOM91a, MOM91b, Asp87, BG90, Bro89, EW90]. All this is not surprising, since, as explained in [Mes92], both the categorical place/transition net models of [MM90] and the linear logic representations of place/transition nets inspired rewriting logic as a generalization of both formalisms. But, as shown in this chapter, the extra algebraic expressiveness of rewriting logic is very useful to model in a simple and natural way not only place/transition nets, but also *high-level nets*, such as algebraic net specifications and colored Petri nets.

Our proposed unification of Petri net models is not only of conceptual interest.

Given that, under reasonable assumptions, rewrite theories can be executed, the representation maps that we propose provide a uniform operational semantics in terms of efficient logical deduction. Furthermore, using a rewriting logic language implementation such as Maude [CDE⁺99a, CDE⁺00b], it is possible to use the results of this chapter to create execution environments for different classes of Petri nets. In addition, because of Maude's reflective capabilities [Cla98], the Petri nets thus represented cannot only be executed, but they can also be formally analyzed and model checked by means of *rewriting strategies* that explore and analyze at the metalevel the different rewriting computations of a given rewrite specification.

The general way of representing Petri nets within rewriting logic that we propose is by no means limited to the net classes explicitly discussed in this chapter. We briefly address how similar representations could be defined for other Petri net classes, such as colored Petri nets based on (higher-order) programming languages [Jen92], nets with macroplaces [Ani91, AKMP96], nets with FIFO places [FM82, KMT88, FC88, Fan92], object-oriented variants of Petri nets [Sib94, Lak95], and object nets [Val95, Val98, Far99, Val00] where nets are viewed as token objects.

We conclude this introduction with a brief overview of the chapter: We introduce in Section 3.1 a category of place/transition nets together with a functor that associates the process semantics of Best and Devillers [BD87] with each place/transition net. We then define the rewriting specification associated with a place/transition net and establish a semantic connection in terms of a natural isomorphism at the level of symmetric monoidal categories. We conclude the section on place/transition nets by showing how test arcs can be incorporated using a slightly richer state space that satisfies certain symmetries. In Section 3.2 we generalize the rewriting semantics for place/transition nets to algebraic net specifications, which we view as colored net specifications over membership equational logic. As it is the case for rewriting logic, the concept of colored net specifications is quite general, since it is parameterized over an underlying logic. However, for the sake of concreteness we only deal with rewriting logic and colored net specifications over membership equational logic in this chapter. As in the previous section we relate the Best-Devillers process semantics and the model-theoretic semantics obtained via rewriting logic in terms of a natural isomorphism. Finally, in Section 3.3 we briefly discuss how our approach can be generalized or extended to other models of Petri nets like those mentioned before.

3.1 Place/Transition Nets

Place/transition nets (PTNs) are a model of concurrency in which behavior is governed by local state changes in a distributed state space. The global distributed state of the system is represented by a *marking*, which assigns a number of indis-

tinguishable *tokens* to each *place*. State changes that may occur in the system are specified by *transitions*. Each transition can only affect the part of the marking that is local to the transition, i.e., present in the places the transition is connected to. More precisely, a local state change corresponds to the atomic occurrence of a transition which removes tokens from its *input places* and produces tokens on its *output places*. The number of tokens that are transported by an arc is specified by its *inscription*.

As an example consider the PTN modeling an instance of the well-known banker's problem depicted in Figure 3.1, which models the situation of a bank loaning money to (in this case two) clients. As usual, places and transitions are drawn as circles and rectangles, respectively. The flow relation and the weight function are given by arrows and their inscriptions. An additional initial marking is specified by place inscriptions. The money available for clients is modeled by the number of tokens in the place BANK. Furthermore, each client n has an individual credit limit modeled by a place CLAIM- n . The fact that client n requests and receives money is modeled by a transition GRANT- n and we assume that after exhausting the credit limit client n returns all the money via the transition RETURN- n .

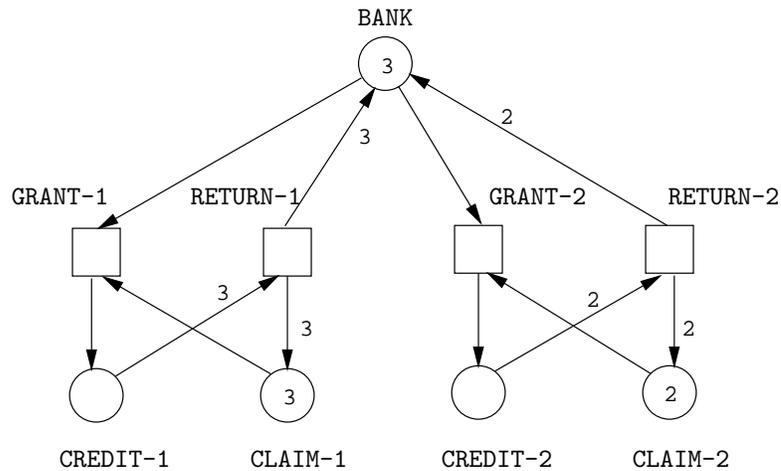


Figure 3.1: Banker's problem with two clients

We now give formal definitions of basic nets and define a PTN as a particular form of an inscribed net. Instead of just finite nets we admit infinite nets, but we restrict our attention to nets with transitions that can affect only a finite part of the marking (locality principle) so that each transition can be represented in a finitary way.

Definition 3.1.1 A net N consists of a set of *places* P_N , a set of *transitions* T_N disjoint from P_N , and a *flow relation* $F_N \subseteq (P_N \times T_N) \cup (T_N \times P_N)$ such that $\bullet t = \{p \mid p F_N t\}$ and $t^\bullet = \{p \mid t F_N p\}$ are finite for each $t \in T_N$ (local finiteness). A net is *finite* iff the sets P_N and T_N are finite.

Given nets N and N' , a *net morphism* $H : N \rightarrow N'$ consists of functions $H_P : P_N \rightarrow P_{N'}$ and $H_T : T_N \rightarrow T_{N'}$ such that $H_P(\bullet t) = \bullet H_T(t)$ and $H_P(t\bullet) = H_T(t)\bullet$. Nets together with their morphisms form a category **Net**.

A place/transition net is essentially a net with arcs inscribed by natural numbers.

Definition 3.1.2 A *place/transition net (PTN)* \mathcal{N} consists of: a net $N_{\mathcal{N}}$ and an *arc inscription* $W_{\mathcal{N}} : F_{\mathcal{N}} \rightarrow \mathbb{N}$. $W_{\mathcal{N}}$ is extended to $W_{\mathcal{N}} : (P_{\mathcal{N}} \times T_{\mathcal{N}}) \cup (T_{\mathcal{N}} \times P_{\mathcal{N}}) \rightarrow \mathbb{N}$ in such a way that $(x, y) \notin F_{\mathcal{N}}$ implies $W_{\mathcal{N}}(x, y) = 0$.

Given PTNs \mathcal{N} and \mathcal{N}' , a *PTN morphism* $H : \mathcal{N} \rightarrow \mathcal{N}'$ is a net morphism $H : N_{\mathcal{N}} \rightarrow N_{\mathcal{N}'}$ such that:

1. $W_{\mathcal{N}'}(p', t') = W_{\mathcal{N}}(p_1, t) + \dots + W_{\mathcal{N}}(p_n, t)$
for all $p' \in P_{\mathcal{N}'}$, $t' \in T_{\mathcal{N}'}$, $t \in H^{-1}(t')$,
and $\{p_1, \dots, p_n\} = H^{-1}(p') \cap \bullet t$ with distinct p_i , and
2. $W_{\mathcal{N}'}(t', p') = W_{\mathcal{N}}(t, p_1) + \dots + W_{\mathcal{N}}(t, p_n)$
for all $p' \in P_{\mathcal{N}'}$, $t' \in T_{\mathcal{N}'}$, $t \in H^{-1}(t')$,
and $\{p_1, \dots, p_n\} = H^{-1}(p') \cap t\bullet$ with distinct p_i .

PTNs together with their morphisms form a category **PTN**. Each net N can be conceived as a PTN \mathcal{N} with $N_{\mathcal{N}} = N$ and $W_{\mathcal{N}}(x, y) = 1$ iff $x F_N y$.

The notion of net morphism we use here is more restrictive than the (topological) net morphisms used in [Pet96] and close to, but slightly stronger than, the (algebraic) net morphisms used in [MM90]. The justification for our definition is that net morphisms should be morphisms in the sense of [Pet96] and should preserve the behavior in the strongest reasonable sense. Given a net morphism $H : \mathcal{N} \rightarrow \mathcal{N}'$, the intention is that the behavior of \mathcal{N} is subsumed by the behavior of \mathcal{N}' , although \mathcal{N}' may exhibit a richer behavior. In this chapter we focus on a description of behavior by Best-Devillers processes in a way that generalizes the well-known step semantics. Indeed, not only the interleaving semantics but also the step semantics and the process semantics can be regarded as labeled transition systems where the states are markings and the labels are steps or processes, respectively. In the case of Best-Devillers processes, the labeled transition system is equipped with additional algebraic structure which will be made explicit by regarding the transition system as a symmetric monoidal category.

Definition 3.1.3 Let \mathcal{N} be a PTN. A *marking* is a multiset of places. A (*concurrent*) *step* is a nonempty finite multiset of transitions. The *set of markings* and the *set of steps* are denoted by $Mrk_{\mathcal{N}}$ and $Stp_{\mathcal{N}}$, respectively. We define *preset* and *postset functions* $\partial_0, \partial_1 : T_{\mathcal{N}} \rightarrow Mrk_{\mathcal{N}}$ by $\partial_0(t)(p) = W_{\mathcal{N}}(p, t)$ and $\partial_1(t)(p) =$

$W_{\mathcal{N}}(t, p)$, respectively. The (*concurrent*) *step semantics* of a place/transition net \mathcal{N} is given by the labeled transition system which has $Mrk_{\mathcal{N}}$ as its set of states, $Stp_{\mathcal{N}}$ as its set of labels and a transition relation $\xrightarrow{e} \subseteq Mrk_{\mathcal{N}} \times Mrk_{\mathcal{N}}$ for each $e \in Stp_{\mathcal{N}}$ defined by $m_1 \xrightarrow{e} m_2$ iff there is a marking m such that, for all $p \in P_{\mathcal{N}}$,

$$\begin{aligned} m_1(p) &= m(p) + \partial_0(e)(p) \text{ and} \\ m_2(p) &= m(p) + \partial_1(e)(p) . \end{aligned}$$

Writing the occurrence rule in the way given above makes it evident that the occurrence of an action replaces its preset by its postset, whereas the remainder of the marking, here denoted by m , is not involved in this process. This is an important fact that will be made formally explicit in the process semantics that we review subsequently in a somewhat informal style. For details we refer to [DMM96] and [BD87].

Definition 3.1.4 An *occurrence net* \mathcal{N} is a net such that $F_{\mathcal{N}}$ is acyclic and $|\bullet p|, |p\bullet| \leq 1$ for each $p \in P_{\mathcal{N}}$. Given an occurrence net \mathcal{N} , $F_{\mathcal{N}}$ induces a partial order $(<) = F_{\mathcal{N}}^+$ on $P_{\mathcal{N}} \cup T_{\mathcal{N}}$, and its minimal and maximal elements are denoted by $Max(\mathcal{N})$ and $Min(\mathcal{N})$, respectively.

Let \mathcal{N} be a PTN. Then a *finite process* \mathcal{P} of \mathcal{N} with *origin* marking m_1 and *destination* marking m_2 consists of a finite occurrence net $N_{\mathcal{P}}$ and a PTN morphism $L_{\mathcal{P}} : N_{\mathcal{P}} \rightarrow \mathcal{N}$ (where $N_{\mathcal{P}}$ is viewed as a PTN) such that $L_{\mathcal{P}}(Min(N_{\mathcal{P}})) = m_1$ and $L_{\mathcal{P}}(Max(N_{\mathcal{P}})) = m_2$.

Given finite processes \mathcal{P} and \mathcal{P}' , the *parallel composition* of \mathcal{P} and \mathcal{P}' is defined as the disjoint union of the underlying nets and label functions. Given processes \mathcal{P} and \mathcal{P}' such that the destination of \mathcal{P} is equal to the origin of \mathcal{P}' , a *sequential composition* of \mathcal{P} and \mathcal{P}' is obtained by disjoint union (as above) pairwise identifying maximal places of \mathcal{P} with minimal places of \mathcal{P}' , where every two places to be identified must have the same label. Notice that in general the result of sequential composition is not unique [DMM96].

Intuitively, a process of a PTN is generated by “temporal unfolding” starting from a marking that becomes the origin of the process. Observe that for a given finite process \mathcal{P} of \mathcal{N} , not only $Min(\mathcal{P})$ and $Max(\mathcal{P})$ but each snapshot (S-cut in the sense of [BF88]) of \mathcal{P} corresponds to a marking of \mathcal{N} by virtue of $L_{\mathcal{P}}$. The ambiguity of the result of sequential composition is caused by a snapshot corresponding to a marking with several identical tokens in some place, say p . Consider a transition in \mathcal{N} that removes one token from p . A single firing of this transition gives rise to two different processes, since identical tokens are represented by different places in the process net. An obvious solution to avoid this ambiguity is to restrict our attention to *safe processes*, i.e. processes that take place in the safe part of the state space where such situations do not occur.

A marking m is said to be a *safe* marking iff all markings m' reachable from m in the step semantics satisfy $m'(p) \leq 1$ for all $p \in P$. A process is said to be *safe* iff its origin is safe. Safe processes coincide with the classical notion of processes if we consider 1-safe PTNs which are equivalent to contact-free elementary net systems [BF88, Pet96]. Our definition of safe processes is restrictive enough to ensure that the class of safe finite processes is always closed under sequential composition, a property that is not shared by the subclass of finite processes with the weaker property that all markings m corresponding to snapshots (S-cuts) satisfy $m(p) \leq 1$ for each $p \in P$.

Definition 3.1.5 A *(strict) monoidal category (MC)* \mathbf{C} is a category equipped with a monoidal operation $_{\mathbf{C}} \otimes _{\mathbf{C}}$ and an identity object $\text{id}_{\mathbf{C}}$ such that $_{\mathbf{C}} \otimes _{\mathbf{C}}$ is an associative bifunctor with left and right identity $\text{id}_{\mathbf{C}}$. A monoidal category morphism $h : \mathbf{C} \rightarrow \mathbf{C}'$ is a functor that preserves $_{\mathbf{C}} \otimes _{\mathbf{C}}$ and id , i.e., $h(u \otimes_{\mathbf{C}} v) = h(u) \otimes_{\mathbf{C}'} h(v)$ and $h(\text{id}_{\mathbf{C}}) = \text{id}_{\mathbf{C}'}$. If in addition $_{\mathbf{C}} \otimes _{\mathbf{C}}$ is commutative, then we say that \mathbf{C} is a *(strictly) symmetric (strict) monoidal category (SMC)*. The category of SMCs is denoted by **SMC**.

A variation of an SMC is a *partial SMC* \mathbf{C} where $_{\mathbf{C}} \otimes _{\mathbf{C}}$ is a partial functor and each equation in the definition of SMCs is only required to be satisfied iff both sides are defined. The category of partial SMCs is denoted by **PSMC**. Clearly, **SMC** is a subcategory of **PSMC**.

Definition 3.1.6 The *safe process semantics* $\mathbf{SP}(\mathcal{N})$ of a PTN \mathcal{N} is given by a partial SMC that has safe markings as objects and safe processes as arrows. Arrow composition is given by sequential composition, the partial monoidal operation is given by parallel composition, and the identity for an object m is given by the finite process without transitions with origin m and destination m . **SP** can be extended to a functor $\mathbf{SP} : \mathbf{SPTN} \rightarrow \mathbf{PSMC}$, where **SPTN** is the subcategory of **PTN** obtained by restricting morphisms to safe PTN morphisms. Here, a PTN morphism $H : \mathcal{N} \rightarrow \mathcal{N}'$ is *safe* iff it maps each safe marking in \mathcal{N} to a safe marking in \mathcal{N}' . Now **SP** lifts each safe PTN morphism $H : \mathcal{N} \rightarrow \mathcal{N}'$ to a functor $\mathbf{SP}(H) : \mathbf{SP}(\mathcal{N}) \rightarrow \mathbf{SP}(\mathcal{N}')$ defined in the obvious way.

If we restrict our attention to safe markings there is a close correspondence between the step semantics and the process semantics: Each step sequence, i.e. each computation w.r.t. the step semantics, generates a unique process, and a process determines a set of step sequences that contains the original one. As a consequence processes are more abstract than step sequences. A similar correspondence exists for the interleaving semantics, i.e., if we restrict steps to single transitions. Both correspondences are investigated in [BD87]. On the other hand, the authors of [BD87] observe that step sequences and processes become incomparable when we admit markings that are not safe, which means that the natural

view of processes as an abstraction of step sequences does not hold anymore. In order to recover this correspondence a more abstract notion of process is needed, and in fact Best-Devillers processes [BD87], which became also known as commutative processes [MM90, DMM96], provide such a notion. In contrast to processes which adhere to the *individual token philosophy*,¹ Best-Devillers processes share with step sequences the *collective token philosophy*, meaning that identical tokens on a place in the system are not distinguished in the process. This allows us to define an operation of sequential composition that has a unique result whenever sequential composition is possible. The following definition of Best-Devillers processes is equivalent to the definition given in [BD87], except for the fact that [BD87] does not make explicit the algebraic and categorical structure.

Definition 3.1.7 Let \mathcal{P} and \mathcal{P}' be finite processes and let $p_1, p_2 \in P_{\mathcal{P}}$ with $L_{\mathcal{P}}(p_1) = L_{\mathcal{P}}(p_2)$. We define a predicate $\text{swap}(\mathcal{P}, \mathcal{P}', p_1, p_2)$ which holds iff $P_{\mathcal{P}'} = P_{\mathcal{P}}$, $T_{\mathcal{P}'} = T_{\mathcal{P}}$, $L_{\mathcal{P}'} = L_{\mathcal{P}}$ and:

1. $t F_{\mathcal{P}'} p \Leftrightarrow t F_{\mathcal{P}} p$,
2. $p F_{\mathcal{P}'} t \Leftrightarrow p F_{\mathcal{P}} t$ if $p \neq p_1$ and $p \neq p_2$,
3. $p_1 F_{\mathcal{P}'} t \Leftrightarrow p_2 F_{\mathcal{P}} t$,
4. $p_2 F_{\mathcal{P}'} t \Leftrightarrow p_1 F_{\mathcal{P}} t$.

We define an *equivalence* on finite processes as the smallest equivalence relation that contains $(\mathcal{P}, \mathcal{P}')$ if there are $p_1, p_2 \in P_{\mathcal{P}}$ such that $L_{\mathcal{P}}(p_1) = L_{\mathcal{P}}(p_2)$ and $\text{swap}(\mathcal{P}, \mathcal{P}', p_1, p_2)$ holds. The equivalence classes are called *Best-Devillers processes*.

The notions of *origin*, *destination*, and *parallel/sequential composition* of processes are lifted to Best-Devillers processes in the obvious way. At this level the result of sequential composition becomes unique, since all potentially different results obtained by composing two processes fall into the same equivalence class.

Definition 3.1.8 The *Best-Devillers process semantics* $\mathbf{BDP}(\mathcal{N})$ of a PTN \mathcal{N} is given by an SMC that has markings as objects and Best-Devillers processes as arrows. Arrow composition is given by sequential composition, the monoidal operation is given by parallel composition, and the identity for an object m is given by the Best-Devillers process without transitions with origin m and destination m . \mathbf{BDP} can be extended to a functor $\mathbf{BDP} : \mathbf{PTN} \rightarrow \mathbf{SMC}$ that sends each PTN morphism $H : \mathcal{N} \rightarrow \mathcal{N}'$ to a functor $\mathbf{BDP}(H) : \mathbf{BDP}(\mathcal{N}) \rightarrow \mathbf{BDP}(\mathcal{N}')$.

¹A functorial semantics following the individual token philosophy has recently been given in [BMMS99] by using pre-nets, a refinement of PTNs.

The above definition is also equivalent to the one given in [DMM96], although we define Best-Devillers processes as a quotient of (classical) processes as in [BD87] rather than as a quotient of *concatenable processes* as in [DMM96]. Concatenable processes are a slight refinement of finite (classical) processes: a concatenable process is a finite process together with a total ordering of $\{p \in \text{Min}(\mathcal{N}) \mid L(p) = p'\}$ for each place p' in the origin and a total ordering of $\{p \in \text{Max}(\mathcal{N}) \mid L(p) = p'\}$ for each place p' in the destination. Using this refined notion of process the obvious definition of sequential composition, where places are only identified if they have the same position in this order, yields a unique result, which allows us to view the class of concatenable processes as a category.

Since a safe process is only equivalent to itself, it corresponds to a Best-Devillers process given by a singleton equivalence class. Hence each safe process can be regarded as a Best-Devillers process giving rise to an injection $\iota(\mathcal{N}) : \mathbf{SP}(\mathcal{N}) \rightarrow \mathbf{BDP}(\mathcal{N})$. Actually we can state the following stronger

Remark 3.1.9 $\mathbf{SP} : \mathbf{SPTN} \rightarrow \mathbf{PSMC}$ is a subfunctor of $\mathbf{BDP} : \mathbf{SPTN} \rightarrow \mathbf{PSMC}$ (the obvious restriction of $\mathbf{BDP} : \mathbf{PTN} \rightarrow \mathbf{SMC}$) as witnessed by $\iota : \mathbf{SP} \rightarrow \mathbf{BDP}$ which is in fact a natural transformation.

We take this remark as a justification for focusing primarily on the Best-Devillers processes in the following, keeping in mind that classical safe processes form an important subcategory. In the context of nets with individual tokens we shall give some additional arguments for the relevance of this subcategory.

3.1.1 A Toy Example

Rewriting logic can provide a direct semantics of PTNs following the motto “Petri nets are monoids” advocated in [MM90]. In fact, the categorical semantics presented in that work and also the relation between PTNs and linear logic explained in [MOM91b] inspired the development of rewriting logic.

The PTN of the banker’s problem can be represented by the following RWS given in Maude syntax [CDE⁺99a, CDE⁺00b], which consists of a MES specification and a set of rewrite rules. As usual in Maude, the rewrite kind `[Marking]` is implicitly introduced by introducing a sort `Marking` of this kind.²

```
sort Marking .
```

```
op empty : -> Marking .
```

```
op _>_ : Marking Marking -> Marking [assoc comm id: empty] .
```

²In fact, here and in the rest of the chapter `Marking` and `[Marking]` can be identified, since the latter does not contain any additional (error) elements (cf. [BJM00, Mes98]).

```

ops BANK CREDIT-1 CREDIT-2 CLAIM-1 CLAIM-2 : -> Marking .

r1 [GRANT-1] : BANK CLAIM-1 => CREDIT-1 .

r1 [RETURN-1] : CREDIT-1 CREDIT-1 CREDIT-1 =>
                BANK BANK BANK CLAIM-1 CLAIM-1 CLAIM-1 .

r1 [GRANT-2] : BANK CLAIM-2 => CREDIT-2 .

r1 [RETURN-2] : CREDIT-2 CREDIT-2 =>
                BANK BANK CLAIM-2 CLAIM-2 .

```

Here we have applied the translation of PTNs into rewriting logic suggested in [Mes92], which is closely related to the translation of PTNs into linear logic [MOM91b]. A marking is represented as an element of the finite multiset sort `Marking`. The constant `empty` represents the empty marking and `--` is the corresponding multiset union operator. Associativity, commutativity, and identity laws are specified as structural equations by the operator attributes in square brackets. For each place p there is a constant p , called *token constructor*, representing a single token residing in that place. In fact, under the initial semantics `Marking` is a multiset sort over tokens generated by these token constructors. For each transition t there is a rule, called *transition rule*, labeled by t and stating that its preset marking may be replaced by its postset marking.

As clearly demonstrated by the use of rewrite rules in the above RWS, there is an important difference between the reduction rules induced by computational equations of a MES and the rewrite rules of a RWS: The relation induced by one-step rewrites is in general neither terminating nor confluent, although there may be situations where this is the case. Only terminating systems where for each initial state there is a unique final state can be described by terminating and confluent rewrite rules. Hence this generalization is a practical necessity to represent general system models. For instance, the PTN model of the banker's problem has not only infinite executions but also finite ones due to the possibility of deadlock. Therefore, the transition system is neither terminating nor confluent in this case.

In order to control the execution of a RWS the user can specify a strategy which successively selects rewrite rules and initiates rewriting steps. For instance, in the case of the banker's example a possible strategy could avoid states which are necessarily leading to a deadlock, so that the banker stays always in the "safe" part of the state space. In applications such as net execution and analysis, the choice of a strategy will be guided by the need to explore the behavior of the system under certain conditions. Strategies are well-supported by the Maude engine

via reflection [CDE⁺99a, CDE⁺00b], i.e. the capability to represent rewrite specifications as objects and control their execution at the metalevel, which makes Maude a suitable tool not only for executing place-transition nets but also for analyzing such nets using strategies for (partial) state-space exploration and model checking.

3.1.2 Rewriting Semantics in the General Case

The rewriting semantics that has been explained in terms of the banker's example in the previous section can be conceived as a functor from the category **PTN** of place/transitions nets to the category **SMRWS** of symmetric monoidal RWSs (SMRWSs) that will be introduced next. The characteristic feature of SMRWSs is that their underlying specification has a single rewrite kind **[Marking]** that is specified to be a free commutative monoid over a set of constants. The definition of SMRWSs given below is quite restrictive, but is sufficient for the rewriting semantics of PTNs. In Section 3.2.4 SMRWSs will be generalized to provide a rewriting semantics for nets with individual tokens.

Definition 3.1.10 A RWS \mathcal{R} is a *symmetric monoidal RWS (SMRWS)* iff the following conditions are satisfied:

1. $\mathcal{E}_{\mathcal{R}}^D$ is empty.
2. $\mathcal{E}_{\mathcal{R}}$ contains precisely the following:
 - (a) a kind **[Marking]** together with operator symbols

$$\begin{aligned} \text{empty} &: \rightarrow [\text{Marking}], \\ _ &: [\text{Marking}] [\text{Marking}] \rightarrow [\text{Marking}]; \end{aligned}$$

- (b) any number of operator symbols of the general form

$$p : \rightarrow [\text{Marking}];$$

- (c) the parallel composition axioms

$$\begin{aligned} \forall u, v, w : [\text{Marking}] . u (v w) &= (u v) w, \\ \forall u, v : [\text{Marking}] . u v &= v u, \\ \forall u : [\text{Marking}] . \text{empty } u &= u. \end{aligned}$$

3. Rules in $R_{\mathcal{R}}$ do not have conditions and do not contain any variables.

Given two SMRWSs \mathcal{R} and \mathcal{R}' , a SMRWS morphism $H : \mathcal{R} \rightarrow \mathcal{R}'$ is a RWS morphism that preserves **[Marking]**, **empty** and \dots . SMRWSs together with their morphisms form a subcategory of **RWS** denoted **SMRWS**.

We now specialize the approach of Section 2.4.5 to obtain the initial model-theoretic semantics $\mathbf{I}(\mathcal{R})$ of a SMRWS \mathcal{R} . To this end, we first define the model-theoretic semantics of \mathcal{R} by means of a MES $\mathbf{E}(\mathcal{R})$ which has a standard model-theoretic semantics in terms of $\mathbf{E}(\mathcal{R})$ -algebras. Having done that, we then define $\mathbf{I}(\mathcal{R})$ as $\mathbf{I}(\mathbf{E}(\mathcal{R}))$, i.e., as the initial model of $\mathbf{E}(\mathcal{R})$. We have somewhat simplified the general construction of $\mathbf{E}(\mathcal{R})$ by exploiting the fact that SMRWSs have only a single rewrite kind **[Marking]**, and rewrites do never occur below other rewrites due to the restricted form of the rules.

Definition 3.1.11 The *membership equational presentation* of a SMRWS \mathcal{R} is a MES $\mathbf{E}(\mathcal{R})$ that extends $\mathcal{E}_{\mathcal{R}}$, the underlying MES of \mathcal{R} , by the following:

1. a new kind **[RawProc]** together with new operator symbols called *identity*, *sequential*, and *parallel proof constructors*, respectively

$$\begin{aligned} \text{id} &: [\text{Marking}] \rightarrow [\text{RawProc}], \\ _ _ &: [\text{RawProc}] [\text{RawProc}] \rightarrow [\text{RawProc}], \\ _ ; _ &: [\text{RawProc}] [\text{RawProc}] \rightarrow [\text{RawProc}]; \end{aligned}$$

2. a new operator symbol called *basic proof constructor*

$$t : \rightarrow [\text{RawProc}]$$

for each rule $t : M \rightarrow N$ in $R_{\mathcal{R}}$;

3. a kind **[Proc]** with a sort **Proc** and an operator symbol

$$_ : _ \rightarrow _ : [\text{RawProc}] [\text{Marking}] [\text{Marking}] \rightarrow [\text{Proc}];$$

4. a membership axiom

$$t : M \rightarrow N$$

for each rule $t : M \rightarrow N$ in $R_{\mathcal{R}}$, where we introduce the notation

$$P : M \rightarrow N \quad \text{as a shorthand for} \quad (P : M \rightarrow N) : \text{Proc};$$

5. membership axioms corresponding to the deductive rules of RWL specialized to the underlying MET $\mathcal{E}_{\mathcal{R}}$, namely:

(a) *identity*:

$$\text{id}(u) : u \rightarrow u$$

(b) *composition*:

$$\alpha; \beta : u_1 \rightarrow u_3 \text{ if } \alpha : u_1 \rightarrow u_2 \wedge \beta : u_2 \rightarrow u_3$$

(c) *compatibility* of parallel composition:

$$\alpha_1 \alpha_2 : u_1 u_2 \rightarrow u'_1 u'_2 \text{ if } \alpha_1 : u_1 \rightarrow u'_1 \wedge \alpha_2 : u_2 \rightarrow u'_2$$

6. equational axioms corresponding to the standard rewriting logic axioms, namely:

(a) *identity*:

$$\begin{aligned} \text{id}(u); \alpha &= \alpha \text{ if } \alpha : u \rightarrow u' \\ \alpha; \text{id}(u') &= \alpha \text{ if } \alpha : u \rightarrow u' \end{aligned}$$

(b) *associativity*:

$$\begin{aligned} \alpha; (\beta; \gamma) &= (\alpha; \beta); \gamma \\ \text{if } \alpha : u_1 \rightarrow u_2 \wedge \beta : u_2 \rightarrow u_3 \wedge \gamma : u_3 \rightarrow u_4 \end{aligned}$$

(c) *functoriality* of the parallel composition operator:

$$\begin{aligned} \text{id}(u_1) \text{id}(u_2) &= \text{id}(u_1 u_2) \\ (\alpha_1; \beta_1)(\alpha_2; \beta_2) &= (\alpha_1 \alpha_2); (\beta_1 \beta_2) \\ \text{if } \alpha_1 : u_1 \rightarrow v_1 \wedge \beta_1 : v_1 \rightarrow w_1 \wedge \\ &\alpha_2 : u_2 \rightarrow v_2 \wedge \beta_2 : v_2 \rightarrow w_2 \end{aligned}$$

(d) *inherited equations* for the parallel composition operator:

$$\begin{aligned} \alpha_1 (\alpha_2 \alpha_3) &= (\alpha_1 \alpha_2) \alpha_3 \\ \text{if } \alpha_1 : u_1 \rightarrow u'_1 \wedge \alpha_2 : u_2 \rightarrow u'_2 \wedge \alpha_3 : u_3 \rightarrow u'_3 \\ \alpha_1 \alpha_2 &= \alpha_2 \alpha_1 \\ \text{if } \alpha_1 : u_1 \rightarrow u'_1 \wedge \alpha_2 : u_2 \rightarrow u'_2 \\ \text{id}(\text{empty}) \alpha &= \alpha \text{ if } \alpha : u \rightarrow u' \end{aligned}$$

For better readability we leave universal quantifiers implicit: $u, u', v, w, u_i, u'_i, v_i, w_i$ are distinct variables of kind [Marking] and $\alpha, \beta, \gamma, \alpha_i, \beta_i, \gamma_i$ are distinct variables of kind [RawProc].

E can be extended to a functor $\mathbf{E} : \mathbf{SMRWS} \rightarrow \mathbf{MES}$ in the obvious way. Furthermore, composing $\mathbf{E} : \mathbf{SMRWS} \rightarrow \mathbf{MES}$ with the functor $\mathbf{Mod} : \mathbf{MES} \rightarrow \mathbf{Cat}^{\text{op}}$ we obtain $\mathbf{Mod} \circ \mathbf{E} : \mathbf{SMRWS} \rightarrow \mathbf{Cat}^{\text{op}}$ which is also denoted $\mathbf{Mod} : \mathbf{SMRWS} \rightarrow \mathbf{Cat}^{\text{op}}$. As usual we write \mathbf{U}_H for $\mathbf{Mod}(H)$ given a SMRWS morphism H .

In this chapter we are not interested in the entire algebraic structure of SMRWS models. Instead, our first goal is to relate two different semantics of PTNs, namely, the Best-Devillers process semantics and the rewriting semantics of Definition 3.1.8, in terms of SMCs. In other words, the category **SMC** will serve as a common basis and suitable level of abstraction to compare different descriptions. Below, the initial models of SMRWSs, that are defined in terms of a functor **I**, will be uniformly mapped into the same domain via a forgetful functor **V**.

Definition 3.1.12 Let $\Sigma(\mathbf{Mod})$ be the Grothendieck construction for the functor $\mathbf{Mod} : \mathbf{SMRWS} \rightarrow \mathbf{Cat}^{\text{op}}$ and let $\pi_1 : \Sigma(\mathbf{Mod}) \rightarrow \mathbf{SMRWS}$ be the obvious projection functor that sends (\mathcal{R}, A) to \mathcal{R} . Given a SMRWS \mathcal{R} , we define $\mathbf{I}(\mathcal{R})$ as $\mathbf{I}(\mathbf{E}(\mathcal{R}))$ and $\Sigma\mathbf{I}(\mathcal{R})$ as $(\mathcal{R}, \mathbf{I}(\mathcal{R}))$. Given a SMRWS morphism $H : \mathcal{R} \rightarrow \mathcal{R}'$, we define $\Sigma\mathbf{I}(H)$ as the morphism $(H, \mathbf{I}(H)) : (\mathcal{R}, \mathbf{I}(\mathcal{R})) \rightarrow (\mathcal{R}', \mathbf{I}(\mathcal{R}'))$ with $\mathbf{I}(H)$ the unique morphism $\mathbf{I}(H) : \mathbf{I}(\mathcal{R}) \rightarrow \mathbf{U}_H(\mathbf{I}(\mathcal{R}'))$ guaranteed by the fact that $\mathbf{I}(\mathcal{R})$ and $\mathbf{U}_H(\mathbf{I}(\mathcal{R}'))$ are objects in $\mathbf{Mod}(\mathcal{R})$ with the former being initial. In this way we have defined a functor $\Sigma\mathbf{I} : \mathbf{SMRWS} \rightarrow \Sigma(\mathbf{Mod})$ that is left adjoint to π_1 .

Let $\mathbf{V} : \Sigma(\mathbf{Mod}) \rightarrow \mathbf{SMC}$ be the forgetful functor which sends (\mathcal{R}, \hat{A}) to the SMC defined as follows: The sets of objects and arrows are $\llbracket \mathbf{Marking} \rrbracket_{\hat{A}}$ and $\llbracket \mathbf{Proc} \rrbracket_{\hat{A}}$, respectively. Arrow composition is $\llbracket -; - \rrbracket_{\hat{A}}$ and identities are $\llbracket \text{id} \rrbracket_{\hat{A}}(m)$ for $m \in \llbracket \mathbf{Marking} \rrbracket_{\hat{A}}$. The monoidal operation and its identity are given by $\llbracket - \rrbracket_{\hat{A}}$ and $\llbracket \text{empty} \rrbracket_{\hat{A}}$, respectively. Given a morphism $(H, h) : (\mathcal{R}, \hat{A}) \rightarrow (\mathcal{R}', \hat{A}')$ in $\Sigma(\mathbf{Mod})$, we define $\mathbf{V}(H, h)$ as the SMC morphism given by the obvious restriction of h .

The rewriting semantics of PTNs is then defined as follows:

Definition 3.1.13 Given a PTN \mathcal{N} , the *rewriting semantics* of \mathcal{N} is the smallest SMRWS $\mathbf{R}(\mathcal{N})$ such that:

1. $\mathcal{E}_{\mathbf{R}(\mathcal{N})}$ contains a *token constructor*

$$p : \rightarrow [\mathbf{Marking}]$$

for each place $p \in P_{\mathcal{N}}$;

2. $\mathbf{R}(\mathcal{N})$ has a label t and a rule called a *transition rule*, namely,

$$t : \underbrace{p_1 \dots p_1}_{W(p_1, t)} \dots \underbrace{p_m \dots p_m}_{W(p_m, t)} \rightarrow \underbrace{p_1 \dots p_1}_{W(t, p_1)} \dots \underbrace{p_m \dots p_m}_{W(t, p_m)}$$

for each transition $t \in T_{\mathcal{N}}$ assuming $P_{\mathcal{N}} = \{p_1, \dots, p_m\}$ with distinct p_i .

R can be extended to a functor $\mathbf{R} : \mathbf{PTN} \rightarrow \mathbf{SMRWS}$ that maps each PTN morphism $H : \mathcal{N} \rightarrow \mathcal{N}'$ to the unique SMRWS morphism $G : \mathbf{R}(\mathcal{N}) \rightarrow \mathbf{R}(\mathcal{N}')$ with $G_L(t) = H(t)$ for each $t \in T_{\mathcal{N}}$ and $G_{\mathcal{E}}(p) = H(p)$ for each $p \in P_{\mathcal{N}}$.

The main result in this section states that for a PTN \mathcal{N} the Best-Devillers process semantics $\mathbf{BDP}(\mathcal{N})$ coincides with the initial semantics of $\mathbf{R}(\mathcal{N})$ in the strongest possible categorical sense of a natural isomorphism.

In fact, this theorem is closely related to and can be proved using a result in [DMM96] (Theorem 27), which states that the monoidal category $\mathcal{CP}(\mathcal{N})$ of concatenable processes and a monoidal category $\mathcal{P}(\mathcal{N})$ defined by an inductive equational definition are isomorphic. Both $\mathcal{CP}(\mathcal{N})$ and $\mathcal{P}(\mathcal{N})$ are not symmetric, but they still enjoy certain symmetries. For an exact definition of $\mathcal{CP}(\mathcal{N})$ and $\mathcal{P}(\mathcal{N})$ we refer to [DMM96].

The differences between Theorem 27 in [DMM96] and Theorem 3.1.14 below are that: (1) Theorem 3.1.14 is about Best-Devillers processes which are more abstract than concatenable processes, (2) it uses rewriting logic instead of giving a direct inductive equational definition, and (3) it states a natural isomorphism instead of just an isomorphism, that is, we use not only categories in the small, but we also aim at a systematic categorical treatment in the large.

Theorem 3.1.14 There is a natural isomorphism $\hat{\tau} : \mathbf{BDP} \rightarrow \mathbf{V} \circ \Sigma \mathbf{I} \circ \mathbf{R}$ between the functors $\mathbf{BDP} : \mathbf{PTN} \rightarrow \mathbf{SMC}$ and $\mathbf{V} \circ \Sigma \mathbf{I} \circ \mathbf{R} : \mathbf{PTN} \rightarrow \mathbf{SMC}$ (with $\mathbf{R} : \mathbf{PTN} \rightarrow \mathbf{SMRWS}$ and $\mathbf{V} \circ \Sigma \mathbf{I} : \mathbf{SMRWS} \rightarrow \mathbf{SMC}$).

Proof Sketch.

Let \mathcal{N} be a PTN. Below we refer to the categories $\mathcal{CP}(\mathcal{N})$, $\mathcal{P}(\mathcal{N})$ and $\mathcal{T}(\mathcal{N})$ defined in [DMM96]. Proposition 23 in [DMM96] establishes a homomorphism $\tau(\mathcal{N}) : \mathcal{P}(\mathcal{N}) \rightarrow \mathcal{CP}(\mathcal{N})$ preserving sequential and parallel composition of concatenable processes. More precisely, $\mathcal{P}(\mathcal{N})$ and $\mathcal{CP}(\mathcal{N})$ can be regarded as monoidal categories and $\tau(\mathcal{N})$ is a morphism in the category of monoidal categories. In the proof of Theorem 27 in [DMM96], $\tau(\mathcal{N})$ is shown to be an isomorphism.

$$\begin{array}{ccccc}
 \mathcal{P}(\mathcal{N}) & \xrightarrow{\tau(\mathcal{N})} & \mathcal{CP}(\mathcal{N}) & \xrightarrow{\pi(\mathcal{N})} & \mathcal{PR}(\mathcal{N}) \\
 \eta'' \downarrow & & \downarrow \eta' & & \downarrow \eta \\
 \mathcal{T}(\mathcal{N}) & \xrightarrow{\hat{\tau}'(\mathcal{N})} & \mathbf{BDP}'(\mathcal{N}) & \xrightarrow{\hat{\pi}(\mathcal{N})} & \mathbf{BDP}(\mathcal{N}) \\
 & \xrightarrow{\hat{\tau}(\mathcal{N})} & & &
 \end{array}$$

Figure 3.2: The Best Devillers swap construction

Now let $\eta(\mathcal{N}) : \mathcal{PR}(\mathcal{N}) \rightarrow \mathbf{BDP}(\mathcal{N})$ be the set-theoretic swap construction of Best and Devillers [BD87]. Since $\mathcal{PR}(\mathcal{N})$ is not a category (sequential composition is not unique) we lift the Best and Devillers swap construction to the richer concatenable processes giving rise to $\eta'(\mathcal{N}) : \mathcal{CP}(\mathcal{N}) \rightarrow \mathbf{BDP}'(\mathcal{N})$ such that the right square in Figure 3.2 commutes. $\pi(\mathcal{N})$ is the projection that forgets the

richer structure of concatenable processes. This projection becomes an isomorphism at the level of Best-Devillers processes which we denote by $\widehat{\pi}(\mathcal{N})$. In fact, it is easy to verify that $\widehat{\pi} : \mathbf{BDP}' \rightarrow \mathbf{BDP}$ is a natural isomorphism between functors from **PTN** to **SMC**.

As stated in Theorem 32 of [DMM96], $\mathcal{T}(\mathcal{N})$ is equal to the extension of $\mathcal{P}(\mathcal{N})$ by the swap axiom (notice the abuse of language here). Let $\eta''(\mathcal{N}) : \mathcal{P}(\mathcal{N}) \rightarrow \mathcal{T}(\mathcal{N})$ be the quotient homomorphism induced by this extension. Now the informal statement in [DMM96] that the swap axiom directly represents the swap construction by Best and Devillers can be made precise as the commutativity of the left square in Figure 3.2, where $\widehat{\tau}'(\mathcal{N}) : \mathcal{T}(\mathcal{N}) \rightarrow \mathbf{BDP}'(\mathcal{N})$ makes explicit the isomorphism of Corollary 33 [DMM96]. Notice that $\mathbf{BDP}'(\mathcal{N})$ is an SMC and $\mathcal{T}(\mathcal{N})$ can also be regarded as such. The isomorphism $\widehat{\tau}'(\mathcal{N})$ inherits from $\tau(\mathcal{N})$ the property that it preserves sequential and parallel composition of Best-Devillers processes. Hence, $\widehat{\tau}'(\mathcal{N})$ is an isomorphism in **SMC**.

It is not difficult to see that $\mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(\mathcal{N})))$ is a precise formalization of $\mathcal{T}(\mathcal{N})$ using rewriting logic, which means that $\widehat{\tau}'$ is actually an isomorphism $\widehat{\tau}'(\mathcal{N}) : \mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(\mathcal{N}))) \rightarrow \mathbf{BDP}'(\mathcal{N})$ in **SMC**.

To show that $\widehat{\tau}'$ constitutes a natural isomorphism it is sufficient to verify naturality: Given a PTN morphism

$$H : \mathcal{N} \rightarrow \mathcal{N}'$$

and morphisms

$$\begin{aligned} \mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(H))) : \mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(\mathcal{N}))) &\rightarrow \mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(\mathcal{N}')))) \\ \mathbf{BDP}'(H) : \mathbf{BDP}'(\mathcal{N}) &\rightarrow \mathbf{BDP}'(\mathcal{N}'), \end{aligned}$$

we have to verify

$$\widehat{\tau}'(\mathcal{N}') \circ \mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(H))) = \mathbf{BDP}'(H) \circ \widehat{\tau}'(\mathcal{N}).$$

Indeed, let \widehat{x} denote the elementary concatenable processes generated by x and let $[P] = \eta'(P)$ denote the Best-Devillers process induced by the concatenable process P . Then we have

$$\begin{aligned} &\widehat{\tau}'(\mathcal{N}')(\mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(H))))(\llbracket p \rrbracket_{\mathbf{I}(\mathbf{R}(\mathcal{N}))}) \\ &= \widehat{\tau}'(\mathcal{N}')(\mathbf{I}(\mathbf{R}(H)))(\llbracket p \rrbracket_{\mathbf{I}(\mathbf{R}(\mathcal{N}))}) \\ &= \widehat{\tau}'(\mathcal{N}')(\llbracket p \rrbracket_{\mathbf{U}_{\mathbf{R}(H)}(\mathbf{I}(\mathbf{R}(\mathcal{N}')))})) \\ &= \widehat{\tau}'(\mathcal{N}')(\llbracket H(p) \rrbracket_{\mathbf{I}(\mathbf{R}(\mathcal{N}'))}) = [\widehat{H(p)}] \end{aligned}$$

and

$$\mathbf{BDP}'(H)(\widehat{\tau}'(\mathcal{N}))(\llbracket p \rrbracket_{\mathbf{I}(\mathbf{R}(\mathcal{N}))}) = \mathbf{BDP}'(H)(\llbracket \widehat{p} \rrbracket) = [\widehat{H(p)}].$$

Similarly,

$$\widehat{\tau}(\mathcal{N}')(\mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(H))))(\llbracket t \rrbracket_{\mathbf{I}(\mathbf{R}(\mathcal{N}))}) = [\widehat{H}(t)]$$

and

$$\mathbf{BDP}'(H)(\widehat{\tau}(\mathcal{N}))(\llbracket t \rrbracket_{\mathbf{I}(\mathbf{R}(\mathcal{N}))}) = [\widehat{H}(t)].$$

Observe that these equations extend to arbitrary objects in $\mathbf{V}(\Sigma\mathbf{I}(\mathbf{R}(\mathcal{N})))$.

The natural transformation $\widehat{\tau}$ claimed by the theorem is finally given by the vertical composition $\widehat{\tau}' \circ \widehat{\pi}$ of natural transformations. □

In particular, the previous theorem entails that for each individual PTN we have precisely characterized Best-Devillers processes in rewriting logic via \mathbf{R} as stated by the corollary below. As a byproduct we have obtained a corresponding characterization in membership equational logic via \mathbf{E} .

Corollary 3.1.15 The rewrite specification $\mathbf{R}(\mathcal{N})$ provides a sound and complete axiomatization of the Best-Devillers processes of the PTN \mathcal{N} .

Again, this is closely related to Corollary 33 in [DMM96], which states that the presentation of an SMC denoted by $\mathcal{T}(\mathcal{N})$ provides a complete and sound axiomatization of Best-Devillers processes. Similar to the category $\mathcal{P}(\mathcal{N})$ mentioned before, $\mathcal{T}(\mathcal{N})$ is given by a direct inductive equational definition, whereas here we use the SMRWS $\mathbf{R}(\mathcal{N})$ to express the same category. In other words we use rewriting logic to equip the presentation of $\mathcal{T}(\mathcal{N})$ itself with a first-class formal status.

3.1.3 Petri Nets with Test Arcs

Following [SMÖ01b] we illustrate in this section how the techniques for giving a rewriting logic semantics to place/transition nets can be extended to deal with the important class of place/transition nets with *test arcs* [CH93, MR95, Vog97, BS00]. Petri nets have been equipped with test arcs (also called *read arcs*, or *positive contexts* in contextual nets [MR95]) to naturally model cases where a certain resource may be *read without being consumed* by a transition, such as in a database system where multiple users are allowed to simultaneously read the same piece of data. In contrast to ordinary arcs, several test arcs are allowed to access the same token in the same concurrent step, but a token accessed by a test arc may not be accessed by an ordinary arc in the same step.³ Test arcs cannot change the marking of a place.

³This last restriction is omitted in some definitions of Petri nets with test arcs (see e.g. [Vog97]).

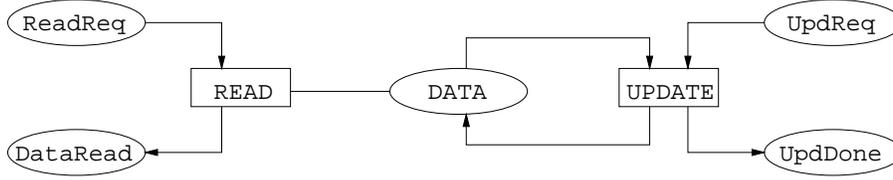


Figure 3.3: Small database example using test arcs.

Formally, a place/transition net with test arcs \mathcal{N} is a place/transition net together with a set of *test arcs* $TA_{\mathcal{N}} \subseteq P_{\mathcal{N}} \times T_{\mathcal{N}}$. We define the *context function* $\partial_{TA} : Stp_{\mathcal{N}} \rightarrow Mrk_{\mathcal{N}}$ on steps e by $\partial_{TA}(e)(p) = 1$ if there is a transition $t \in e$ with $(p, t) \in TA_{\mathcal{N}}$, and by $\partial_{TA}(e)(p) = 0$ otherwise. The *step semantics* of a place/transition net with test arcs is defined as for place/transition nets (see Section 3.1) with the modification that for $m_1 \xrightarrow{e} m_2$ to hold we require additionally that, for each place $p \in P_{\mathcal{N}}$, $\partial_{TA}(e)(p) \leq m(p)$.

We propose a rewriting semantics for a place/transition net with test arcs, defined in terms of a rewrite specification $\mathbf{R}(\mathcal{N})$ similar to the one in Definition 3.1.13, but specifying tokens by means of a kind $[\text{Place}]$ and two operators $[-], \langle - \rangle : [\text{Place}] \rightarrow [\text{Marking}]$, so that a token residing at place p is represented by the term $[p]$. An occurrence of $[p]$ may not be shared by more than one rewrite at the same time; to allow simultaneous rewrites with *read-only* access to a token at place p , we consider a token $[p]$ to be equivalent to an arbitrary number of read-only tokens of the form $\langle p \rangle$. This can be accomplished, using a technique described in [Mes96], by adding to our specification $\mathbf{R}(\mathcal{N})$ an operator $\{- | -\} : [\text{Marking}] \rightarrow [\text{Nat}]$ and two *copying equations*⁴

$$[p] = \{p | 0\} \quad \text{and} \quad \{p | n\} = \{p | n + 1\} \langle p \rangle,$$

where p and n are variables ranging, respectively, over $[\text{Place}]$ and $[\text{Nat}]$.

A transition t which consumes the tokens a_1, \dots, a_n , produces the tokens b_1, \dots, b_m , and “reads” the tokens c_1, \dots, c_k , is modeled by a rewrite rule

$$t : [a_1] \dots [a_n] \langle c_1 \rangle \dots \langle c_k \rangle \rightarrow [b_1] \dots [b_m] \langle c_1 \rangle \dots \langle c_k \rangle.$$

The database example in Figure 3.3, taken from [CH93], where multiple users may read some data simultaneously, but where only one at a time is allowed to update the data, is, therefore, modeled in rewriting logic by the following rules:

$$\begin{aligned} \text{READ} : \quad & [\text{ReadReq}] \langle \text{Data} \rangle \longrightarrow [\text{DataRead}] \langle \text{Data} \rangle \\ \text{UPDATE} : \quad & [\text{UpdReq}] [\text{Data}] \longrightarrow [\text{UpdDone}] [\text{Data}]. \end{aligned}$$

⁴The counting of the read-only copies and their read-only use guarantee that all the copies must have been “folded back together” in order for the original token to be engaged in a transition that consumes the token.

Let $\mathbf{R}(\mathcal{N})$ be the rewrite specification representing a place/transition net with test arcs \mathcal{N} as explained above, and for any marking m in \mathcal{N} , let m^\sharp denote the term of kind `[Marking]` which contains exactly $m(p)$ occurrences of the term `[p]` for each place p in \mathcal{N} . Then, there is a step $m_1 \xrightarrow{e} m_2$ in \mathcal{N} iff there is a one-step concurrent rewrite $P : m_1^\sharp \rightarrow m_2^\sharp$ in $\mathbf{R}(\mathcal{N})$, where, in addition, the step e can be extracted from the proof P . Furthermore, as in Definition 3.1.12 we can define a functor that associates with $\mathbf{R}(\mathcal{N})$ a symmetric monoidal category determined by the initial semantics. This provides a categorical semantics for all the concurrent computations of the net \mathcal{N} that is closely related to the one recently proposed by Bruni and Sassone in [BS00].

3.2 High-Level Petri Nets

We use the term *high-level Petri nets* to refer to a range of extensions of PTNs by individual tokens, a line of research that has been initiated by the introduction of *predicate/transition nets* in [GL79, GL81, Gen91]. High-level Petri nets make use of an underlying formalism, such as first-order logic in the case of predicate/transition nets, to describe the information that is associated with each token and its transformation. *Colored nets*⁵ introduced in [Jen81] are another quite general model of this kind with a more set-theoretic flavour. They generalize PTNs in such a way that tokens can be arbitrary set-theoretic objects. Quite different from, but closely related to, colored nets are high-level Petri nets that use an algebraic specification language as an underlying formalism [Vau85, BCC⁺86, Vau87, RV88, Rei91, Rei98b, DHP91, BCM88]. In this chapter we subsume such approaches under the general notion of *algebraic net specifications*, parameterized over an underlying equational specification language. The main feature that algebraic net specifications have in common with predicate/transition nets is that an algebraic net specification does not necessarily specify a single colored net, but instead denotes a *class* of colored nets that satisfy the specification. In the following we first define colored nets, and then we introduce algebraic net specifications over MEL, a straightforward generalization of algebraic net specifications over many-sorted equational-logic (MSA). Both, algebraic net specifications and rewriting logic are specification formalisms that admit a variety of models. From an even more general point of view that is only briefly sketched in this chapter, one can define colored net specifications parameterized over an underlying logic. In fact, predicate/transition nets can essentially be regarded as colored net specifications over first-order logic. From this more general point of view we restrict our attention in this chapter to the

⁵In fact, the nets introduced in [Jen81] are called *colored Petri nets (CPNs)*, but this name has later been used for the more syntactic version introduced in [Jen92], which is also the sense for which we would like to reserve this term.

particular class of colored net specifications over MEL, that we also call algebraic net specifications (over MEL), to establish a systematic connection to rewriting logic (over MEL). Later, in Section 3.3, we will discuss how other high-level Petri net extensions can be covered as generalizations or variants of our approach.

3.2.1 Colored Nets and Colored Net Specifications

Algebraic net specifications will be introduced later as a formal specification language for colored nets. In the following we define the most general set-theoretic version of colored nets [Jen81]. We also give a suitable notion of colored net morphism and we use **CN** to abbreviate the resulting category of colored nets.

Colored nets are nets with places, transitions, and arcs inscribed with additional information given by functions C and W . The color set $C(p)$ of a place p is the set of possible objects p can carry. The color set $C(t)$ of a transition t can be seen as a set of modes in which t may occur. The arc inscription W defines a multiset of objects (“colored” tokens) that are transported by an arc when the associated transition occurs. In fact, this multiset may depend on the mode in which the transition occurs, which is why $W(p, t)$ and $W(t, p)$ take the form of functions in the definition below.

Definition 3.2.1 A *colored net (CN)* \mathcal{N} consists of:

1. a finite net $N_{\mathcal{N}}$;
2. a set of *color sets* $CS_{\mathcal{N}}$;
3. a *color function* $C_{\mathcal{N}} : P_{\mathcal{N}} \cup T_{\mathcal{N}} \rightarrow CS_{\mathcal{N}}$; and
4. an *arc inscription* $W_{\mathcal{N}}$ on $F_{\mathcal{N}}$ such that

$$\begin{aligned} W_{\mathcal{N}}(p, t) &: C_{\mathcal{N}}(t) \rightarrow \mathcal{FMS}(C_{\mathcal{N}}(p)), \text{ and} \\ W_{\mathcal{N}}(t, p) &: C_{\mathcal{N}}(t) \rightarrow \mathcal{FMS}(C_{\mathcal{N}}(p)). \end{aligned}$$

$W_{\mathcal{N}}$ is extended to a function on $(P_{\mathcal{N}} \times T_{\mathcal{N}}) \cup (T_{\mathcal{N}} \times P_{\mathcal{N}})$ in such a way that $(p, t) \notin F_{\mathcal{N}}$ implies $W_{\mathcal{N}}(p, t)(b) = \emptyset$ and $(t, p) \notin F_{\mathcal{N}}$ implies $W_{\mathcal{N}}(t, p)(b) = \emptyset$ for each $b \in C_{\mathcal{N}}(t)$.

Let \mathcal{N} and \mathcal{N}' be CNs. A *CN morphism* $H : \mathcal{N} \rightarrow \mathcal{N}'$ consists of a net morphism $H_N : N_{\mathcal{N}} \rightarrow N_{\mathcal{N}'}$, and functions $H_x : C_{\mathcal{N}}(x) \rightarrow C_{\mathcal{N}'}(H_N(x))$ for each $x \in P_{\mathcal{N}} \cup T_{\mathcal{N}}$ such that:

1. $W_{\mathcal{N}'}(p', t')(H_t(b)) = H_{p_1}(W_{\mathcal{N}}(p_1, t)(b)) \oplus \dots \oplus H_{p_n}(W_{\mathcal{N}}(p_n, t)(b))$
for all $p' \in P_{\mathcal{N}'}$, $t' \in T_{\mathcal{N}'}$, $t \in H_N^{-1}(t')$, $b \in C(t)$,
and $\{p_1, \dots, p_n\} = H_N^{-1}(p') \cap \bullet t$ with distinct p_i ;

2. $W_{\mathcal{N}'}(t', p')(H_t(b)) = H_{p_1}(W_{\mathcal{N}}(t, p_1)(b)) \oplus \dots \oplus H_{p_n}(W_{\mathcal{N}}(t, p_n)(b))$
 for all $p' \in P_{\mathcal{N}'}$, $t' \in T_{\mathcal{N}'}$, $t \in H_N^{-1}(t')$, $b \in C(t)$,
 and $\{p_1, \dots, p_n\} = H_N^{-1}(p') \cap t^\bullet$ with distinct p_i .

CNs together with their morphisms form a category denoted by **CN**.

CNs generalize PTNs. The two dual objects of generalization are places and transitions. PTNs arise as the special case in which $C(x)$ is a singleton set for each $x \in P \cup T$. This gives rise to an obvious inclusion functor $\iota : \mathbf{PTN} \rightarrow \mathbf{CN}$.

Although CNs can be seen as a generalization of PTNs, there is a more fundamental justification for introducing CNs, namely, that a CN is just a convenient abbreviation for a typically rather complex PTN [Jen92, Gen91]. Indeed, this connection can be exploited to lift low-level concepts such as markings, safe processes, and Best-Devillers processes to the higher level. This is achieved by the following flattening functor $(\cdot)^{\flat} : \mathbf{CN} \rightarrow \mathbf{PTN}$ which associates to each CN the PTN obtained by “spatial unfolding.” We call this operation *flattening* to clearly distinguish it from “temporal unfolding” which generates the processes of a PTN as we defined them earlier.

Definition 3.2.2 Given a CN \mathcal{N} , we define the *flattening* \mathcal{N}^{\flat} of \mathcal{N} as the unique PTN that satisfies:

1. $P_{\mathcal{N}^{\flat}} = \{(p, c) \mid p \in P_{\mathcal{N}}, c \in C_{\mathcal{N}}(p)\}$;
2. $T_{\mathcal{N}^{\flat}} = \{(t, b) \mid t \in T_{\mathcal{N}}, b \in C_{\mathcal{N}}(t)\}$;
3. $W_{\mathcal{N}^{\flat}}((p, c), (t, b)) = W_{\mathcal{N}}(p, t)(b)(c)$; and
4. $W_{\mathcal{N}^{\flat}}((t, b), (p, c)) = W_{\mathcal{N}}(t, p)(b)(c)$

for $p \in P_{\mathcal{N}}, c \in C_{\mathcal{N}}(p), t \in T_{\mathcal{N}}, b \in C_{\mathcal{N}}(t)$.

Flattening is extended to a functor $(\cdot)^{\flat} : \mathbf{CN} \rightarrow \mathbf{PTN}$ as follows: Given a CN morphism $H : \mathcal{N} \rightarrow \mathcal{N}'$, the PTN morphism $H^{\flat} : \mathcal{N}^{\flat} \rightarrow \mathcal{N}'^{\flat}$ is given by

1. $H^{\flat}((p, c)) = (H_N(p), H_p(c))$,
2. $H^{\flat}((t, b)) = (H_N(t), H_t(b))$

for $p \in P_{\mathcal{N}}, c \in C_{\mathcal{N}}(p), t \in T_{\mathcal{N}}$, and $b \in C_{\mathcal{N}}(t)$.

It is important to point out that although we have defined the notion of a colored net, we have not yet introduced a notion of finite specification of colored nets. This is unsatisfactory if we want to reason about colored net specifications instead

of just reasoning about colored nets. It is also unsatisfactory if we want to apply tools for execution, analysis and verification of colored nets, since such tools rely on a finitary, formal specification. Although a formal inscription language can be obtained by a formalization of set theory, such an enterprise is cumbersome and is of little help when we are interested in effective net execution and analysis. Also, the direct use of formalized set theory for specification and verification purposes is not very convenient and could be compared with the use of a low-level programming language.

Colored Petri nets, a more syntactic, finitary version of colored nets based on an underlying programming language, are proposed in [Jen92]. A remarkable point is that this definition leaves open the particular choice of the underlying programming language. We use $\mathbf{CPN}_{\mathcal{L}}$ to abbreviate the class of colored Petri nets over a programming language \mathcal{L} . A quite well known instance of this definition is \mathbf{CPN}_{ML} , the class supported by the execution and analysis tool Design/CPN [Jen92] that employs the functional programming language ML. Apart from their operational flavor, the essential characteristic of colored Petri nets is that each colored Petri net denotes a single well-defined colored net in the above sense. A more logic-oriented view of colored nets (which emphasizes classes of models) is given by colored net specifications that are introduced subsequently.

As a useful concept, we informally introduce *colored net specifications (CNS)* which capture the essential idea shared by predicate/transition nets and algebraic net specifications, namely, that they denote an entire class of colored nets instead of just a single one. In fact, there is a general concept of CNSs that is parameterized by an underlying logic.⁶ We denote by $\mathbf{CNS}_{\mathcal{L}}$ the class of colored net specifications over the underlying logic \mathcal{L} . Possible candidates for \mathcal{L} include equational logics such as many-sorted equational logic (MSA), order-sorted equational logic (OSA), or membership equational logic (MEL). We refer to CNSs over such equational logics also as *algebraic net specifications (ANS)*, and we denote by $\mathbf{ANS}_{\mathcal{L}}$ the class of algebraic net specifications over \mathcal{L} . Obviously, there are other possible choices for the underlying logic, such as full first-order logic (as in predicate/transition nets), a version of higher-order logic, or a higher-order algebraic specification language (as in [Hof00]).

3.2.2 Algebraic Net Specifications

In the following we use the term *algebraic net specification (ANS)* to specifically refer to ANSs over MEL, since MEL is sufficiently expressive to cover other commonly used algebraic specification languages such as MSA and OSA [Mes98].

⁶The concept of a logic, which has a deductive system and a model-theoretic semantics, is made precise by the theory of general logics [Mes89a] (cf. Section 2.3) which contain institutions [GB92, GB94] as the model-theoretic component.

The use of MEL is particularly attractive, because it is weak enough to admit initial models. Indeed, under the initial semantics (which can be internally specified using constraints in the data subspecification) an ANS denotes a unique CN. Another benefit of the use of membership equational logic is that, under the restrictions mentioned in Section 2.4.4, it comes with a natural operational semantics (which is actually implemented in the Maude engine) so that it can be used directly as a programming language or, more generally, as a metalanguage to specify the logical and operational semantics of other specification or programming languages. As a consequence, colored Petri nets in $\mathbf{CPN}_{\mathcal{L}}$ which use \mathcal{L} as a programming language can be seen as a special case of algebraic net specifications in $\mathbf{ANS}_{\mathbf{MEL}}$ if the semantics of \mathcal{L} can be specified in MEL.

Due to the fact that MEL generalizes MSA in an obvious way, ANSs over MEL are a straightforward generalization of ANSs over MSA, i.e. many-sorted algebraic net specifications. Disregarding the issue of the underlying specification language, the definition we give below is equivalent to the one in [KR96, KV98], generalizing [Rei91] by so-called *flexible arcs*, which transport variable multisets of tokens in the sense that the number of tokens transported by an arc is not fixed but can depend on the mode in which the associated transition occurs. Later, in Section 3.2.3 we will illustrate by means of an example how an executable subset of the specification language can be used to obtain executable specifications of net models.

An ANS presupposes an underlying specification that has a multiset kind for each place domain. Hence we introduce a generic notion of multiset specification first.

Definition 3.2.3 A *MES of finite multisets* over a kind k consists of:

1. a MET having kinds k and $[\mathbf{FMS}_k]$ with operator symbols

$$\begin{aligned} \mathbf{empty}_k &: [\mathbf{FMS}_k], \\ \mathbf{single} &: k \rightarrow [\mathbf{FMS}_k], \\ \mathbf{-} &: [\mathbf{FMS}_k] [\mathbf{FMS}_k] \rightarrow [\mathbf{FMS}_k]; \end{aligned}$$

equational axioms

$$\begin{aligned} \forall a, b, c : [\mathbf{FMS}_k] . a (b c) &= (a b) c, \\ \forall a, b : [\mathbf{FMS}_k] . a b &= b a, \\ \forall a : [\mathbf{FMS}_k] . \mathbf{empty}_k a &= a; \end{aligned}$$

2. and a constraint stating that this theory is free over k .

To simplify notation we write M instead of $\mathbf{single}(M)$. To further simplify the exposition we assume without loss of generality that $[[\mathbf{FMS}_k]] = \mathcal{FMS}([k])$, i.e., $[\mathbf{FMS}_k]$ is interpreted in the standard way, and the operator symbols are interpreted accordingly.

The subsequent definition of algebraic net specifications should be regarded as an instance of CNSs over a logic \mathcal{L} choosing MEL for \mathcal{L} . In fact, the only requirements that \mathcal{L} has to meet is that it has a notion of type and that it is expressive enough to axiomatize multisets.

Definition 3.2.4 An *algebraic net specification (ANS)* \mathcal{N} consists of:

1. a MES $\mathcal{E}_{\mathcal{N}}$;
2. a finite net $N_{\mathcal{N}}$;
3. a *place declaration*, i.e. a function $D_{\mathcal{N}} : P_{\mathcal{N}} \rightarrow \text{Kind}_{\mathcal{E}_{\mathcal{N}}}$ assigning a kind $D_{\mathcal{N}}(p)$ to each place $p \in P_{\mathcal{N}}$ such that $\mathcal{E}_{\mathcal{N}}$ includes a MES of finite multisets over $D_{\mathcal{N}}(p)$;
4. a *variable declaration*, i.e. a function $V_{\mathcal{N}}$ on $T_{\mathcal{N}}$ associating to each transition $t \in T_{\mathcal{N}}$ a kinded variable set $V_{\mathcal{N}}(t)$;
5. an *arc inscription*, i.e. a function $W_{\mathcal{N}}$ on $F_{\mathcal{N}}$ such that for $p \in P_{\mathcal{N}}, t \in T_{\mathcal{N}}$,
 - (a) $(p, t) \in F_{\mathcal{N}}$ implies $W_{\mathcal{N}}(p, t) \in \text{Trm}_{\mathcal{E}_{\mathcal{N}}}(V_{\mathcal{N}}(t))_{[\text{FMS}_{D_{\mathcal{N}}(p)}]}$ and
 - (b) $(t, p) \in F_{\mathcal{N}}$ implies $W_{\mathcal{N}}(t, p) \in \text{Trm}_{\mathcal{E}_{\mathcal{N}}}(V_{\mathcal{N}}(t))_{[\text{FMS}_{D_{\mathcal{N}}(p)}]}$;
6. a *guard definition*, i.e. a function $G_{\mathcal{N}}$ on $T_{\mathcal{N}}$ with $G_{\mathcal{N}}(t)$ being an $\mathcal{E}_{\mathcal{N}}$ -condition over $V_{\mathcal{N}}(t)$.

$W_{\mathcal{N}}$ is extended to a function on $(P_{\mathcal{N}} \times T_{\mathcal{N}}) \cup (T_{\mathcal{N}} \times P_{\mathcal{N}})$ such that $(p, t) \notin F_{\mathcal{N}}$ implies $W_{\mathcal{N}}(p, t) = \text{empty}_{D_{\mathcal{N}}(p)}$ and $(t, p) \notin F_{\mathcal{N}}$ implies $W_{\mathcal{N}}(t, p) = \text{empty}_{D_{\mathcal{N}}(p)}$ for $p \in P_{\mathcal{N}}$ and $t \in T_{\mathcal{N}}$.

Let \mathcal{N} and \mathcal{N}' be ANSs. An *ANS morphism* $H : \mathcal{N} \rightarrow \mathcal{N}'$ consists of a MES morphism $H_{\mathcal{E}} : \mathcal{E}_{\mathcal{N}} \rightarrow \mathcal{E}_{\mathcal{N}'}$ of the underlying MESs, a net morphism $H_N : N_{\mathcal{N}} \rightarrow N_{\mathcal{N}'}$, and a function $H_V^t : V_{\mathcal{N}}(t) \rightarrow V_{\mathcal{N}'}(t)$ for each $t \in T_{\mathcal{N}}$ such that $x \in V_{\mathcal{N}}(t)_k$ implies $H_V^t(x) \in V_{\mathcal{N}'}(t)_{H_{\mathcal{E}}(k)}$ for $k \in \text{Kind}_{\mathcal{E}_{\mathcal{N}}}$, and the following conditions are satisfied:

1. $H_{\mathcal{E}}(D_{\mathcal{N}}(p)) = D_{\mathcal{N}'}(H_N(p))$ for each $p \in P_{\mathcal{N}}$;
2. $\mathcal{E}_{\mathcal{N}'} \models \forall V_{\mathcal{N}'}(t) . H_{\mathcal{E}}^t(G_{\mathcal{N}}(t)) \Rightarrow G_{\mathcal{N}'}(H_N(t))$ for each $t \in T_{\mathcal{N}}$;
3. $\mathcal{E}_{\mathcal{N}'} \models \forall V_{\mathcal{N}'}(t) . H_{\mathcal{E}}^t(G_{\mathcal{N}}(t)) \Rightarrow$

$$W_{\mathcal{N}'}(p', t') = H_{\mathcal{E}}^t(W_{\mathcal{N}}(p_1, t)) \dots H_{\mathcal{E}}^t(W_{\mathcal{N}}(p_n, t))$$
 for all $p' \in P_{\mathcal{N}'}, t' \in T_{\mathcal{N}'}, t \in H_N^{-1}(t')$,
 and $\{p_1, \dots, p_n\} = H_N^{-1}(p') \cap \bullet t$ with distinct p_i ;

4. $\mathcal{E}_{\mathcal{N}'} \models \forall V_{\mathcal{N}'}(t) . H_{\mathcal{E}}^t(G_{\mathcal{N}}(t)) \Rightarrow$
 $W_{\mathcal{N}'}(t', p') = H_{\mathcal{E}}^t(W_{\mathcal{N}}(t, p_1)) \dots H_{\mathcal{E}}^t(W_{\mathcal{N}}(t, p_n))$
 for all $p' \in P_{\mathcal{N}'}$, $t' \in T_{\mathcal{N}'}$, $t \in H_{\mathcal{N}}^{-1}(t')$,
 and $\{p_1, \dots, p_n\} = H_{\mathcal{N}}^{-1}(p') \cap t^\bullet$ with distinct p_i ;

where $H_{\mathcal{E}}^t : \text{Trm}_{\mathcal{E}_{\mathcal{N}}}(V_{\mathcal{N}}(t)) \rightarrow \text{Trm}_{\mathcal{E}_{\mathcal{N}'}}(V_{\mathcal{N}'}(t))$ is the common extension of $H_{\mathcal{E}}$ and H_V^t to terms. We assume for the above definition that validity \models has been extended to first-order formulae in the standard way.

ANSs together with their morphisms form a category **ANS**.

A typical ANS admits several colored nets as models. Since we want to state our results for an arbitrary but fixed model we also consider interpreted ANSs, i.e. ANSs together with distinguished data models. We furthermore equip interpreted ANS with a notion of morphism that allows us to express simultaneous transformations at the level of the ANSs and at the level of the data models.

To this end, we first introduce interpreted MESs together with a general notion of morphism that reflects a transformation of the specification as well as a transformation of the algebras possibly associated with different specifications. An *interpreted MES* (\mathcal{E}, A) consists of a MES \mathcal{E} and a \mathcal{E} -algebra A . The category **IMES** of interpreted MES is given by the Grothendieck construction $\Sigma(\mathbf{Mod})$ where $\mathbf{Mod} : \mathbf{MES} \rightarrow \mathbf{Cat}^{\text{op}}$. Recall that a morphism $(H, h) : (\mathcal{E}, A) \rightarrow (\mathcal{E}', A')$ in $\Sigma(\mathbf{Mod})$ consists of morphisms $H : \mathcal{E} \rightarrow \mathcal{E}'$ and $h : A \rightarrow \mathbf{U}_H(A')$ satisfying the conditions of the Grothendieck construction [TBG91].

Definition 3.2.5 An *interpreted ANS* (\mathcal{N}, A) consists of an ANS \mathcal{N} and a $\mathcal{E}_{\mathcal{N}}$ -algebra A . An *interpreted ANS morphism* $(H, h) : (\mathcal{N}, A) \rightarrow (\mathcal{N}', A')$ consists of an ANS morphism $H : \mathcal{N} \rightarrow \mathcal{N}'$ and an interpreted MES morphism $(H_{\mathcal{E}}, h) : (\mathcal{E}_{\mathcal{N}}, A) \rightarrow (\mathcal{E}_{\mathcal{N}'}, A')$. Interpreted ANSs together with their morphisms form a category **IANS**.

Interpreted ANSs are considerably richer than CNs, since they contain their specification together with a model equipped with a corresponding algebraic structure. In this sense they are similar to concrete predicate/transition nets [GL79, GL81, Gen91] and algebraic high-level nets [EPR94]. In fact, interpreted ANS, concrete predicate/transition nets [Gen90], and algebraic high-level nets [EPR94] can be regarded as instances of a general notion of *interpreted CNSs*.⁷ The transition from interpreted ANSs to CNs can be described by a forgetful functor as follows.

Definition 3.2.6 Given an interpreted ANS (\mathcal{N}, A) , the *CN semantics* of (\mathcal{N}, A) is given by the CN $\mathbf{CN}(\mathcal{N}, A)$ defined as follows:

⁷To be precise, arc inscriptions have to be restricted, since flexible arcs are not available in predicate/transition nets and algebraic high-level nets.

1. the underlying net $N_{\mathbf{CN}(\mathcal{N},A)}$ is precisely $N_{\mathcal{N}}$;
2. the color function $C_{\mathbf{CN}(\mathcal{N},A)}$ is defined by

$$C_{\mathbf{CN}(\mathcal{N},A)}(p) = \llbracket D_{\mathcal{N}}(p) \rrbracket_A$$
 for $p \in P_{\mathcal{N}}$ and

$$C_{\mathbf{CN}(\mathcal{N},A)}(t) = B_{\mathcal{N},A}(t)$$
 for $t \in T_{\mathcal{N}}$,
 where $B_{\mathcal{N},A}(t)$ is the set of *valid bindings* of $t \in T_{\mathcal{N}}$, i.e. the set of assignments $\beta : V_{\mathcal{N}}(t) \rightarrow A$ satisfying $G_{\mathcal{N}}(t)$;
3. the set of color sets $CS_{\mathbf{CN}(\mathcal{N},A)}$ is the smallest set that contains all $C_{\mathbf{CN}(\mathcal{N},A)}(x)$ for $x \in P_{\mathcal{N}} \cup T_{\mathcal{N}}$; and
4. the arc inscription $W_{\mathbf{CN}(\mathcal{N},A)}$ is defined by

$$W_{\mathbf{CN}(\mathcal{N},A)}(p, t)(\beta) = \llbracket W_{\mathcal{N}}(p, t) \rrbracket_{A,\beta}$$
 and

$$W_{\mathbf{CN}(\mathcal{N},A)}(t, p)(\beta) = \llbracket W_{\mathcal{N}}(t, p) \rrbracket_{A,\beta}$$
 for $p \in P_{\mathcal{N}}$, $t \in T_{\mathcal{N}}$ and assignments $\beta : V_{\mathcal{N}}(t) \rightarrow A$.

CN is extended to a functor $\mathbf{CN} : \mathbf{IANS} \rightarrow \mathbf{CN}$ that maps each morphism $(H, h) : (\mathcal{N}, A) \rightarrow (\mathcal{N}', A')$ to the morphism $G : \mathbf{CN}(\mathcal{N}, A) \rightarrow \mathbf{CN}(\mathcal{N}', A')$ satisfying $G_N = H_N$ and $G_x = h_{D_{\mathcal{N}}(x)}$ for $x \in P_{\mathcal{N}} \cup T_{\mathcal{N}}$.

We lift the flattening functor $(-)^{\flat} : \mathbf{CN} \rightarrow \mathbf{PTN}$ to interpreted ANS, denoting also by $(-)^{\flat} : \mathbf{IANS} \rightarrow \mathbf{PTN}$ the composition $(-)^{\flat} \circ \mathbf{CN}$. Using flattening we furthermore lift $\mathbf{BDP} : \mathbf{PTN} \rightarrow \mathbf{SMC}$ by defining $\mathbf{BDP} : \mathbf{IANS} \rightarrow \mathbf{SMC}$ as $\mathbf{BDP} \circ (-)^{\flat}$.

3.2.3 A Case Study

In the following we generalize the rewriting semantics from PTNs to ANSs. Before dealing with the general case we try to convey the main ideas using a distributed network algorithm as a running example, and we show how the rewriting semantics is obtained in this particular but typical case.

An algorithm which admits a very natural presentation as an algebraic net specification is the well-known echo algorithm, also called PIF algorithm (where PIF stands for “propagation of information with feedback”). The algebraic net model we use here has been developed and verified in [KRVW97].

Given a network of agents with bidirectional channels, the echo problem can be informally described as follows. A distinguished agent initiates the transmission of a piece of information which should be propagated (possibly using other agents) to all agents participating in the network. After that the initiator should receive feedback about the successful completion of this task, i.e., that each agent has received the information transmitted.

A possible solution to this problem is modeled by the algebraic net specification described below. To focus on the algorithm itself, the model abstracts from the

concrete information that is transmitted. This information can be easily added by refining the messages without major changes to the algorithm.

We assume that the agents can be distinguished in terms of their identifiers, which are modeled by a sort `Id`. The network of agents is represented as a directed multigraph, i.e. as a finite multiset of (directed) channels, where each channel is a pair of agent identifiers. In the specification fragment below, `Pair` is the sort of pairs of identifiers and `FMS-Id` is the sort of finite multisets over such pairs. Finite multisets are equationally axiomatized as discussed before. The obvious freeness constraints for `FMS-Id`, `Pair`, and `FMS-Id` can be specified using (parameterized) functional modules in Maude [CDE⁺99a, CDE⁺00b], but for the sake of brevity we omit the details here.

```
sort Id FMS-Id Pair FMS-Id .

op (_,_) : Id Id -> Pair .

op empty-Id : -> FMS-Id .
op single : Id -> FMS-Id .
op __ : FMS-Id FMS-Id -> FMS-Id [assoc comm id: empty-Id] .

op empty-Pair : -> FMS-Id .
op single : Pair -> FMS-Id .
op __ : FMS-Id FMS-Id -> FMS-Id [assoc comm id: empty-Pair] .

var x y x' y' : Id .
var fmsp fmsp' : FMS-Id .
var p p' : Pair .
```

To work with a concrete example we assume agent identifiers and a network as specified below. Actually, the algorithm is parametric in the choice of agent identifiers and in the network topology, the only assumptions being that there is a distinguished initiator and that the network is a strongly connected network with bidirectional channels. Again this parameterization could be made explicit in Maude by viewing the entire specification as a parameterized module which can be instantiated, for instance, by the following choices for `Id` and `network`.⁸ Again, a suitable initiality constraint should be added for `Id` to express that we have given complete enumeration of all its distinct elements.

```
ops i a b c d e : -> Id .
```

⁸If we were interested in (abstract) formal verification rather than (concrete) execution we would leave open the interpretation of `Id` and `network` and in this way obtain an ANS admitting a rich variety of quite different models (cf. the abstract OCC specification in Section 8.3.5).

```

op  sym : Id Id -> FMS-Pair .
eq  sym(x,y) = ((x,y) (y,x)) .

op  network : -> FMS-Pair .
eq  network = (sym(i,a) sym(i,b) sym(e,b) sym(e,d)
              sym(c,d) sym(c,i) sym(c,a) sym(a,b)) .

```

Now we equationally specify three auxiliary functions operating on finite multisets of pairs. The first one `_-` removes one occurrence of a given pair from a multiset of pairs. The other functions `out` and `in` will be used with `network` as a first argument: `out(network,x)` denotes the multiset of messages to be sent to neighbours of `x` and, correspondingly, `in(network,x)` denotes the multiset of messages to be received from neighbours of `x`. We should point out here that pairs are used for different purposes in the specification: As in the network above a channel directed from `x` to `y` is written as `(x,y)`, but on the other hand we keep the convention of [KRVW97] that a message directed from `x` to `y` is written as `(y,x)`.

```

op  _- : FMS-Pair Pair -> FMS-Pair .
eq  empty-Pair - p = empty-Pair .
eq  (p fmsp) - p = fmsp .
ceq (p' fmsp) - p = (p' (fmsp - p)) if p /= p' .

op  in : FMS-Pair Id -> FMS-Pair .
eq  in(empty-Pair,y') = empty-Pair .
eq  in(((x,y) fmsp),y) = ((y,x) in(fmsp,y)) .
ceq in(((x,y) fmsp),y') = in(fmsp,y') if y /= y' .

op  out : FMS-Pair Id -> FMS-Pair .
eq  out(empty-Pair,x') = empty-Pair .
eq  out(((x,y) fmsp),x) = ((y,x) out(fmsp,x)) .
ceq out(((x,y) fmsp),x') = out(fmsp,x') if x /= x' .

```

This concludes the MES. We are now ready to define the ANS on top of it. Its inscribed net is depicted in Figure 3.4. In the center we have a message pool **MESSAGES** modeling messages in transit. The net elements at the top model the activity of the initiating agent `i`, which is initially in a state **QUIET**, whereas the net elements at the bottom model the activities of all the remaining agents which are initially **UNINFORMED**. More precisely, the activities of initiators and noninitiators are the following:

- After the initiator i sends out a message to all its neighbours (transition **ISEND**) it will remain in the **WAITING** state until it receives an acknowledgement message from all its neighbours. If this happens, it will go into the **TERMINATED** state (transition **IRECEIVE**), i.e., the initiator has locally detected that all agents have received a message.
- After a noninitiator x receives a message from an agent y , it sends messages to all its neighbours except for y (transition **SEND**), and goes into a **PENDING** state, where it remembers that it is pending after receiving a message from y . As soon it receives acknowledgement messages from all neighbours except for y it goes into the **ACCEPTED** state (transition **RECEIVE**).

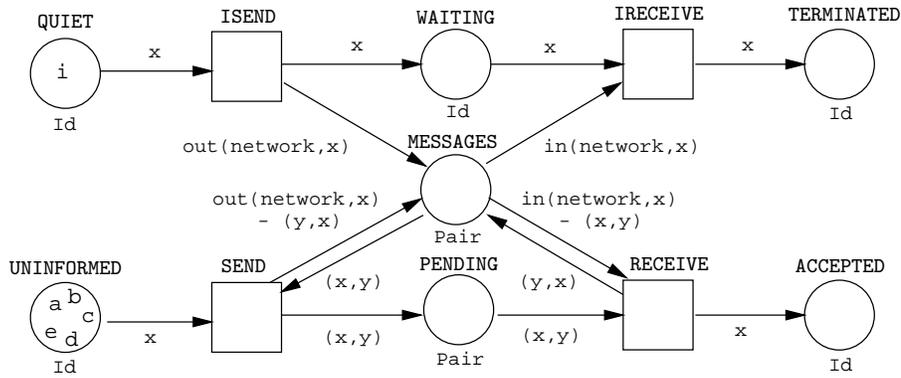


Figure 3.4: Echo Algorithm

The initial marking specification m_0 for our concrete choice of the network is given by the terms inside places. It is

$$\begin{aligned}
 m_0(\text{QUIET}) &= i & m_0(\text{WAITING}) &= \text{empty-Id} \\
 m_0(\text{UNINFORMED}) &= a \ b \ c \ d \ e & m_0(\text{MESSAGES}) &= \text{empty-Pair} \\
 m_0(\text{TERMINATED}) &= \text{empty-Id} & m_0(\text{PENDING}) &= \text{empty-Pair} \\
 m_0(\text{ACCEPTED}) &= \text{empty-Id} & &
 \end{aligned}$$

In Section 3.1 we have already discussed a rewriting semantics for the PTN of the banker’s problem. Using the echo algorithm we will demonstrate how the rewriting semantics generalizes to ANSs. It is worth mentioning that our semantics is designed to cope with flexible arcs as the ones connected with the place **MESSAGES** in the echo algorithm.

The rewriting semantics associated to a ANS extends but does not modify the underlying MES of the net, the advantage being that properties established for the equational logic specification are preserved and their proofs remain valid.

As in the PTN case, we represent a marking as an element of the kind **[Marking]**, which is equipped with a monoidal structure via the operations **empty** and **...**. For

each place p we have a *token constructor*, also written as p , representing the fact that a single token resides in place p . A difference with respect to the PTN rewriting semantics is that tokens carry data, which is reflected in the fact that token constructors are functions instead of being constants. For instance, a token `MESSAGES(msg)` represents a token carrying the data msg residing in the place `MESSAGES`. So the token constructor can be seen as a function *tagging* a data object with information about the place in which it is currently located.

```
sort Marking .
```

```
op empty : -> Marking .
op _ : Marking Marking -> Marking
  [assoc comm id: empty] .
```

```
ops MESSAGES PENDING : Pair -> Marking .
ops QUIET WAITING TERMINATED UNINFORMED ACCEPTED : Id -> Marking .
```

When formulating the transition rule for `ISEND` we are faced with the problem of how to translate the flexible arc between `ISEND` and `MESSAGES` appropriately. We would like to express that the multiset `out(network, x)` is added to the place `MESSAGES`, but this presupposes an interpretation of places as containers of objects which is different from our current one, where tokens are tagged objects “mixed up in a soup together with other tokens.”

An elegant solution is the linear extension of `MESSAGES` to multisets. For this purpose we *generalize* the token constructor `MESSAGES` which has been declared above to

```
op MESSAGES : FMS-Pair -> Marking .
```

and we add two equations expressing linearity of `MESSAGES`, which will also be called *place linearity equations*:

```
seq MESSAGES(empty-Pair) = empty .
seq MESSAGES(fmsp fmsp') = MESSAGES(fmsp) MESSAGES(fmsp') .
```

The place linearity equations express the equivalence of different ways of looking at the same marking of an ANS. So, as indicated by the keyword `seq`, from a high-level specification point of view it is reasonable to assign them to the class of structural equations expressing symmetries of the state representation. For reasons of uniformity we generalize the remaining token constructors correspondingly and we impose corresponding place linearity equations that we omit here.

Now the translation of transitions into rewrite rules can be done in full analogy with the rewriting semantics for PTNs. Each transition is represented as a rewrite

rule, also called a *transition rule*, replacing its preset marking by its postset marking. If the transition has a guard, then that guard becomes a condition of the rewrite rule. In this way we obtain the following rules:

```

r1  [ISEND]:    QUIET(x) =>
                WAITING(x) MESSAGES(out(network,x)) .

r1  [IRECEIVE]: WAITING(x) MESSAGES(in(network,x)) =>
                TERMINATED(x) .

r1  [SEND]:     UNINFORMED(x) MESSAGES((x,y)) =>
                PENDING((x,y)) MESSAGES(out(network,x)-(y,x)) .

r1  [RECEIVE]:  PENDING((x,y)) MESSAGES(in(network,x)-(x,y)) =>
                ACCEPTED(x) MESSAGES((y,x)) .

```

According to our initial explanation, a place can be seen as the tag of an object which indicates the place the token resides in. This is what we call the *tagged-object view*. The place linearity equations suggest a complementary view which is encountered more often in the context of Petri nets: a place is simply a container of objects. We call this the *place-as-container view*. The place linearity equations express our intention to consider both views as equivalent.

3.2.4 Rewriting Semantics in the General Case

Generalizing the above example, we now define for an arbitrary ANS its associated rewriting semantics. We also show in which sense the rewriting semantics is equivalent to the Best-Devillers process semantics of ANSs, which we have defined by lifting the Best-Devillers process semantics of PTNs to ANSs via the flattening construction. First we generalize symmetric monoidal RWSs (SMRWSs) to extended symmetric monoidal RWSs (ESMRWSs), which will serve as a suitable domain for the rewriting semantics. Notice that in ESMRWSs the data subspecification is not required to be empty. A second difference w.r.t. SMRWSs is that token constructors are extended to multisets and place linearity equations are added.

Definition 3.2.7 A RWS \mathcal{R} is an *extended symmetric monoidal RWS (ESMRWS)* iff the following conditions are satisfied:

1. $\mathcal{E}_{\mathcal{R}}$ extends $\mathcal{E}_{\mathcal{R}}^D$ precisely by:

- (a) a new kind [Marking] and new operator symbols

$$\text{empty} : [\text{Marking}], \quad _ : [\text{Marking}] [\text{Marking}] \rightarrow [\text{Marking}];$$

- (b) any number of new operator symbols of the general form

$$p : [\mathbf{FMS}_k] \rightarrow [\mathbf{Marking}],$$

where k is a kind in $\mathcal{E}_{\mathcal{R}}^D$ such that $\mathcal{E}_{\mathcal{R}}^D$ includes a MES of finite multisets over k ;

- (c) the axioms for parallel composition

$$\forall u, v, w : [\mathbf{Marking}] . u (v w) = (u v) w,$$

$$\forall u, v : [\mathbf{Marking}] . u v = v u,$$

$$\forall u : [\mathbf{Marking}] . \mathbf{empty} u = u; \text{ and}$$

- (d) the
- place linearity equations*

$$p(\mathbf{empty}_k) = \mathbf{empty},$$

$$\forall a, b : [\mathbf{FMS}_k] . p(a b) = p(a) p(b)$$

for each operator $p : [\mathbf{FMS}_k] \rightarrow [\mathbf{Marking}]$ introduced above.

2. Rules in
- $R_{\mathcal{R}}$
- contain only variables with kinds in
- $\mathcal{E}_{\mathcal{R}}^D$
- and have
- $\mathcal{E}_{\mathcal{R}}^D$
- conditions.

Given two ESMRWSs \mathcal{R} and \mathcal{R}' , an ESMRWS morphism $H : \mathcal{R} \rightarrow \mathcal{R}'$ is a RWS morphism that preserves $[\mathbf{Marking}]$, \mathbf{empty} and \dots . ESMRWSs together with their morphisms form a subcategory of **RWS** that is denoted by **ESMRWS**.

As in the case of SMRWSs, we specialize the approach of Section 2.4.5 to obtain the free model-theoretic semantics of an ESMRWS \mathcal{R} via a map \mathbf{E} into MEL. Again we have exploited the specific structure of ESMRWSs, namely that there is only a single rewrite kind $[\mathbf{Marking}]$ and that rewrites do never occur below other rewrites due to the form of the rules, to simplify the general construction of $\mathbf{E}(\mathcal{R})$.

Definition 3.2.8 The *membership equational presentation* of an ESMRWS \mathcal{R} is a MES $\mathbf{E}(\mathcal{R})$ that extends $\mathcal{E}_{\mathcal{R}}$, the underlying MES of \mathcal{R} , as explained in Definition 3.1.11, but modifying items 2 and 4 as follows:

2. a new operator symbol called
- basic proof constructor*

$$t : \bar{k} \rightarrow [\mathbf{RawProc}],$$

for each rule $\forall X . t : M \rightarrow N$ **if** $\bar{\phi}_1 \wedge \dots \wedge \bar{\phi}_n$ in $R_{\mathcal{R}}$;

4. a membership axiom

$$\forall X . t(\bar{x}) : M \rightarrow N \text{ **if** } \bar{\phi}_1 \wedge \dots \wedge \bar{\phi}_n$$

for each rule $\forall X . t : M \rightarrow N$ if $\bar{\phi}_1 \wedge \dots \wedge \bar{\phi}_n$ in $R_{\mathcal{R}}$, assuming that $X = \bar{x} : \bar{k}$. As in Definition 3.1.11, \mathbf{E} can be extended to a functor $\mathbf{E} : \mathbf{ESMRWS} \rightarrow \mathbf{MES}$ in the obvious way.

Definition 3.2.9 An *interpreted ESMRWS* (\mathcal{R}, A) consists of an ESMRWS \mathcal{R} and a $\mathcal{E}_{\mathcal{R}}^D$ -model A . An interpreted ESMRWS morphism $(H, h) : (\mathcal{R}, A) \rightarrow (\mathcal{R}', A')$ consists of an ESMRWS morphism $H : \mathcal{R} \rightarrow \mathcal{R}'$ and an interpreted MES morphism $(H_D, h) : (\mathcal{E}_{\mathcal{R}}^D, A) \rightarrow (\mathcal{E}_{\mathcal{R}'}^D, A')$. Interpreted ESMRWSs together with their morphisms form a category **IESMRWS**.

Definition 3.2.10 For an interpreted ESMRWS (\mathcal{R}, A) we define $\mathbf{Mod}(\mathcal{R}, A)$ as the subcategory of $\mathbf{Mod}(\mathcal{R})$ (i.e. $\mathbf{Mod}(\mathbf{E}(\mathcal{R}))$), with objects being \mathcal{R} -algebras (i.e. $\mathbf{E}(\mathcal{R})$ -algebras) \hat{A} satisfying $\hat{A}|_{\mathcal{E}_{\mathcal{R}}^D} = A$. In fact, this gives rise to a functor $\mathbf{Mod} : \mathbf{IESMRWS} \rightarrow \mathbf{Cat}^{\text{op}}$, and again we write \mathbf{U}_H for $\mathbf{Mod}(H)$ given a ESMRWS morphism H .

To simplify the exposition we assume as before that, given an interpreted ESMRWS (\mathcal{R}, A) and the obvious inclusion $K : \mathcal{E}_{\mathcal{R}}^D \hookrightarrow \mathbf{E}(\mathcal{R})$, the free functor \mathbf{F}_K has been defined in such a way that $\eta_K(A)$ becomes the identity and therefore $\mathbf{U}_K(\mathbf{F}_K(A)) = A$. The subsequent adaption of Lemma 2.4.3 to interpreted ESMRWS ensures that this is possible without loss of generality.

Lemma 3.2.11 (Protection Lemma) Let (\mathcal{R}, A) be an interpreted ESMRWS and consider the obvious inclusion $K : \mathcal{E}_{\mathcal{R}}^D \hookrightarrow \mathbf{E}(\mathcal{R})$. Then $\eta_K(A) : A \rightarrow \mathbf{U}_K(\mathbf{F}_K(A))$ is an isomorphism.

$$\begin{array}{ccc}
 \mathcal{E}_{\mathcal{R}}^D & \xrightarrow{K} & \mathbf{E}(\mathcal{R}) \\
 \mathbf{E}(H_D) \downarrow & & \downarrow \mathbf{E}(H) \\
 \mathcal{E}_{\mathcal{R}'}^D & \xrightarrow{K'} & \mathbf{E}(\mathcal{R}')
 \end{array}
 \qquad
 \begin{array}{ccc}
 \mathbf{Mod}(\mathcal{E}_{\mathcal{R}}^D) & \xrightleftharpoons[\mathbf{U}_K]{\mathbf{F}_K} & \mathbf{Mod}(\mathcal{R}) \\
 \mathbf{U}_{H_D} \uparrow & & \uparrow \mathbf{U}_H \\
 \mathbf{Mod}(\mathcal{E}_{\mathcal{R}'}^D) & \xrightleftharpoons[\mathbf{U}_{K'}]{\mathbf{F}_{K'}} & \mathbf{Mod}(\mathcal{R}')
 \end{array}$$

Figure 3.5: Morphisms in Definition 3.2.12

Definition 3.2.12 Let $\Sigma(\mathbf{Mod})$ be the Grothendieck construction for the functor $\mathbf{Mod} : \mathbf{IESMRWS} \rightarrow \mathbf{Cat}^{\text{op}}$ and let $\pi_1 : \Sigma(\mathbf{Mod}) \rightarrow \mathbf{IESMRWS}$ be the obvious projection functor that sends $((\mathcal{R}, A), \hat{A})$ to (\mathcal{R}, A) . Furthermore, let (\mathcal{R}, A) and (\mathcal{R}', A') be interpreted ESMRWSs and let $K : \mathcal{E}_{\mathcal{R}}^D \hookrightarrow \mathbf{E}(\mathcal{R})$ and $K' : \mathcal{E}_{\mathcal{R}'}^D \hookrightarrow \mathbf{E}(\mathcal{R}')$ be the obvious inclusions (cf. Figure 3.5). We then define $\mathbf{F}(\mathcal{R}, A)$ as $\mathbf{F}_K(A)$ and $\Sigma\mathbf{F}(\mathcal{R}, A)$ as $((\mathcal{R}, A), \mathbf{F}(\mathcal{R}, A))$. Given an interpreted ESMRWS

morphism $(H, h) : (\mathcal{R}, A) \rightarrow (\mathcal{R}', A')$ with $H : \mathcal{R} \rightarrow \mathcal{R}'$ and $h : A \rightarrow \mathbf{U}_{H_D}(A')$, we define $\Sigma\mathbf{F}(H, h)$ as the morphism $((H, h), \mathbf{F}(H) \circ \mathbf{F}_K(h)) : ((\mathcal{R}, A), \mathbf{F}_K(A)) \rightarrow ((\mathcal{R}', A'), \mathbf{F}_{K'}(A'))$ where $\mathbf{F}_K(h) : \mathbf{F}_K(A) \rightarrow \mathbf{F}_K(\mathbf{U}_{H_D}(A'))$ and $\mathbf{F}(H)$ is the unique morphism $\mathbf{F}(H) : \mathbf{F}_K(\mathbf{U}_{H_D}(A')) \rightarrow \mathbf{U}_H(\mathbf{F}_{K'}(A'))$ guaranteed by the fact that $\mathbf{F}_K(\mathbf{U}_{H_D}(A'))$ and $\mathbf{U}_H(\mathbf{F}_{K'}(A'))$ are objects in $\mathbf{Mod}(\mathcal{R}, \mathbf{U}_{H_D}(A'))$, since using Lemma 3.2.11 we find $\mathbf{U}_K(\mathbf{F}_K(\mathbf{U}_{H_D}(A'))) = \mathbf{U}_{H_D}(A')$ and $\mathbf{U}_K(\mathbf{U}_H(\mathbf{F}_{K'}(A'))) = \mathbf{U}_{H_D}(\mathbf{U}_{K'}(\mathbf{F}_{K'}(A'))) = \mathbf{U}_{H_D}(A')$, and by the fact that $\mathbf{F}_K(\mathbf{U}_{H_D}(A'))$ is initial. In this way we have defined a functor $\Sigma\mathbf{F} : \mathbf{IESMRWS} \rightarrow \Sigma(\mathbf{Mod})$ that is left adjoint to π_1 .

Furthermore, let $\mathbf{V} : \Sigma(\mathbf{Mod}) \rightarrow \mathbf{SMC}$ be the forgetful functor, which sends $((\mathcal{R}, A), \hat{A})$ to a SMC, defined as in Definition 3.1.12.

Definition 3.2.13 Given an ANS \mathcal{N} , the *rewriting semantics* of \mathcal{N} is the smallest ESMRWS $\mathbf{R}(\mathcal{N})$ with an underlying data specification $\mathcal{E}_{\mathbf{R}(\mathcal{N})}^D = \mathcal{E}_{\mathcal{N}}$ such that:

1. $\mathcal{E}_{\mathbf{R}(\mathcal{N})}$ contains a *token constructor*

$$p : [\mathbf{FMS}_{D_{\mathcal{N}}(p)}] \rightarrow [\mathbf{Marking}]$$

for each place $p \in P_{\mathcal{N}}$; and

2. $\mathbf{R}(\mathcal{N})$ has a label t and a rule called *transition rule*, namely,

$$\begin{aligned} \forall V_{\mathcal{N}}(t) . t : (p_1(W_{\mathcal{N}}(p_1, t)) \dots p_m(W_{\mathcal{N}}(p_m, t))) \rightarrow \\ (p_1(W_{\mathcal{N}}(t, p_1)) \dots p_m(W_{\mathcal{N}}(t, p_m))) \text{ if } G_{\mathcal{N}}(t) \end{aligned}$$

for each transition $t \in T_{\mathcal{N}}$, assuming $P_{\mathcal{N}} = \{p_1, \dots, p_m\}$ with distinct p_i .

\mathbf{R} can be extended to a functor $\mathbf{R} : \mathbf{ANS} \rightarrow \mathbf{ESMRWS}$ that maps each ANS morphism $H : \mathcal{N} \rightarrow \mathcal{N}'$ to the unique ESMRWS morphism $G : \mathbf{R}(\mathcal{N}) \rightarrow \mathbf{R}(\mathcal{N}')$ with $G_{\mathcal{E}}(p) = H_{\mathcal{N}}(p)$ for each $p \in P_{\mathcal{N}}$ and $G_L(t) = H_{\mathcal{N}}(t)$ for each $t \in T_{\mathcal{N}}$.

The functor $\mathbf{R} : \mathbf{ANS} \rightarrow \mathbf{ESMRWS}$ is naturally extended to a functor $\mathbf{R} : \mathbf{IANS} \rightarrow \mathbf{IESMRWS}$ sending each interpreted ANS (\mathcal{N}, A) to the interpreted ESMRWS $(\mathbf{R}(\mathcal{N}), A)$. Furthermore, \mathbf{R} sends each interpreted ANS morphism $(H, h) : (\mathcal{N}, A) \rightarrow (\mathcal{N}', A')$ to the interpreted ESMRWS morphism $(\mathbf{R}(H), h) : \mathbf{R}(\mathcal{N}, A) \rightarrow \mathbf{R}(\mathcal{N}', A')$.

Definition 3.2.14 Given an interpreted ESMRWS (\mathcal{R}, A) , we define the *flattening* of (\mathcal{R}, A) as the smallest SMRWS $(\mathcal{R}, A)^{\flat}$ satisfying the following conditions:

1. For each operator $p : [\mathbf{FMS}_k] \rightarrow [\mathbf{Marking}]$ in $\mathcal{E}_{\mathcal{R}}$ and for each $a \in \llbracket k \rrbracket_A$ there is a constant $p^a : \rightarrow [\mathbf{Marking}]$ in $\mathcal{E}_{(\mathcal{R}, A)^{\flat}}$.

2. For each rule $\forall X . t : M \rightarrow N$ if $\bar{\phi}_1 \wedge \dots \wedge \bar{\phi}_n$ in $R_{\mathcal{R}}$ and for each assignment $\beta : X \rightarrow A$ with $A, \beta \models \bar{\phi}_1 \wedge \dots \wedge \bar{\phi}_n$ we define functions σ and σ_p for each operator p as above by

$$\begin{aligned} \sigma(\mathbf{empty}) &= \mathbf{empty}, \quad \sigma(p(M)) = \sigma_p(\llbracket M \rrbracket_{A,\beta}), \quad \sigma(M N) = \sigma(M) \sigma(N), \\ \sigma_p(\llbracket - \rrbracket(\llbracket \mathbf{single} \rrbracket(a_1), \dots, \llbracket \mathbf{single} \rrbracket(a_m))) &= p^{a_1} \dots p^{a_m} \end{aligned}$$

($\llbracket - \rrbracket$ is naturally extended to an arbitrary number of arguments)
and we add a rule

$$t^{\beta(\bar{x})} : \sigma(M) \rightarrow \sigma(N)$$

to $R_{(\mathcal{R},A)^b}$, assuming that $X = \bar{x} : \bar{k}$.

$(-)^b$ is extended to a functor $(-)^b : \mathbf{IESMRWS} \rightarrow \mathbf{SMRWS}$ as follows: $(-)^b$ sends each interpreted ESMRWS morphism $(H, h) : (\mathcal{R}, A) \rightarrow (\mathcal{R}', A')$ with $H : \mathcal{R} \rightarrow \mathcal{R}'$ and $h : A \rightarrow \mathbf{U}_{H_D}(A')$ to a SMRWS morphism $(H, h)^b : (\mathcal{R}, A)^b \rightarrow (\mathcal{R}', A')^b$ defined such that $(H, h)^b(p^a) = H_{\mathcal{E}}(p)^{h(a)}$ and $(H, h)^b(t^{\bar{a}}) = H_L(t)^{h(\bar{a})}$ for $\bar{a} = \beta(\bar{x})$ and all p, t, a, β, \bar{x} as above.

The theorem and the corollary below are stated in complete analogy to the corresponding results for PTNs. Indeed the former results can be seen as special cases of the latter via an inclusion $\iota : \mathbf{PTN} \rightarrow \mathbf{ANS}$ which is the counterpart of $\iota : \mathbf{PTN} \rightarrow \mathbf{CN}$ on the specification level. However, for the proof we exploit the opposite direction, namely that Theorem 3.2.17 can be reduced to Theorem 3.1.14 via the flattening constructions introduced earlier. This can be done by a combination of commutative diagrams using the following two lemmas.

The first lemma essentially states that the rewriting semantics is compatible with flattening. Notice the overloading of \mathbf{R} and $(-)^b$.

Lemma 3.2.15 There is a natural isomorphism $\sigma : (-)^b \circ \mathbf{R} \rightarrow \mathbf{R} \circ (-)^b$ between the functors $(-)^b \circ \mathbf{R} : \mathbf{IANS} \rightarrow \mathbf{SMRWS}$ (with $\mathbf{R} : \mathbf{IANS} \rightarrow \mathbf{IESMRWS}$ and $(-)^b : \mathbf{IESMRWS} \rightarrow \mathbf{SMRWS}$) and $\mathbf{R} \circ (-)^b : \mathbf{IANS} \rightarrow \mathbf{SMRWS}$ (with $(-)^b : \mathbf{IANS} \rightarrow \mathbf{PTN}$ and $\mathbf{R} : \mathbf{PTN} \rightarrow \mathbf{SMRWS}$).

Proof Sketch.

Given an arbitrary interpreted ANS (\mathcal{N}, A) , we define $\sigma(\mathcal{N}, A) : (\mathbf{R}(\mathcal{N}, A))^b \rightarrow \mathbf{R}((\mathcal{N}, A)^b)$ as the unique SMRWS morphism with

$$\begin{aligned} \sigma(\mathcal{N}, A)_{\mathcal{E}}(p^a) &= (p, a) \\ \sigma(\mathcal{N}, A)_L(t^{\bar{a}}) &= (t, \bar{a}) \end{aligned}$$

for $p \in P_{\mathcal{N}}$, $t \in T_{\mathcal{N}}$, $a \in \llbracket k \rrbracket_A$, $\bar{a} \in \llbracket \bar{k} \rrbracket$ with $k = D_{\mathcal{N}}(p)$ assuming that $V_{\mathcal{N}}(t) = \bar{x} : \bar{k}$. To verify that $\sigma(\mathcal{N}, A)$ is indeed a SMRWS morphism and even an isomorphism, observe that each valid instance of a transition of (\mathcal{N}, A) gives rise to a transition rule in $(\mathbf{R}(\mathcal{N}, A))^{\flat}$ and another transition rule in $\mathbf{R}((\mathcal{N}, A)^{\flat})$ which are equal up to the renaming given by $\sigma(\mathcal{N}, A)$.

To show that σ constitutes a natural isomorphism it is sufficient to verify naturality: Given an interpreted ANS morphism

$$(H, h) : (\mathcal{N}, A) \rightarrow (\mathcal{N}', A')$$

and morphisms

$$\begin{aligned} (\mathbf{R}(H, h))^{\flat} &: (\mathbf{R}(\mathcal{N}, A))^{\flat} \rightarrow (\mathbf{R}(\mathcal{N}', A'))^{\flat}, \\ \mathbf{R}((H, h)^{\flat}) &: \mathbf{R}((\mathcal{N}, A)^{\flat}) \rightarrow \mathbf{R}((\mathcal{N}', A')^{\flat}), \end{aligned}$$

we have to verify

$$\sigma(\mathcal{N}', A') \circ (\mathbf{R}(H, h))^{\flat} = \mathbf{R}((H, h)^{\flat}) \circ \sigma(\mathcal{N}, A).$$

Indeed, we have

$$\begin{aligned} \sigma(\mathcal{N}', A')((\mathbf{R}(H, h))^{\flat}(p^a)) &= \sigma(\mathcal{N}', A')(H_{\mathcal{E}}(p)^{h(a)}) = (H_{\mathcal{E}}(p), h(a)), \\ \mathbf{R}((H, h)^{\flat})(\sigma(\mathcal{N}, A)(p^a)) &= \mathbf{R}((H, h)^{\flat})(p, a) = (H_{\mathcal{E}}(p), h(a)). \end{aligned}$$

Similarly,

$$\begin{aligned} \sigma(\mathcal{N}', A')((\mathbf{R}(H, h))^{\flat}(t^{\bar{a}})) &= \sigma(\mathcal{N}', A')(H_L(t)^{h(\bar{a})}) = (H_L(t), h(\bar{a})), \\ \mathbf{R}((H, h)^{\flat})(\sigma(\mathcal{N}, A)(t^{\bar{a}})) &= \mathbf{R}((H, h)^{\flat})(t, \bar{a}) = (H_L(t), h(\bar{a})). \end{aligned}$$

□

The second lemma expresses that flattening preserves models at the level of abstraction given by SMCs.

Lemma 3.2.16 There is a natural isomorphism $\rho : \mathbf{V} \circ \Sigma \mathbf{F} \rightarrow \mathbf{V} \circ \Sigma \mathbf{I} \circ (-)^{\flat}$ between the functors $\mathbf{V} \circ \Sigma \mathbf{F} : \mathbf{IESMRWS} \rightarrow \mathbf{SMC}$ and $\mathbf{V} \circ \Sigma \mathbf{I} \circ (-)^{\flat} : \mathbf{IESMRWS} \rightarrow \mathbf{SMC}$ (with $(-)^{\flat} : \mathbf{IESMRWS} \rightarrow \mathbf{SMRWS}$ and $\mathbf{V} \circ \Sigma \mathbf{I} : \mathbf{SMRWS} \rightarrow \mathbf{SMC}$).

Proof Sketch.

Given an arbitrary interpreted ESMRWS (\mathcal{R}, A) , we define the SMC morphism $\rho(\mathcal{R}, A) : \mathbf{V}(\Sigma\mathbf{F}(\mathcal{R}, A)) \rightarrow \mathbf{V}(\Sigma\mathbf{I}((\mathcal{R}, A)^b))$ as follows:

$$\begin{aligned} \rho(\mathcal{R}, A)(\llbracket p \rrbracket_{\mathbf{F}_K(A)}(\llbracket \mathbf{single} \rrbracket_{\mathbf{F}_K(A)}(a)) &= \llbracket p^a \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)} \\ \rho(\mathcal{R}, A)(\llbracket \mathbf{empty} \rrbracket_{\mathbf{F}_K(A)}) &= \llbracket \mathbf{empty} \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)} \\ \rho(\mathcal{R}, A)(\llbracket - \rrbracket_{\mathbf{F}_K(A)}(m_1, m_2)) &= \llbracket - \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)}(\rho(\mathcal{R}, A)(m_1), \rho(\mathcal{R}, A)(m_2)) \\ \rho(\mathcal{R}, A)(\llbracket - : - \rightarrow - \rrbracket_{\mathbf{F}_K(A)}(P, m_1, m_2)) &= \\ &= \llbracket - : - \rightarrow - \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)}(\rho'(\mathcal{R}, A)(P), \rho(\mathcal{R}, A)(m_1), \rho(\mathcal{R}, A)(m_2)) \end{aligned}$$

where

$$\begin{aligned} \rho'(\mathcal{R}, A)(\llbracket t \rrbracket_{\mathbf{F}_K(A)}(\bar{a})) &= \llbracket t^{\bar{a}} \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)} \\ \rho'(\mathcal{R}, A)(\llbracket \mathbf{id} \rrbracket_{\mathbf{F}_K(A)}(m)) &= \llbracket \mathbf{id} \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)}(\rho(\mathcal{R}, A)(m)) \\ \rho'(\mathcal{R}, A)(\llbracket - \rrbracket_{\mathbf{F}_K(A)}(P_1, P_2)) &= \llbracket - \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)}(\rho'(\mathcal{R}, A)(P_1), \rho'(\mathcal{R}, A)(P_2)) \\ \rho'(\mathcal{R}, A)(\llbracket - ; - \rrbracket_{\mathbf{F}_K(A)}(P_1, P_2)) &= \llbracket - ; - \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)}(\rho'(\mathcal{R}, A)(P_1), \rho'(\mathcal{R}, A)(P_2)) \\ &\text{if } P_1 \text{ and } P_2 \text{ are composable} \end{aligned}$$

That $\rho(\mathcal{R}, A)$ is an isomorphism in **SMC** follows from the following observations:

1. $\rho(\mathcal{R}, A)$ establishes a bijection between objects in $\mathbf{V}(\Sigma\mathbf{F}(\mathcal{R}, A))$ of the form $\llbracket p \rrbracket(\llbracket \mathbf{single} \rrbracket(a))$ and objects in $\mathbf{V}(\Sigma\mathbf{I}((\mathcal{R}, A)^b))$ of the form $\llbracket p^a \rrbracket$.
2. It is easy to see from the definition above that $\rho(\mathcal{R}, A)$ respects the algebraic structure of **SMC** regarding objects.
3. (\mathcal{R}, A) and $(\mathcal{R}, A)^b$ impose the same closure properties on objects. Although objects in (\mathcal{R}, A) are generated by a richer signature, this difference is compensated by the place linearity equations. Using these equations each object in $\mathbf{V}(\Sigma\mathbf{F}(\mathcal{R}, A))$ can be written as

$$\begin{aligned} \llbracket - \rrbracket(\llbracket p_1 \rrbracket \llbracket \mathbf{single} \rrbracket(a_{1,1}), \dots, \llbracket p_1 \rrbracket \llbracket \mathbf{single} \rrbracket(a_{1,m}) \dots \\ \llbracket p_n \rrbracket \llbracket \mathbf{single} \rrbracket(a_{n,1}), \dots, \llbracket p_n \rrbracket \llbracket \mathbf{single} \rrbracket(a_{n,m})). \end{aligned}$$

which in $\mathbf{V}(\Sigma\mathbf{I}((\mathcal{R}, A)^b))$ corresponds to

$$\begin{aligned} \llbracket - \rrbracket(\llbracket p_1^{a_{1,1}} \rrbracket, \dots, \llbracket p_1^{a_{1,m}} \rrbracket \dots \\ \llbracket p_n^{a_{n,1}} \rrbracket, \dots, \llbracket p_n^{a_{n,m}} \rrbracket). \end{aligned}$$

4. (\mathcal{R}, A) and $(\mathcal{R}, A)^b$ impose the same identifications on objects. The only potential difference comes from the place linearity equations which are present in (\mathcal{R}, A) but not in $(\mathcal{R}, A)^b$. However, these equations do not impose any identifications on the objects of $\mathbf{V}(\Sigma\mathbf{F}(\mathcal{R}, A))$ which are of the form given above.

5. $\rho(\mathcal{R}, A)$ establishes a bijection between morphisms in $\mathbf{V}(\Sigma\mathbf{F}(\mathcal{R}, A))$ that are generated by a rule $\forall X . t : M \rightarrow N$ if $\bar{\phi}_1 \wedge \dots \wedge \bar{\phi}_n$ in (\mathcal{R}, A) and morphisms in $\mathbf{V}(\Sigma\mathbf{I}((\mathcal{R}, A)^b))$ that are generated by a rule $t^{\beta(\bar{x})} : \sigma(M) \rightarrow \sigma(N)$ in $(\mathcal{R}, A)^b$.
6. It is easy to see from the definition above that $\rho(\mathcal{R}, A)$ respects the algebraic structure of **SMC** regarding morphisms.
7. (\mathcal{R}, A) and $(\mathcal{R}, A)^b$ impose the same closure properties and identifications on morphisms (relative to objects which have been discussed before).

To show that ρ is a natural isomorphism it is sufficient to verify naturality: Given an interpreted ESMRWS morphism

$$(H, h) : (\mathcal{R}, A) \rightarrow (\mathcal{R}', A')$$

and morphisms

$$\begin{aligned} \mathbf{V}(\Sigma\mathbf{F}(H, h)) &: \mathbf{V}(\Sigma\mathbf{F}(\mathcal{R}, A)) \rightarrow \mathbf{V}(\Sigma\mathbf{F}(\mathcal{R}', A')), \\ \mathbf{V}(\Sigma\mathbf{I}((H, h)^b)) &: \mathbf{V}(\Sigma\mathbf{I}((\mathcal{R}, A)^b)) \rightarrow \mathbf{V}(\Sigma\mathbf{I}((\mathcal{R}', A')^b)), \end{aligned}$$

we have to verify

$$\rho(\mathcal{R}', A') \circ \mathbf{V}(\Sigma\mathbf{F}(H, h)) = \mathbf{V}(\Sigma\mathbf{I}((H, h)^b)) \circ \rho(\mathcal{R}, A).$$

Indeed, we have

$$\begin{aligned} &\rho(\mathcal{R}', A')(\mathbf{V}(\Sigma\mathbf{F}(H, h))(\llbracket p \rrbracket_{\mathbf{F}_K(A)}(\llbracket \mathbf{single} \rrbracket_{\mathbf{F}_K(A)}(a)))) \\ &= \rho(\mathcal{R}', A')(\mathbf{F}(H)(\mathbf{F}_K(h)(\llbracket p \rrbracket_{\mathbf{F}_K(A)}(\llbracket \mathbf{single} \rrbracket_{\mathbf{F}_K(A)}(a)))))) \\ &= \rho(\mathcal{R}', A')(\mathbf{F}(H)(\llbracket p \rrbracket_{\mathbf{F}_K(\mathbf{U}_{H_D}(A'))}(\llbracket \mathbf{single} \rrbracket_{\mathbf{F}_K(\mathbf{U}_{H_D}(A'))}(h(a)))))) \\ &= \rho(\mathcal{R}', A')(\llbracket p \rrbracket_{\mathbf{U}_H(\mathbf{F}_{K'}(A'))}(\llbracket \mathbf{single} \rrbracket_{\mathbf{U}_H(\mathbf{F}_{K'}(A'))}(h(a)))))) \\ &= \rho(\mathcal{R}', A')(\llbracket H(p) \rrbracket_{\mathbf{F}_{K'}(A')}(\llbracket \mathbf{single} \rrbracket_{\mathbf{F}_{K'}(A')}(h(a)))) = \llbracket H(p)^{h(a)} \rrbracket_{\mathbf{I}((\mathcal{R}', A')^b)}, \end{aligned}$$

and

$$\begin{aligned} &\mathbf{V}(\Sigma\mathbf{I}((H, h)^b))(\rho(\mathcal{R}, A)(\llbracket p \rrbracket_{\mathbf{F}_K(A)}(\llbracket \mathbf{single} \rrbracket_{\mathbf{F}_K(A)}(a)))) \\ &= \mathbf{V}(\Sigma\mathbf{I}((H, h)^b))(\llbracket p^a \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)}) \\ &= \mathbf{I}((H, h)^b)(\llbracket p^a \rrbracket_{\mathbf{I}((\mathcal{R}, A)^b)}) \\ &= \llbracket p^a \rrbracket_{\mathbf{U}_{(H, h)^b}(\mathbf{I}((\mathcal{R}', A')^b))} \\ &= \llbracket (H, h)^b(p^a) \rrbracket_{\mathbf{I}((\mathcal{R}', A')^b)} = \llbracket H(p)^{h(a)} \rrbracket_{\mathbf{I}((\mathcal{R}', A')^b)}. \end{aligned}$$

theory, but they seem to be sufficient in practice as witnessed by [Rei98b], which presents a methodology for modeling and verification of distributed algorithms based on a version of ANSs that only admits safe processes.

Another related issue, namely the gap between the individual token philosophy and the collective token philosophy which clearly exists at the level of PTNs seems to become less relevant at the level of CNs, because of the increase of expressivity. We argue that interpreting CNs under the collective token philosophy is not only simpler and less dependent on the structure of the state space but also sufficient in principle, since by a suitable transformation of the CN we can equip tokens with unique identities in such a way that each original process corresponds to a safe process of the resulting CN.⁹ As we discussed earlier, individual and collective token philosophies coincide for safe processes. Nonsafe processes of the resulting CN are not considered any more. In this sense, the individual token philosophy can be seen as a special case of the collective token philosophy. Beyond that it may well be adequate for certain applications to mix the individual and the collective token views in the same system model, and indeed this is possible with the approach that we propose, namely by adopting the collective token semantics as a framework semantics and equipping tokens with additional identity attributes whenever needed for modeling purposes. Indeed this view reveals that individual and collective tokens semantics are just two extreme levels of abstraction and there are many intermediate levels that can be covered in this way. A good example of a very similar experience giving support to this point of view is the work [MT99] on a partial order semantics for object-oriented systems that, although typical of the individual token philosophy, is shown to be isomorphic to the rewriting semantics typical of the collective token philosophy, thanks to the unique identities of objects and messages.

3.2.5 Execution of Algebraic Net Specifications

First of all we lift the notion of executability from rewriting logic to net specifications. We say that a net specification \mathcal{N} is *weakly/strongly executable* iff its rewriting semantics $\mathcal{R} = \mathbf{R}(\mathcal{N})$ is weakly/strongly executable. For the operational semantics to be complete under an exhaustive rewrite strategy, and hence useful for the execution of net specifications, we should furthermore require that $\mathcal{E}_{\mathcal{R}}$ is sort-decreasing, confluent, and terminating, and that \mathcal{R} is coherent.

To actually execute a specification it is necessary to have an implementation of

⁹One policy to maintain unique identities is to encode the local history, i.e. the information about all events in the past cone of a token, in the identity of the token itself, and to ensure locally that the identities of the tokens produced by a transition are distinct. Of course, it is easy to imagine interesting classes of nets, e.g. object-oriented versions of high-level nets, where tokens are already equipped with unique identities so that this transformation is not needed at all.

a matching algorithm for the structural equations used in the specification. A typical rewrite engine such as Maude supports matching modulo all combinations of the laws of associativity, commutativity, and identity (ACU) [CDE⁺99a, CDE⁺00b]. Since the place linearity equations belong to a class of equations that are typically not supported by standard rewrite engines, we distinguish in the following between direct execution using ACUL-matching (L stands for linearity) and an alternative approach, namely execution via ACU-matching, which makes use of a simple semantics-preserving translation that can achieve executability without structural linearity equations.

Direct Execution via ACUL-Matching

It is easy to verify that the underlying MES in our example is already executable when the place linearity equations are seen as structural equations, and it is furthermore sort-decreasing, confluent and terminating. Still the entire RWS is not coherent and, as a consequence, the execution of the net specification would be potentially incomplete.

A subterm of the form `in(network,x)` which occurs in the lefthand side of the rewrite axiom `IRECEIVE` can be reduced using the equations for `in`, so that the rewrite axiom is not applicable anymore. An obvious solution is to replace the arc inscription `in(network,x)` of the transition `IRECEIVE` by a variable `fmsmsg` and to add the guard `fmsmsg == in(network,x)` to this transition. A corresponding modification of the net specification has to be carried out for the transition `RECEIVE`. In the rewriting semantics these changes are reflected by the modified rewrite rules given below.

```
var fmsmsg : FMS-Message .

cr1 [IRECEIVE]: WAITING(x) MESSAGES(fmsmsg) =>
                TERMINATED(x)
  if fmsmsg == in(network,x) .

cr1 [RECEIVE]:  PENDING((x,y)) MESSAGES(fmsmsg) =>
                ACCEPTED(x) MESSAGES((y,x))
  if fmsmsg == in(network,x)-(x,y) .
```

After this simple semantics-preserving transformation the rewrite specification is indeed coherent and satisfies all our conditions. To execute the RWS it is sufficient to use rewriting modulo associativity, commutativity, identity, and linearity for the representation of markings.

Execution using ACU-Matching

We show in the following that, given an executable ANS such as the one we have just obtained, there is an alternative approach to net execution by regarding the place linearity equations as computational equations instead of as structural equations. Of course, from the viewpoint of the abstract algebraic semantics nothing will change. An immediate consequence is, however, that the net specification can be executed using a standard rewriting engine such as Maude, without the need for a new matching algorithm.

The first step is to regard the place linearity equations as reduction rules, i.e.

```

eq  MESSAGES(empty-Pair) = empty .
ceq MESSAGES(fmsp fmsp') = (MESSAGES(fmsp) MESSAGES(fmsp'))
    if fmsp /= empty-Pair and fmsp' /= empty-Pair .

```

After applying this modification to all place linearity equations, the reduction rules are terminating (the condition avoids potential nonterminating computations) and confluent.

However, as a consequence of the use of place-linearity equations as reduction rules instead of as structural equations, the rewrite specification is not coherent anymore, because of the rules for `IRECEIVE` and `RECEIVE` and the new equations above. Again, we can carry out a simple semantics-preserving transformation by introducing a variable `mmsg` ranging over markings containing only tokens on `MESSAGES` and satisfying the equality condition `mmsg == MESSAGES(fmsmsg)`. By introducing the inverse `inv-MESSAGES` of `MESSAGES`, this condition becomes `inv-MESSAGES(mmsg) == fmsmsg`. Therefore, `inv-MESSAGES(mmsg)` gives us access to the flexible arc inscription `fmsmsg`. As a result we replace these two rules by the following, which make the specification coherent:

```

sort Empty MESSAGES-Marking Marking .
subsort Empty < MESSAGES-Marking < Marking .

var mmsg mmsg' : MESSAGES-Marking .

op  empty : -> Empty .
op  __ : Marking Marking -> Marking
    [assoc comm id: empty] .
op  __ : MESSAGES-Marking MESSAGES-Marking -> MESSAGES-Marking
    [assoc comm id: empty] .
op  __ : Empty Empty -> Empty
    [assoc comm id: empty] .

```

```

op MESSAGES : FMS-Pair -> MESSAGES-Marking .

op inv-MESSAGES : MESSAGES-Marking -> FMS-Pair .
eq inv-MESSAGES(empty) = empty-Pair .
eq inv-MESSAGES(MESSAGES(fmosp)) = fmosp .
ceq inv-MESSAGES(mmsg mmsg') =
  (inv-MESSAGES(mmsg) inv-MESSAGES(mmsg'))
  if mmsg /= empty and mmsg' /= empty .

cr1 [IRECEIVE]: WAITING(x) mmsg =>
  TERMINATED(x)
  if inv-MESSAGES(mmsg) == in(network,x) .

cr1 [RECEIVE]: PENDING((x,y)) mmsg =>
  ACCEPTED(x) MESSAGES((y,x))
  if inv-MESSAGES(mmsg) == in(network,x)-(x,y) .

```

It should be clear from this example how the general translation works. It takes the form of a conservative theory transformation from the original RWS of an ANS executable by ACUL matching to a logically equivalent RWS executable by ACU matching. The transformation can be applied to any executable ANS satisfying the mild condition that flexible arcs are inscribed by variables, as it is the case in the executable version of the echo algorithm.¹⁰

Even though the resulting RWS is strongly executable, a strategy to execute the specification or to partially explore the state space can be useful, because of the highly nondeterministic nature of the algorithm. A strategy of this kind can be seen as restricting the possible rewrites leading to a subcategory of the original transition category of all rewrites. If the RWS is only weakly executable, as in the example discussed in [Stec], the strategy can play an additional role, namely to find suitable instantiations for the variables that cannot be determined by matching.

3.2.6 Abstract Net Execution

In distributed network algorithms such as the echo algorithm treated in our case study, the dynamics depends heavily on the structure of the network. To execute the net specification we have specified a particular network graph and there seem to be no reasons why we should consider a semantics of the underlying specification of the net other than the initial one. On the other hand, if our objective is to verify the algorithm formally, it is highly desirable to develop proofs that are

¹⁰A more general transformation is possible if we use conditions with matching equations, a feature supported by the most recent version of Maude [CDE⁺00a].

independent of the concrete choice of the network topology but depend instead only on its abstract properties such as strong connectivity.

Since one of the main characteristics of a specification language, that distinguishes it from a programming language, is the admission of a typically rich class of models instead of only a single one, the power of the equational specification approach has not been fully exploited so far. We find it natural to generalize the usual notion of execution in the initial model to abstract execution w.r.t. a possibly infinite class of models. This more general notion has the advantage of covering the spectrum from concrete execution, where all details of the model are known, to abstract execution, where only limited knowledge is available. This notion of abstract execution is a natural generalization of the idea of symbolic simulation, which is a well-established verification technique in the context of hardware verification.¹¹ Symbolic simulation, which evolved from ternary logic simulation, is mostly concerned with boolean functions and synchronous designs, which can be naturally specified in a purely functional way. Abstract execution using rewriting logic, on the other hand, allows execution in a variety of equationally definable state spaces (the multiset representation of Petri nets being just one example) and favors a notion of asynchronous nonequational computation. In spite of its roots in hardware verification, it appears to us that the idea of abstract execution can be fruitfully applied in software or high-level system verification, which can partly be seen as an interplay between case analysis, structural induction, and abstract execution.

In addition to its relation to symbolic simulation, it is furthermore noteworthy that a form of abstract execution, called partial evaluation [JGS93], is well-known in the context of functional programming languages. Similar to abstract execution, partial evaluation proceeds as far as the currently available information (which is in this case given by some actual arguments) admits.¹² A difference to abstract execution, however, is that functional programming languages have a single well-defined model, whereas abstract execution in a specification language typically covers a rich class of models. Another difference is that we are concerned here with dynamic systems, meaning that abstract execution is not restricted to partial functional evaluation, but it can be applied to system models that include functional and nonfunctional parts.

Abstract execution does not require any changes to the approach introduced before. The key point is that the underlying specification of a net is usually less restrictive and admits a rich variety of models instead of just a single one. As a simple example, that clearly demonstrates the point of abstract execution, consider the net specification depicted in Figure 3.7 based on the following MES:

¹¹Bryant91,Bryant92,HaSe97,BoGeR000

¹²It might be interesting in this context that an approach similar to partial evaluation has also been used to optimize functional programs (representing layers of communication protocols) in the Nuprl proof assistant [Kre99].

```

sort Nat .

op 0 : -> Nat .
op suc : Nat -> Nat .
op _+_ : Nat Nat -> Nat [assoc comm].

var n m : Nat .

eq 0 + m = m .
eq suc(n) + m = suc(n + m) .

sorts Even Odd .
subsort Even < Nat .
subsort Odd < Nat .

mb 0 : Even .
cmb suc(n) : Even if n : Odd .
cmb suc(n) : Odd if n : Even .

mb n + n : Even .

cmb n + m : Even if n : Even and m : Even .
cmb n + m : Even if n : Odd and m : Odd .
cmb n + m : Odd if n : Even and m : Odd .

sort FMS-Nat .

op empty-nat : -> FMS-Nat .
op single : Nat -> FMS-Nat .
op __ : FMS-Nat FMS-Nat -> FMS-Nat [assoc comm id: empty-nat] .

```

We add the following constraints: `Nat` is freely generated by `0` and `suc`. `Even` and `Odd` are freely generated by the first three (conditional) memberships. `FMS-Nat` is freely generated by its constructors and the structural equations specified. The last three memberships about `Even` and `Odd` are logical consequences of these assumptions that can be proved by induction.

Up to isomorphism the specification described so far has a single model that coincides with its initial model. Now we extend the specification to a specification that has a number of quite different models without admitting an initial model anymore.

We declare constants `a`, `b`, `c`, and `d` of sort `Nat` and the only assumptions are `b : Even` and `c : Odd`.

```
ops a b c d : -> Nat .
```

```
mb b : Even .
```

```
mb c : Odd .
```

Semantically, a , b , c , and d cannot generate new elements due to the initiality imposed on Nat before. They can only be interpreted as existing elements of this sort. As a consequence, imposing initiality on the last operator declaration would lead to a contradiction, i.e. to the empty class of models. So we do not impose any further constraints and let a , b , c , and d range over elements of Nat without any restriction. This leads to a rich class of models that can constitute the underlying models of our net specification depicted in Figure 3.7.

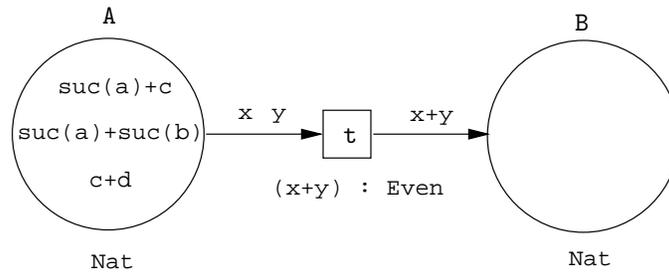


Figure 3.7: Example for Abstract Execution

The rewriting semantics is given by the following specification, where we have omitted place linearity equations for the token constructors A and B. They are not necessary, since flexible arcs are not used in the net.

```
var fmsn fmsn' : FMS-Nat .
```

```
sorts Marking .
```

```
var m : Marking .
```

```
op empty : -> Marking .
```

```
op __ : Marking Marking -> Marking [assoc comm id: empty] .
```

```
op A : FMS-Nat -> Marking .
```

```
op B : FMS-Nat -> Marking .
```

```
crl [t] : A(n) A(m) => B(n + m)
```

```
if n + m : Even .
```

Now the symbolic representation of the initial marking, i.e.,

$A((\text{suc}(a) + c)(\text{suc}(a) + \text{suc}(b))(c + d))$.

rewrites to the symbolic marking

$A(c + d) B(\text{suc}(\text{suc}(\text{suc}(a + a + b + c))))$

and, since we do not know anything about d and $c + d$, this is the only possible rewrite that is valid in all models.

To sum up, abstract execution has the capacity of generating executions that are valid in all models of the underlying specification. Depending on the application, abstract execution can be of value in quite different ways:

- In applications with a dynamic behavior that is data independent, abstract execution can cover the entire system behavior. The advantage over execution in the ordinary sense is that an infinite number of concrete executions are replaced by a single abstract execution.
- In applications where the dynamic behavior heavily depends on the concrete contents of data objects such as in the example given above, we cannot in general expect that abstract execution can be fully automated as in our simple example. Instead, the abstract executions generated automatically are typically of a partial nature and execution can proceed automatically as far as the computable knowledge about the current system state admits.

Even though in the second case fully automatic abstract execution of systems is impossible in principle, due to the lack of knowledge about the state of the system, we think that partial abstract execution is no less important because of this limitation. In a theorem proving environment these partial executions can be intertwined with (interactive) theorem proving steps such as case analysis or, more generally, structural induction, and hence partial executions can be concatenated to full executions under appropriate conditions. An approach that can provide a framework for this form of integration of abstract execution in the more general activity of formal theorem proving is the open calculus of constructions presented in Chapter 8 (see also Section 3.3.2 below).

Last but not least, it is important to point out that abstract execution is a capability of rewriting logic that can be applied to the kind of Petri nets used here but it should be applicable without changes to extensions and variations of Petri net models as well, and more generally to all formalisms that are represented using rewriting logic as a semantic framework.¹³

¹³For instance, some preliminary experience has been gained in the context of our specification of the active network programming language PLAN (see Appendix B and [ST02a]), where we are interested in properties of PLAN programs in a rich class of models generated by different network topologies.

3.3 Generalizations of Our Approach

We have demonstrated the capabilities of rewriting logic as a semantic framework by representing a very general form of algebraic net specifications and we have elaborated the idea of net execution by rewriting at different levels of abstraction. Our results and case studies suggest that rewriting logic offers a practical framework for the development of tools to execute and analyze different classes of Petri nets. In the present section, we would like to address another benefit of embedding Petri nets into a more general formalism, namely that rewriting logic provides a uniform formal framework for existing extensions of Petri nets and suggests a number of new extensions and generalizations of Petri nets. Altogether, we feel that rewriting logic could be a source of inspiration and a useful guide for future research in this subject.

3.3.1 Higher-Order Programming Languages

In view of our first-order treatment in this chapter, a natural question is how colored Petri nets based on higher-order programming paradigms can be formally represented. Paradigms of particular interest include higher-order functional programming, e.g. based on the λ -calculus, and object-oriented programming, e.g. based on Abadi and Cardelli's ζ -calculus [AC96].

Due to the fact that rewriting logic has good properties as a metalanguage it is possible to embed other higher-order languages into rewriting logic, once we have a general method to deal with the syntax of higher-order calculi in a first-order way. As a generic solution we have developed the CINNI calculus, a generic first-order calculus of names and explicit substitutions that will be introduced in Chapter 5. This calculus, together with the metatheoretic results that we provide, leads to a systematic way of representing higher-order programming languages in the first-order framework of membership equational logic and rewriting logic. As a consequence, all the results in the present chapter apply to the higher-order versions of colored Petri nets mentioned above, and other versions whose underlying programming languages can be represented following the approach of Chapter 5.

3.3.2 Higher-Order Logic and Types

We feel that the transition to a more expressive logic than the many-sorted equational logic used by algebraic approaches to Petri nets is a practical necessity if the idea of rigorous formal specification using Petri nets should be taken seriously. Already in [Rei98b] it becomes obvious that an equational language is not sufficient to give the specification of a number of distributed network algo-

rithms. This problem remains unsolved in [Rei98b] that uses an ad hoc extension of algebraic specifications by a form of first-order logic in an informal way. The combination of Petri nets with type theory and higher-order logic is a consequent next step which seems to be not only an advantage for the modeling and abstraction capabilities on the specification level, but it is also need to develop useful techniques and methods for rigorous formal verification of net specifications based on a strongly typed formalism.

Typically, a verification task involves proving a number of mathematical theorems of quite general nature. For instance, induction is an important reasoning principle needed in nearly all nontrivial verification tasks. This suggests using a logic such as higher-order logic that is expressive enough to formulate such principles internally. Furthermore, the underlying type theory should be expressive enough to allow for a natural formalization of the concepts needed for the verification task. The approach we propose in Chapter 8 is based on the open calculus of constructions (OCC), a type theory which allows us to reason formally about net models that are embedded into its higher-logic logic via their rewriting semantics. The higher-order logic of OCC is expressive enough to represent higher-order colored net specifications in a strongly typed and formally rigorous fashion that can be supported by computer tools. Indeed, OCC can be seen as a natural higher-order generalization of membership equational logic and rewriting logic, and has a similar objective, namely to integrate execution, analysis, and formal reasoning about system models in a simple and uniform framework.

3.3.3 Objects and Active Tokens

Another interesting generalization of Petri nets is related to different variants of *object Petri nets* [Val95, Val00, Val98, Far99], where tokens can themselves be nets with their own independent dynamic behaviour. Another related but different line of research is the integration of object-oriented techniques with Petri nets. As a result there are a number of variants of object-oriented Petri nets [Sib94, Lak95] where the tokens are objects according to standard object-oriented terminology. As a unifying generalization of both approaches we propose to study *active token nets*. In contrast to object Petri nets, tokens cannot only be nets, but arbitrary objects with an internal dynamic behavior. In contrast to *object-oriented approaches to Petri nets*, active tokens are not static but dynamic entities. In particular, they can evolve concurrently with the overall system behavior, and they can also interact or communicate with each other, similar to objects in the rewriting-logic approach to concurrent object-oriented programming [Mes93]. The autonomous activity of tokens and their communication indicate some similarity to agent-oriented programming approaches. This similarity is further supported by the fact that a net defines a topology in the abstract mathematical sense that provides an environment for group activities of agents.

It might appear that the complexity of such models is beyond the scope of a rigorous formal treatment. However, a closer look reveals that the approach to algebraic nets presented here is closer to the ideas described above as it might appear at first sight. In fact our approach to algebraic nets can be easily generalized to active token nets just by using rewrite rules more liberally. So far we have employed rewrite rules only to represent transitions of the net. In order to describe tokens with internal activity we could use rewrite rules that transform individual tokens. To capture group activity such as interaction (which corresponds to synchronous communication) and asynchronous communication between agents, we have to add rewrite rules that operate on a group of tokens. In other words, active token nets are defined on top of an underlying rewrite specification. Hence they can exploit the full modeling power of membership equational rewriting logic. This is the main difference to algebraic nets, where the underlying specification is restricted to membership equational logic which is only used to model static aspects of a system.

Finally, we would also like to mention that there is a complementary view of active token nets as structured rewrite specifications. An active token net can be seen as a particular way to structure a rewrite specification in a way that could also lead to a formal graphical representation of a certain class of well-structured rewriting logic specifications. So the Petri net view could not only be seen as a structuring principle for a certain class of rewrite specifications but it could also provide a bridge between symbolic presentations that are useful for logical reasoning and graphical presentations that seem to be more intuitive. From a more general point of view, such an approach could provide a formal link between rewriting logic and visual specification techniques, which have recently received a considerable degree of interest, especially after the development of the Unified Modeling Language [RJB98].

3.3.4 Structure of the State Space

Apart from generalizations of the underlying specification or programming language, which can itself be defined using rewriting logic as a metalanguage, there is another potential source of Petri net generalizations, namely the structure of the state space. Instead of considering a flat state space as in ordinary Petri nets, we could choose a hierarchical one or we could consider extensions such as *macroplaces* [Ani91, AKMP96], that can be seen as combining several places into a single one from the viewpoint of certain transitions. Also we could consider different kinds of places, for instance we could distinguish ordinary high-level places, that carry a multiset of tokens, from places, that are organized as a queue or as a stack. The former idea has already been studied in the literature in terms of *FIFO-nets* [FM82, KMT88, FC88, Fan92]. We conjecture that rewriting logic is a suitable formalism to represent and unify such variations of Petri nets since

the state space can be specified by an equational theory that is entirely user-definable. A related approach that allows some freedom in the choice of the state space algebra and specializes to different low-level and high-level Petri net classes is presented in [EP97]. The approach of rewriting logic has the advantage of being more general, in the sense that it goes beyond Petri-net like models and hence provides a bridge to formalisms that are quite different from ordinary Petri nets.

3.4 Final Remarks

In this chapter we have explained how rewriting logic can be used as a semantic framework in which a wide range of Petri net models can be naturally unified. Specifically, we have explored how place/transition nets, nets with test arcs, algebraic net specifications, and colored Petri nets can all be naturally expressed in rewriting logic, and how well-known semantic models often coincide with (in the sense of being naturally isomorphic to) the natural semantic models associated to the rewriting logic representations of the given nets. The classes of Petri nets that can be naturally represented in rewriting logic are by no means limited to the classes that we treated in detail. They also include the classes briefly sketched above and other interesting classes such as timed Petri nets as demonstrated in [SMÖ01b, SMÖ01a].

In our view, the unification of Petri net models within the rewriting logic framework is useful not only for conceptual reasons, but also for purposes of execution, formal analysis, and formal reasoning about Petri net specifications. Via the rewriting semantics, we can exploit existing languages and architectures in order to build tools supporting these tasks. For instance, using the reflective and metalinguage capabilities of Maude, it is possible to build execution environments for Petri net specifications where the model description provided by the user and the user interaction could all take place at the Petri net level with which the user is familiar.

As an important instance of a general notion of colored net specifications over a logic, we have treated ANSs over MEL in detail, and we have elaborated the idea of (abstract) net execution in this context. The definition of ANSs that we use is sufficiently general to cover logic-oriented approaches such as algebraic nets and more operational approaches such as colored Petri nets. In other words, we have shown that the specification-oriented approach and the programming-oriented approach can be unified in the single notion of ANSs, and we therefore think that ANSs are a useful concept to bridge the gap between execution and verification, which we consider as two extremes of a spectrum rather than as entirely different issues.

Strategy-guided execution of Petri net models is an major application of our rewriting semantics. It can be useful to locate flaws in the design or analysis

phase, but clearly it cannot replace the verification of a system with infinitely many possible executions. In some cases however, e.g. if the state space is finite, general strategies performing certain analysis tasks, like reachability and deadlock analysis or, more generally, temporal logic model checking, can be used for automatic verification of Petri nets with modest state-space complexity. In a more complex or more abstract scenario we often have to resort to formal verification of Petri nets, which is another important application of our rewriting semantics.

Apart from its operational aspects the rewriting semantics provides a logical representation of Petri nets and their processes which can be used in formal theorem proving environments. In our view, Petri net execution and analysis are very special cases of automated theorem proving. In the long term we propose that an integrated tool for verification and execution of Petri nets should be embedded into a general purpose theorem proving environment which allows reasoning about Petri nets via their rewriting semantics in an expressive logic with partial automation. The open calculus of constructions that we develop in Chapter 8 is a first step in this direction, as illustrated by the example in Section 8.2.8.

We have also discussed another benefit of embedding Petri nets into the more general framework of rewriting logic, namely that it may provide a uniform formal framework for existing extensions of Petri nets, and that it can inspire and guide new extensions and generalizations of Petri nets. In addition, it also facilitates integration of Petri nets with other paradigms which can also be given a rewriting semantics, e.g. object-oriented concurrent programming [Mes93]. Although we often emphasized and exploited the generality of rewriting logic, we feel that rewriting logic is very much in the spirit of Petri net theory in the sense that its essential features, namely monotonicity and locality of changes, are also the main features of rewriting logic. Therefore, we believe that rewriting logic is neither excessively general nor an arbitrary model of concurrency, but rather a consequent generalization of Petri nets within the borders of a single paradigm.

3.5 Acknowledgements

This chapter is based on my earlier work [Stec] conducted in the scope of the European Community project MATCH (CHRX-CT94-0452) and on subsequent joint work [SMÖ01b, SMÖ01a] with José Meseguer and Peter Ölveczky. Support by DARPA through Rome Laboratories Contract F30602-C-0312 and NASA through Contract NAS2-98073, by Office of Naval Research Contract N00014-99-C-0198, and by National Science Foundation Grant CCR-9900334 is gratefully acknowledged. I am also indebted to Manuel Clavel, Francisco Durán, and Steven Eker for their help during my initial experiments with net execution in Maude.

Furthermore, I would like to thank Narciso Martí-Oliet, Rüdiger Valk, and the referees of [Stec, SMÖ01b, SMÖ01a] for their constructive criticism and many helpful suggestions, and Roberto Bruni for his detailed feedback on [SMÖ01b]. Last but not least, I would like to express my appreciation to Amin Coja-Oghlan, who wrote his Studienarbeit [CO02] on the results of Section 3.1, for his careful analysis. An important contribution of his work, which appeared in [COS02], is a more direct and very detailed proof of Theorem 3.1.14 that is independent of [DMM96]. It is more direct, because it is uniformly conducted on the level of the collective token semantics, avoiding especially the construction of the intermediate category of processes $\mathcal{P}(\mathcal{N})$, which reflects the individual token semantics.

Chapter 4

Meta-Logical Reasoning in the Calculus of Inductive Constructions:

A Formalized Generalization of UNITY

In this chapter we present an approach to the specification and verification of concurrent programs and systems that is based on a logical type theory. Type theories have been applied successfully in functional programming and formal specification of verification of such programs. Here we develop an embedding of concurrent programs and systems together with a UNITY-style temporal logic into the calculus of inductive constructions (CIC). The embedding is not only very direct but also allows us to take advantage of essential features of the CIC, namely its expressive higher-order logic, its rich type system which includes dependent and inductive types, its computational features, its metatheoretic properties, and, last but not least, its capabilities to treat propositions as types and proofs as objects.

The UNITY methodology has been developed in the late 80s by Chandy and Misra [CM88]. Its formal techniques comprise a programming language and a temporal logic. Programs are viewed as *Unbounded Nondeterministic Iterative Transformations*, and the specification language is a variant of linear time temporal logic. Intended applications are specification, design and verification of imperative programs, parallel programs, concurrent systems and asynchronous hardware. Today, UNITY is a well-known methodology that has proven to provide a surprisingly elegant technique for (compositional) system verification in many case studies and still attracts the interest of researchers in (formal) program verification as witnessed for instance by the project [Pau99] and other recent references such as [CK97, Hey97, Che98, Pau00, Mis94, Mis95, CC99a, Gol99, Mis01]. Examples and case studies can be found in [CM88, Che98, Pau00, SLU90, Kna92, Sta93, CM90, SO89, Sta88, Mis90a, Gol92b]. Interesting languages closely related to UNITY are Seuss [Mis98, Mis99] and Mobile UNITY [RMP96, MR97, RMP97, PRM97, MR98, RM99].

We present a rigorous formal development of a generalization of the UNITY temporal logic in CIC. Although we demonstrate the use of the resulting temporal logic library by means of a simple example at the end of this chapter, the emphasis in this thesis is not on reasoning within the temporal logic but about it. Our main goal in this thesis is to explore the suitability of a logical type theory, namely CIC, for this abstract kind of metalogical reasoning. Reasoning within the UNITY-style temporal logic is the subject of [Stea] and [Steb]. In these references we show how the results of this chapter can be applied to verification of concrete systems modeled as place/transition nets and high-level Petri nets.

In contrast to most existing formalizations of UNITY we follow the more recent proposal for New UNITY [Mis94, Mis95], which is compatible with the original UNITY approach but develops the logic in terms of more elementary temporal operators. (New) UNITY suggests a number of interesting generalizations concerning the notion of program and the notion of fairness which provides a basis to deal with system specification languages quite different from the UNITY programming language such as Petri nets and rewriting logic, to name just two

examples. Indeed, it is this generalization of (New) UNITY which is the subject of the present chapter. On the formal side, our treatment benefits from the use of type theory in several ways. A particularly interesting result of the use of type theory is a new compositionality theorem for liveness assertions. We think it is no exaggeration to say the the formal elegance with which this result can be formulated and proved in CIC is made possible by the propositions-as-types and proofs-as objects interpretation. As far as we know this is a new application of this interpretation that has not been considered so far.

The motivation to develop an approach that is more general than UNITY comes from the observation that the UNITY temporal logic is to a large degree independent of the particularities of UNITY programs. The idea of adapting the UNITY temporal logic to other system models is not new: For instance, variations of the UNITY temporal logic have been applied to the verification of colored nets [MV91], [Mac91] and [Stea], and a related but quite different line of research based on partial order semantics started in [DGK⁺92] has led to a method called DAWN (Distributed Algorithms Working Notation) [WWV⁺97]. This method is applied to different kinds of distributed algorithms modeled as algebraic net specifications in [Rei98b]. Another approach that is inspired by UNITY is presented in [Sha93]. The references [Stea] and [Steb]¹ can be seen as an application of the results presented in this chapter to colored nets and hence also to algebraic net specifications as defined in Chapter 3.

In contrast to the formulation of a theory in conventional mathematical language we present a fully formalized development that has been carried out using a proof development system, namely the COQ proof assistant. All theorems have been mechanically checked and hence we can have a degree of confidence in their correctness that is much higher than our trust in mathematical theorems that have only informal proofs. In order to convey the development to the reader we have chosen a level of abstraction that presents formal theorems and important lemmas. Although it seems to be unavoidable to present the development in some detail, we have omitted many theorems and lemmas, and we have also refrained from presenting the formal proofs. We think that in order to understand the formalization in full detail it is unavoidable to explore the development including the proofs in an interactive session using COQ and to apply it to concrete examples. Also we would like to encourage researchers to apply or extend the temporal logic library. For these reasons the complete development has been made publicly available on the author's web page.²

¹This reference also contains a detailed verification of a distributed network algorithm which is given by an algebraic net specification presented in [Stec].

²This version supercedes an earlier version of the library described in [Ste98], which was essentially limited to UNITY programs, or more precisely to finitary functional transition systems with unconditional fairness.

4.1 A Sound and Complete Fragment of UNITY

We first give an informal view of UNITY that will be the starting point for the generalizations that we introduce in Section 4.2. Although we intentionally try to avoid major deviations from the original approach, there are subtle differences with respect to [CM88] in order to avoid well-known problems with formal consistency, an issue that we will briefly address below.

4.1.1 The Programming Language

In the following we give an informal and simplified overview of the UNITY programming language. In fact, [CM88] avoids a precise specification of the language which provides some freedom to adopt a language suited for the particular application domain. As explained in [CM88] the UNITY programming language is a typed language and programs are given by: (1) a *declaration of program variables together with their types*, and (2) a finite *set of conditional multiple assignment statements*.³

The semantics of UNITY programs is an *interleaving semantics*: A *state* is a well-typed assignment of values to variables. In contrast to the original UNITY approach that requires programs to start in a state satisfying an initialization condition, which is a fixed part of the program, we omit the initialization condition and consequently use a more liberal semantics for reasons explained below. Therefore, an *execution sequence* is an infinite sequence of states that starts with an *arbitrary* state and it is obtained by the *fair* and *nondeterministic* selection and execution of statements. Fairness in UNITY means that each statement is executed infinitely often. It is also called *unconditional fairness*, since the conditions of statements are not taken into account.

Even though UNITY adopts a straightforward interleaving semantics, a concrete implementation can admit the concurrent execution of statements whenever the effect of such an execution cannot be distinguished from a sequential execution. In other words, UNITY can be regarded as a language which specifies valid state space transformations on an abstract level which typically allows many different (possibly concurrent) implementations.⁴

³For technical reasons, all UNITY programs are assumed to contain `skip`, a statement which does not have any effect [Mis94].

⁴The idea of avoiding the complications of explicit concurrency on the specification/programming level is even more noticeable in the Seuss language [Mis96, Mis98, Mis99], which can be regarded as a successor of the UNITY programming language that is more suitable for describing distributed and object-oriented systems. In fact, statements in the Seuss language are typically coarse-grained, e.g. a sequence of atomic statements, rather than fine-grained as the assignment statements in the UNITY programming language. In the abstract model of transition systems that we will use later in this chapter there is no need to make

4.1.2 The Temporal Logic

The UNITY methodology is based on a simple temporal logic comprising only a few temporal operators called STABLE, INVARIANT, UNLESS, ENSURES and LEADS TO. These are unary or binary operators formulating assertions on *state predicates*, i.e. predicates depending on the system state. Although the UNITY temporal logic should be classified as a variant of linear time temporal logic, there are some interesting peculiarities that make it impossible to define all its operators in terms of standard linear time logic [MP92].

The present section is inspired by a more recent presentation of UNITY logic called New UNITY in [Mis94, Mis95]. In addition to UNLESS and ENSURES, which are the basic operators in [CM88], we follow the New UNITY approach by introducing two new operators CO and TRANSIENT, since they provide a more elementary basis for defining the other operators.⁵

Unfortunately, a formal inconsistency in [CM88] has led to some confusion about the meanings of UNITY assertions. This problem and possible solutions are discussed in [Mis90b, Pet91, San91, Pra94, RF93, Kna94, Kin95, Hey97]. Assuming some familiarity with this issue — a good presentation of the essence of the problem can be found in [Kin95] — we briefly motivate the solution we adopt in our formalization. First of all, it appears that the most straightforward way to avoid the inconsistency is to abandon the so-called substitution axiom and to maintain the traditional quantification over all states (instead of the subset of reachable states) in the definitions of assertions. Unfortunately, this solution renders the UNITY logic incomplete for programs with an initialization condition [GP89]. Another possible solution is to keep the substitution axiom and to restrict quantification to reachable states [Kna94]. This, however, leads to severe difficulties with compositionality if the set of reachable states is a proper subset of all states [Kin95]. Possible solutions to these difficulties are presented in [CK97, Hey97].

It is interesting to note that both solutions mentioned above can be unified if the sets of all states and of reachable states would coincide, a property which in the original UNITY approach can be achieved by choosing a trivial initialization condition that is satisfied by all states. In fact, this amounts to working within a sound and complete fragment of the UNITY methodology where we only consider programs of this particular kind. This approach can be formally regarded as a *conservative restriction* rather than an *ad hoc modification* of UNITY. Of course, the trivial initialization condition of UNITY programs is useless under this view and their semantics can be simplified: Instead of speaking about programs with an initialization condition that is always satisfied, we have omitted the initialization condition and have adopted the liberal execution semantics explained above.

any assumptions about the granularity of statements, so that our formalization can be used to reason about UNITY programs as well as about Seuss programs.

⁵The operators CO and TRANSIENT have also been used in [Cha94].

It may appear that abandoning the initialization condition makes the formalism useless in practice, but in fact we do not lose expressivity. Speaking about initialization conditions will just become an issue relegated to the temporal logic. This leads to a more flexible approach where a program can be observed under possibly different initialization conditions. We will show in Section 4.4 how initialization conditions are made explicit in the temporal assertions. The problematic substitution axiom, which is trivially satisfied for UNITY programs under the new interpretation (and therefore becomes redundant), will be replaced by theorems about relativized temporal operators similar to the approach in [San91]. However, in contrast to [San91] we do not introduce new subscripted temporal operators. Instead we prefer to view relativized operators just as ordinary UNITY operators of a particular kind.

In the following we give a brief informal overview of the temporal logic in terms of the execution semantics. When we introduce a UNITY-style temporal logic for transition systems in the next section, however, we will *not* deal with a concrete execution semantics. Instead we will define temporal assertions directly in terms of the system in the spirit of [CM88]. The advantage of such an approach is not only its mathematical elegance but also a considerable degree of semantics independence.

The following four temporal assertions, which in fact can be regarded as a fragment of a linear time temporal logic, are properties of individual execution sequences lifted to sets of execution sequences (which by definition have to satisfy the requirement of unconditional fairness).

- p CO q :⁶
If p holds in an arbitrary state then q holds in the successor state.
- p STABLE:
Once p holds it remains true.
- p UNLESS q :
If p holds then p remains true at least as long as q does not hold.
- p LEADS TO q :
If p holds then q will eventually become true.⁷

In addition, there are two auxiliary operators of a different kind, which are usually not employed in high-level specifications as the assertions introduced so far. Instead they are used as auxiliary assertions to establish LEADS TO assertions during the verification phase:

⁶pronounced “ p constrains q ”

⁷This includes the case where q holds without any delay. We always use “eventually” in this sense.

- p TRANSIENT:
 p cannot hold permanently because there is a single statement which falsifies p once it is true.
- p ENSURES q :
 p UNLESS q and if p holds then q will eventually become true because there is a single statement which falsifies $p \wedge (\neg q)$.

An important requirement in the definition of TRANSIENT and ENSURES is that the single statement which is responsible for the progress is the same for all execution sequences under consideration. Such a condition cannot be expressed by a state-based linear time temporal logic such as the one of [MP92]. So far, we closely followed [Mis94, Mis95] for the definition of temporal assertions. The definition of TRANSIENT will be considerably generalized in the subsequent section.

In addition to the temporal operators introduced so far we use two more kinds of assertions: The IND. INVARIANT assertion, and the INVARIANT assertion.⁸ Both of them depend on an additional state predicate IC which will usually specify a set of states considered to be initial in a particular context:

- p IND. INVARIANT $_{IC}$:
 IC implies p and p STABLE.
- p INVARIANT $_{IC}$:
 p holds in every state reachable from some state satisfying IC .

It is clear that every inductive invariant is an invariant. The converse is not true, since an invariant is an assertion about reachable states only and does not imply stability for arbitrary states.⁹ Hence, (noninductive) invariants in general allow us to avoid over-specification of program behavior. On the other hand, noninductive invariants are less useful for compositional reasoning, since the set of reachable states is in general not robust under composition. Inductive invariants are instead compositional, since they are assertions about arbitrary states. As a consequence, both kinds of invariants are needed in practice. A clarifying discussion of these issues and their relation to the UNITY substitution axiom can be found in [Kin95].

⁸Since under our new interpretation (without initialization conditions in the programs) the original UNITY invariant assertion becomes useless (it is equivalent to STABLE), we introduce new invariant assertions which make use of an initialization condition that has to be specified explicitly.

⁹The distinction between invariant and inductive properties has already appeared in [Kel76], where place/transition nets with inscriptions are used as models for parallel programs. In the original approach to UNITY [CM88], however, there is no distinction between these two kinds of invariants. Indeed, the substitution axiom requires that original UNITY invariants are closed under implication, a property that holds for invariants but does not hold for inductive invariants in the above sense.

To conclude our presentation it is worth pointing out that UNITY also admits conditional assertions of the form $R \Rightarrow G$ for arbitrary assertions R and G . Such assertions are useful for rely-guarantee style compositional reasoning. In this context R is a assertion that a system component can rely on in order to guarantee G . For more details and examples using conditional assertions we refer to [CM88].

4.2 A UNITY-Style Logic for Labeled Transition Systems

From a semantic point of view UNITY programs can be seen as a particular kind of transition systems. These transition systems have been called *functional transition systems* in [Ste98], since for each state and for each action there is a unique successor state. In this section we define a UNITY-style temporal logic for arbitrary labeled transition systems. Transition systems have the advantage of being general enough to provide an abstract semantics for most kinds of program and system models. This class includes in particular different kinds of Petri nets and rewrite specifications as they have been studied in Chapter 3.

UNITY programs can be regarded as a special case of transition systems as follows: Each conditional multiple assignment statement of the program is viewed as an action. The associated transition relation takes the form of a function that describes the effect of the statement on the state. If the condition of a UNITY statement is not satisfied the corresponding action is enabled but the effect of the action is the identity relation on states. Hence, the enabledness predicate is trivially satisfied in such representations of UNITY programs.

Remember that in our definition of transition systems (Section 2.2) we do not specify a particular set of initial states. This corresponds to our choice not to have explicit initialization conditions in the system models we discussed previously, that is UNITY programs, Petri nets or rewrite specifications. We always consider the specification of initial states as an information that can be specified in addition to the system.

As defined in Section 2.2, transition systems can be equipped with an interleaving semantics, which in the most simple case is the liberal execution semantics. Often, a more restricted execution semantics is obtained as a subset of sequences from the liberal execution semantics by imposing fairness requirements. In the case of a UNITY program, the enablement predicate is always satisfied, which justifies the use of unconditional fairness: An execution sequence is *unconditionally fair* iff it contains each action infinitely often. In this and the following sections we use the more general notions of *weak fairness* and *weak group fairness*, which take enabledness of actions into account.

In the remainder of this section we will proceed as follows: After introducing the central notions of state predicate and state function, the presentation starts with elementary safety and liveness operators CO and TRANSIENT and defines the operators UNLESS and ENSURES in terms of these. Finally, the more complex liveness operator LEADS TO is introduced on top of ENSURES. The formalization in Section 4.3 will closely follow the definitions we give below, except for the fact that the logical language will be CIC rather than set theory.

4.2.1 State Predicates and Functions

For the following we presuppose an arbitrary but fixed labeled transition system with a set St of states, a set Act of actions, and a transition relation $\xrightarrow{e} \subseteq St \times St$ for each $e \in Act$.

Definition 4.2.1 A *state function* f is a function from the set of states into some domain. A *state predicate* p is a state function into the boolean domain $\mathbb{B} = \{\mathbb{F}, \mathbb{T}\}$. We write $p(s)$ to abbreviate $p(s) = true$. In this case we also say that s satisfies p or that p holds in s . We use \mathcal{SP} to denote the set of state predicates. A *family of state predicates* P indexed by a set I associates a state predicate P_i to each $i \in I$. Subsequently, p, q, r and P, Q, R range over state predicates and families of state predicates, respectively. Logical connectives, quantifiers and operators are naturally lifted to the level of state predicates and state functions. Additionally, the *everywhere operator* $[p]$ is available on state predicates: $[p]$ holds iff $p(s)$ holds for all states s .

4.2.2 Basic Concepts

The definitions of temporal operators will be based on two kinds of basic assertions, inspired by Hoare triples. Originally, Hoare triples of the form $\{p\} S \{q\}$ with state predicates p and q have been used as assertions about a program (statement) S [Hoa69]. In the partial correctness interpretation such a Hoare triple states that if the execution of S is initiated in a state satisfying p then by executing S we reach a state satisfying q whenever the execution of S terminates. In the total correctness interpretation such a Hoare triple also states that S terminates in every state satisfying p . Subsequently, we will adopt a similar notation for assertions about actions of a transition system.

For an action e , state predicates p and q and a set E of actions we define the following kinds of basic state predicates and assertions:

Definition 4.2.2 (Basic state predicates)

We define the function $EN(-) : Act \rightarrow \mathcal{SP} \rightarrow \mathbb{B}$ and its extension $EN(-) : \mathcal{PS}(Act) \rightarrow \mathcal{SP} \rightarrow \mathbb{B}$ as follows:

1. $\text{EN}(e)(s)$ holds iff e is enabled in s .
2. $\text{EN}(E)(s)$ holds iff E is enabled in s ,
i.e., there is an action $e \in E$ such that e is enabled in s .

Definition 4.2.3 (Basic assertions)

We define the ternary relations $-\{\cdot\}-, -\langle\cdot\rangle- \subseteq \mathcal{SP} \times \text{Act} \times \mathcal{SP}$ and their extensions $-\{\cdot\}-, -\langle\cdot\rangle- \subseteq \mathcal{SP} \times \mathcal{PS}(\text{Act}) \times \mathcal{SP}$ as follows:

1. $p \{e\} q$ holds iff for all states s, s' ,
 s satisfies p and $s \xrightarrow{e} s'$ implies s' satisfies q .
2. $p \langle e \rangle q$ holds iff $\text{EN}(e)(p)$ and $p \{e\} q$,
where $\text{EN}(e)(p)$ holds iff $\text{EN}(E)(s)$ for some s satisfying p .
3. $p \{E\} q$ holds iff for all states s, s' , and for each action $e \in E$,
if s satisfies p and $s \xrightarrow{e} s'$ then s' satisfies q .
4. $p \langle E \rangle q$ holds iff $\text{EN}(E)(p)$ and $p \{E\} q$,
where $\text{EN}(E)(p)$ holds iff $\text{EN}(E)(s)$ for some s satisfying p .

Observe that if e is not enabled at p the assertion $p \{e\} q$ holds vacuously, but the assertion $p \langle e \rangle q$ is not satisfied. We separated these two kinds of assertions, since temporal safety assertions can be defined using triples of the first kind, whereas definitions of liveness assertions will also involve triples of the second kind.

Notice that as in Hoare's original definition all the states of the transition system are considered in the definition above and not only a subset of reachable states. The justification is that the set of reachable states is usually complicated and unknown, in particular if we are concerned with an open system (component) that is part of a larger system. Indeed, as pointed out before, the set of reachable states is not robust under general composition (union) of transition systems (cf. [Kin95]). To support compositional reasoning quantification over all states is the most general choice, since it does not require any a priori restrictions on the composition of systems. In fact, we adopt the view that any restrictions should be made explicit using the temporal logic instead of complicating the system model.

4.2.3 Safety Assertions

Once basic state predicates and basic assertions are defined, a UNITY-style temporal logic can be built on top of these concepts. We first define safety assertions and then define liveness assertions in the next subsection.

Another noteworthy point is that all operators are relative to a *view*, i.e. to a subset of actions. By adopting a certain view E we focus on the changes performed only by actions in E , or more precisely we consider the restriction of the transition system induced by the actions in E . Views can be a useful structuring mechanism to manage large models and their verification. In fact, for compositional reasoning it is natural to regard the *components* of a system as special views which constitute a partitioning of the actions of the system.

We have the following safety assertions: $p \text{ CO } q$ expresses that if p holds at some point then q holds at the next state. $p \text{ UNLESS } q$ means that p holds at least until q holds (including the possibility that q will never hold after p and p will remain true forever). In other words, if we want to leave a state satisfying p but not q , then the only possibility is to move directly into a state satisfying q . The assertion $p \text{ STABLE}$ requires that once p is true, it continues to hold. For p to be inductively invariant w.r.t. some initialization condition IC , written $p \text{ IND. INVARIANT}_{IC}$, we additionally require that IC implies p . Furthermore, we say that p is an invariant w.r.t. IC , written $p \text{ INVARIANT}_{IC}$, to state that p is implied by some inductive invariant. In particular, it is implied by the strongest (inductive) invariant SI_{IC} , which obviously characterizes the set of reachable states. Hence, $p \text{ INVARIANT}_{IC}$ is equivalent to saying that p is satisfied for all states reachable from IC . As explained before, all assertions are relative to a view E .

Definition 4.2.4 (Safety assertions)

1. $p \text{ CO } q \text{ IN } E$ iff $p \{E\} q$.
2. $p \text{ UNLESS } q \text{ IN } E$ iff $(p \wedge \neg q) \text{ CO } (p \vee q) \text{ IN } E$.
3. $p \text{ STABLE IN } E$ iff $p \text{ CO } p \text{ IN } E$.
4. $p \text{ IND. INVARIANT}_{IC} \text{ IN } E$ iff $(IC \Rightarrow p)$ and $(p \text{ STABLE IN } E)$.
5. $p \text{ INVARIANT}_{IC} \text{ IN } E$ iff $\exists inv . (inv \text{ IND. INVARIANT}_{IC} \text{ IN } E) \wedge (inv \Rightarrow p)$.
6. $\text{SI}_{IC,E}(s)$ iff $\forall inv . (inv \text{ IND. INVARIANT}_{IC} \text{ IN } E) \Rightarrow inv(s)$.

Notice also that $p \text{ INVARIANT}_{IC} \text{ IN } E$ can also be written as $[\text{SI}_{IC,E} \Rightarrow p]$. In general, the strongest invariant $\text{SI}_{IC,E}$ is useful in high-level temporal specifications where overspecification has to be avoided by specifying the behavior of a system only for reachable states. Hence, we treat the strongest invariant as a state predicate which can be used in any specification. Typically, it will be used in connection with relativized assertions to be introduced in Section 4.3.4.

4.2.4 Liveness Assertions

Loosely speaking, safety assertions express that “something bad” must not happen and liveness assertions state that “something good” will eventually happen [Lam77].¹⁰ The original UNITY logic can express LEADS TO assertions, a special class of liveness assertions stating that if some condition holds at some point then some (other) condition will eventually be reached. Independent of UNITY, the relevance of LEADS TO assertions for concurrent program verification has already been pointed out in [Lam77]. In our opinion, an important contribution of the UNITY approach can be found in the tight integration of assertional and temporal logic viewpoints (concerning safety and liveness assertions) and related to this the elegant support for compositional reasoning.

In contrast to safety assertions (nontrivial) liveness assertions can only be proved if the system satisfies certain fairness requirements which are discussed next. According to the execution semantics of UNITY programs, each statement can be executed at any time. Even if its condition is not satisfied, the statement is considered to be executable, but its execution does not have any effect. As a consequence the simple notion of unconditional fairness, which just states that each statement is executed infinitely often, is sufficient in UNITY. In the more general setting of transition systems that we use here, we usually have a nontrivial enabledness predicate and we do not want to force actions to occur if they are not enabled.¹¹ Let us, therefore, first recapitulate the requirement of weak fairness¹², an immediate generalization of unconditional fairness, which is useful in the context of labeled transition systems. An action e which is designated as *weakly fair* behaves in the following way: If e is permanently enabled then e will occur eventually.¹³ The reason that we do not require all actions to be weakly fair is that in many systems we have actions which should not be forced to occur even if they are permanently enabled.¹⁴ A typical example is the interface between a system component and its environment which may activate the component at any time but should not be forced to do so.

Still weak fairness is not sufficient to guarantee progress in the case where a set of actions is permanently enabled. Recall that a set of actions is enabled iff it at least one of its actions is enabled. As in the linear time temporal logic of [Fra86] we employ a more general notion of weak fairness, which is called *weak group fairness* in the following. Instead of specifying weak fairness for single actions, weak fairness for arbitrary sets of actions can be specified. We require

¹⁰A formal definition of safety and liveness properties is given in [AS85].

¹¹We prefer to avoid solving this problem by modifying the transition system, e.g. by adding so-called idle actions which do not have an effect on the state.

¹²also called *justice* or *productivity*

¹³Weak fairness is treated in [Fra86] and [MP92]. In the latter reference it is called justice.

¹⁴The idea that some kind of fairness should be required for only a subset of all actions is also present in [MP92], [DGK⁺92] and [KR96].

that if a weakly fair set of actions is permanently enabled (i.e., at any time in the execution there is always some action in the set which is enabled) then some action of the set will eventually occur. Group fairness is often important if a set of actions is seen as a unity, or conversely, if the fine structure of a single action is exhibited by refining it to a set of actions. Typical examples of such groups occur in high-level Petri nets and rewrite specifications, where each net transition or rewrite rule can be executed in different modes and therefore corresponds to a set of actions rather than to a single one.

In the New UNITY approach the notion of fairness is elegantly encapsulated in the TRANSIENT operator. We generalize this operator from unconditional fairness to weak group fairness as follows: The assertion p TRANSIENT means that p cannot hold permanently, since once it is true it will eventually be falsified due to a *single* weakly fair group of actions. To express this requirement for progress we use SCO, the strong version of CO, which takes enabledness into account.¹⁵ On top of TRANSIENT, the assertion p ENSURES q is defined by requiring that p holds until q holds and $p \wedge \neg q$ cannot hold permanently, because of a single weakly fair group of actions which falsifies $p \wedge \neg q$. This implies that p holds until q holds and q will eventually become true. For the following definition we assume a set \mathcal{WF} of weakly fair groups of actions that is specified in addition to the transition system. To keep the definition of TRANSIENT as simple as possible, we adopt the convention that the empty group is always weakly fair and therefore contained in \mathcal{WF} .¹⁶

Definition 4.2.5 (SCO, TRANSIENT and ENSURES assertions)

1. p SCO q IN E iff $p \langle E \rangle q$.
2. p TRANSIENT IN E iff there is a weakly fair group $E' \in \mathcal{WF}$ such that $E' \subseteq E$ and p SCO $\neg p$ IN E' .
3. p ENSURES q IN E iff p UNLESS q IN E and $p \wedge \neg q$ TRANSIENT IN E .

Subsequently we use the standard inductive definition from [CM88] to introduce LEADS TO as the main operator to express liveness assertions. The assertion p LEADS TO q implies that if p holds at some point q will eventually hold.¹⁷ In

¹⁵SCO is introduced here for reasons of symmetry. It is not a temporal operator in (New)UNITY [Mis94, Mis95].

¹⁶For technical reasons, New UNITY assumes that each program contains a `skip` statement that does not have any effect. Thanks to our convention we do not need a corresponding assumption here.

¹⁷Notice that we only state an implication here. Although the UNITY temporal logic is complete [Pac90, Pac92] as we introduced it in Section 4.1, we have not studied the issue of completeness of our generalization, since we focus on metalogical aspects in this thesis, i.e., we do not formalize a particular execution semantics.

contrast to the meaning of p ENSURES q , p does not necessarily hold until q holds, but it may become false in the meantime.

Definition 4.2.6 (LEADS TO assertions)

p LEADS TO q IN E is inductively defined by the following rules:

$$\frac{p \text{ ENSURES } q \text{ IN } E}{p \text{ LEADS TO } q \text{ IN } E} \quad (\text{basis})$$

$$\frac{p \text{ LEADS TO } q \text{ IN } E, q \text{ LEADS TO } r \text{ IN } E}{p \text{ LEADS TO } r \text{ IN } E} \quad (\text{transitivity})$$

$$\frac{\forall i \in I : P_i \text{ LEADS TO } q \text{ IN } E}{(\exists i \in I : P_i) \text{ LEADS TO } q \text{ IN } E} \quad (\text{disjunction})$$

A final remark concerning this definition is that the use of an inductive rather than a semantic formulation emphasizes the assertional flavour and the view of UNITY as a formal system as opposed to a collection of semantically defined temporal operators. Hence, we think that metalogical reasoning, in the sense of reasoning about a formal system, can be fruitfully applied, although the formal system is mostly represented using a shallow embedding technique which abstracts from syntactic details.

4.3 A Verified Temporal Logic Library

A suitable embedding of a notion of concurrent system and an associated temporal logic into a formal framework enables us to apply the framework and its supporting tools to: (1) reason about concrete systems using the temporal logic and (2) reason about the temporal logic itself. Both issues are closely connected with each other, since as a matter of economy reasoning about systems ideally makes use of proofs rules that are obtained by reasoning about the temporal logic. Furthermore, the issues (1) and (2) can be regarded as two extremes of a spectrum ranging from reasoning about very concrete system specifications up to a very abstract setting, where we do not assume anything about the system. Intermediate levels of abstraction would include parametrized system specifications and classes of systems of a particular kind.

Before we begin with the detailed presentation of the formal development we would like to give some further motivation for our work. There are a number of good reasons why in particular an embedding of a UNITY-style temporal logic into CIC (or similar type theories) is worth investigating:

- CIC is an *expressive type theory* and hence provides an important ingredient that is left unspecified in the UNITY methodology. *Inductive definitions*

allow powerful data types. *Parameterization* by elements and types allows highly generic program/system specifications.

- The *higher-order logic* of CIC is expressive enough to support general mathematical reasoning (algebra, graph theory, etc.). Often correctness proofs rely on mathematical facts in an essential way, but the UNITY methodology does not address this issue. We think that a useful system verification tool can benefit from having good support for *general purpose theorem proving* beyond its specialized capabilities.
- The potential that arises from the tight integration of type theory and higher-order logic with a UNITY-style temporal logic is a very useful by-product of the *shallow embedding* into CIC that we define. In addition, the use of a shallow rather than a deep embedding avoids excessive syntactic details and allows us to focus on the essential aspects of the object logic.
- CIC is a *functional programming language*, which can be used to represent atomic actions (e.g. the statements of the UNITY programming language) as functions operating on the program state. The execution of such representations of atomic actions and finite sequences of these is possible within CIC thanks to its computational nature. Furthermore, CIC is strongly normalizing, i.e., functional programs are terminating, which is the most essential property of atomic actions. As a consequence, termination of atomic actions is proved by type checking alone.
- We furthermore employ the *higher-order logic* of CIC for representing the temporal logic and for metalogical reasoning about its state predicates and assertions. As a consequence, *proof rules become theorems* in higher-order logic and can be themselves subject to verification. Another by-product of the use of CIC as a metalogic is that conditional assertions, which are used in UNITY for rely-guarantee style reasoning, can be directly expressed as implications in the metalogic.
- In this thesis we deal with the representation and the reasoning about formal systems, and this chapter studies a UNITY-style temporal logic as a typical example. In fact, the representation of the main fragment of the temporal logic, that is given by LEADS TO assertions, will be defined inductively in terms of the more elementary temporal operators. In essence, the object-logic is an *inductively defined formal system* and we use CIC not only as a *logical framework* to represent a formal system but also as a *metalogic* to reason about it.
- Often an embedding into a formal logic such as CIC leads to a more *precise presentation* and a *better understanding* of the object logic, e.g., program variables and meta variables are clearly distinguished, proofs are completely

formalized. Formal (metalogical) proofs allow us to keep track of dependencies, which does not only lead to a better understanding, but is a good setting to experiment with potential generalizations of the object logic such as the one we describe in this chapter.

- Last but not least, and as a particular benefit of the view of the UNITY-style temporal logic as an inductively defined formal system, we will be able to use the *propositions-as-types and proofs-as-objects interpretation* to make explicit a notion of temporal logic proofs, and we will also demonstrate the benefits of reasoning about such proofs in the context of *compositional reasoning*.

In the remainder of this section we will first present the formal development of the temporal logic library within an abstract framework which avoids application-specific assumptions. We describe the development in some detail to provide a documentation that encourages interested researchers to use or extend our formal development. Each of the following subsections presents a single module of the development. Later we explain the use of the library by means of a toy example, namely the well-known “Common Meeting Time Problem” with two parties. Finally, we discuss the connection between the propositions-as-types/proofs-as-objects interpretation and compositionality. After developing some general results we demonstrate their application to a simple extension of the “Common Meeting Time Problem” by a third party.

4.3.1 Mathematical Prelude

We begin with the module `init` in which we: (1) prepare the use of the universe `Prop` to represent the propositions of a *classical logic with axioms for logical extensionality and proof irrelevance*, and (2) include further mathematical libraries, in particular those providing data types and support for boolean values, natural numbers, sets and relations, well-founded relations and dependent equality.

An intuitionistic development of a UNITY-style temporal logic seems to be possible if the definitions of the temporal operators are slightly modified w.r.t. the original presentation in [CM88]. However, in the present work we have chosen to stay close to the original definitions of the temporal assertions. Consequently, we decided to employ the following “classical” axioms whenever helpful to derive temporal proof rules. In fact we found that these axioms are rarely needed; we will mention each instance of their use in our development. A classical set-theoretic semantics for CIC that justifies all these axioms can be found in [Wer97]. Alternatively, it is possible to use the semantics presented in Chapter 8 of this thesis.

A minimal form of classical logic is obtained by using the single axiom `classic` given below, expressing the law of the excluded middle for the universe `Prop`.

```
Axiom classic : (p:Prop)(p ∨ ~p).
```

In addition, we assume logical extensionality, stating that logically equivalent propositions in `Prop` are equal.

```
Axiom log_extensionality : (p,q:Prop)(iff p q)->(p==q).
```

Furthermore, we assume proof irrelevance for proofs of propositions in `Prop`, i.e. two proofs of the same proposition are equal.

```
Axiom proof_irrelevance : (P:Prop)(p,q:P)(p==q).
```

The last axiom provides a means to cast an object of type $(F\ x)$ into an object of type $(F\ y)$ given a proof of $x==y$. In CIC this elimination principle, which is in fact the `Type` counterpart of the induction principle `eqT_ind`, is not provided by the inductive definition of `==`, since its definition would require the construction of an informative object from a noninformative one, which is impossible in CIC.

```
Axiom eqT_rec : (A:Type; x:A; F:(A->Type))(F x)->(y:A)x==y->(F y) .
```

Next we include a number of library modules that we briefly explain below.

```
Require Arith.
Require Compare_dec.
Require Peano_dec.
Require Minus.
Require Arith_suppl.
Require Ensembles.      Definition set := [T : Type] (Ensemble T) .
Require Classical_sets .
Require Relations.
Require WfT.
Require WfT_nat.
Require EqdepT.
Require Finite_types.
```

The modules `Arith`, `Peano_dec`, `Compare_dec` and `Minus` contain definitions and facts related to the type `nat` of natural numbers and the standard ordering which is given by predicates `le` and `lt` for “less or equal” and “less than”, respectively. The module `Arith_suppl` contains a set of auxiliary theorems about `nat` needed

in some proofs. The type `(Ensemble T)` of sets over a type `T`, which is defined as `T -> Prop` and we prefer to write as `(set T)`, and corresponding operations are provided by the modules `Ensembles` and `Classical_sets`. Operations that we frequently use are `(Empty_set T)`, denoting the empty set over `T`, `(Add T S s)` denoting the addition of an element `s` to a set `S` over `T` and `(Union T S S')`, denoting the union of sets `S` and `S'` over `T`. `Classical_sets` uses the axiom `classic` given above. The module `EqdepT` provides a dependent equality. Compared with the standard polymorphic equality it is more flexible concerning the argument types (they do not have to be computationally equivalent). It is used in Section 4.3.8 for state spaces with typed variables. Relations and their well-foundedness are defined in the modules `Relations` and `WfT`, respectively. Well-foundedness of the standard ordering on natural number is provided by the module `WfT_nat`. These modules will be used in Section 4.3.5, where temporal logic proof rules for well-founded induction are developed. Finally, we need cardinality and finiteness of types, which is provided by the module `Finite_types`.

The modules `EqdepT`, `WfT`, and `WfT_nat` are our adaptations of the module `Eqdep`, `Wf`, and `Wf_nat`, respectively, provided by the COQ library, the main change being that it uses the universe `Type` rather than the universe `Set`. These modifications reflect our intention of using the predicative universe `Type` for data types rather than the impredicative universe `Set`, which we do not use in any essential way.¹⁸ Except for `Arith_suppl` and `Finite_types` all other modules are part of the mathematical library distributed with the COQ system, and we refer to [BBC⁺99] for more details.

4.3.2 An Abstract Frame

This section describes the module `abstract` which provides an abstract frame for the temporal logic library. This frame is abstract in the sense that it is a skeleton assuming an arbitrary transition system and includes all library modules which do not require any specific assumptions about this system. A slightly less abstract instance of this module will be presented in Section 4.3.7, where we replace the abstract state space by a state space with typed variables. A more concrete instance of this module will be our example in Section 4.4, where we verify a particular program specified by a unique transition system.

We begin by including the module `init`, which in turn includes the libraries discussed in the previous section.

Load `init`.

¹⁸In the COQ library the data types reside in `Set` (which is a subuniverse of `Type`), but we do not exploit its impredicativity in this development. Hence, we assume without loss of generality that `Set` is not used at all, an important point, which makes it possible to understand our development in a purely set-theoretic way, e.g. on the basis of [Wer97].

We now introduce the type `st` of states and the type `act` of actions. In this section we are concerned with the most abstract setting, where we avoid any further assumptions on these types.

```
Variable st:Type.
```

```
Variable act:Type.
```

The ternary transition relation `trans` is declared next. Given an action `e` and states `s`, `s'` the proposition `(trans e s s')` states that there is a state transition from `s` to `s'` with an action `e`.

```
Variable trans : act->st->st->Prop .
```

We also need the notion of a view, which is just a set of actions. As explained before, a view induces a restriction of the presupposed transition system and all temporal assertions are defined relative to a view. For brevity, we often identify a view and its induced transition system in informal explanations.

```
Definition view := (set act).
```

In addition to the transition system we assume a set `wf` of weakly fair groups of actions, which should at least contain the empty group by our convention.

```
Variable wf : (set (set act)).
```

```
Hypothesis wf_contains_empty_set :
  (In (set act) wf (Empty_set act)).
```

For the modules included below we just give some brief explanations here, since they all will be discussed in more detail in the subsequent sections.

The module `state_pred` defines the central type `s_prop` of state predicates and provides a collection of operators lifted from propositions `Prop` to state predicates `s_prop`.

```
Load state_pred.
```

The module `safety` defines temporal logic operators for specifying safety assertions, in particular `co`, `unless`, `stable` and `invariant` and contains a variety of theorems relating these assertions to the system and to each other.

```
Load safety.
```

The module `liveness` contains definitions of temporal operators used to specify liveness assertions. These operators are `transient`, `ensures` and `leads_to`. Also a collection of theorems relating safety and liveness assertions are provided.

Load `liveness`.

Finally, we include the module `composition` which provides support for compositional reasoning.

Load `composition`.

4.3.3 State Predicates

The module `state_pred` introduces the type `s_prop` of state predicates, i.e. propositions depending on states.

Definition `s_prop := (st->Prop)`.

The usual logical operators are lifted from `Prop` to `s_prop` in the natural way.

Definition `s_true := [p:st]`

True.

Definition `s_false := [p:st]`

False.

Definition `s_or := [p,q:s_prop]`

`[s:st] (p s) ∨ (q s)`.

Definition `s_or3 := [p,q,r:s_prop]`

`[s:st] (p s) ∨ (q s) ∨ (r s)`.

Definition `s_and := [p,q:s_prop]`

`[s:st] (p s) ∧ (q s)`.

Definition `s_and3 := [p,q,r:s_prop]`

`[s:st] (p s) ∧ (q s) ∧ (r s)`.

Definition `s_imp := [p,q:s_prop]`

`[s:st] (p s) -> (q s)`.

Definition `s_iff := [p,q:s_prop]`

`[s:st] (p s) <-> (q s)`.

Definition `s_not := [p:s_prop]`

`[s:st] ~(p s)`.

Definition `s_ex := [T:Type] [P:T->s_prop]`

`[s:st] (ExT [t:T] ((P t) s))`.

Definition `s_all := [T:Type] [P:T->s_prop]`

`[s:st] (t:T)((P t) s)`.

Definition `s_all_nat := [b:nat] [P:nat->s_prop]`

`[s:st] ((i:nat)(lt i b)->((P i) s))`.

Definition `everywhere` := `[p:s_prop] (s:st)(p s)`.

An exception is the operator `everywhere`, which does not produce a state predicate but a (state-independent) proposition.

In addition to the type `s_prop` of state predicates we introduce the type `ds_prop` of dependent state predicates, i.e. state predicates that are conditional in another state predicate. More precisely, `(ds_prop r)` for some `r:s_prop` denotes the type of propositions that do not only depend on a state `s` but can also assume that `(r s)` holds. Again the standard logical operators are lifted to this new type.

Definition `ds_prop` := `[r:s_prop](s:st)(r s)->Prop`.

Definition `ds_or` :=

```
[r:s_prop] [dp:(ds_prop r)] [dq:(ds_prop r)]
[s:st] [prf:(r s)] ((dp s prf) \\/ (dq s prf)).
```

Definition `ds_and` :=

```
[r:s_prop] [dp:(ds_prop r)] [dq:(ds_prop r)]
[s:st] [prf:(r s)] ((dp s prf) /\ (dq s prf)).
```

Definition `ds_not` :=

```
[r:s_prop] [dp:(ds_prop r)]
[s:st] [prf:(r s)] ~(dp s prf).
```

Definition `ds_true` :=

```
[r:s_prop]
[s:st] [prf:(r s)] True.
```

Definition `ds_false` :=

```
[r:s_prop]
[s:st] [prf:(r s)] False.
```

Definition `ds_ex` :=

```
[r:s_prop] [d:(ds_prop r)]
[s:st] (ExT [prf:(r s)] (d s prf)).
```

Definition `ds_all` :=

```
[r:s_prop] [d:(ds_prop r)]
[s:st] (prf:(r s))(d s prf).
```

The following operator allows us to cast the proof of a dependent predicate from `(ds_prop r')` to a proof of `(ds_prop r)` if `r` implies `r'`.

Definition `ds_cast` := `[r,r':s_prop]`

```
[imp:(everywhere (s_imp r r'))] [dp:(ds_prop r')]
[s:st] [prf:(r s)] (dp s (imp s prf)).
```

Theorems about logical operators on propositions `Prop` have obvious counterparts at the level of state predicates `s_prop` and dependent state predicates `ds_prop` that we have omitted here.

The axiom `classic` has the following counterpart at the level of state predicates:

Theorem `s_classic` : $(p:s_prop)(\text{everywhere } (s_or\ p\ (s_not\ p)))$.

Also, logical extensionality can be lifted from propositions to state predicates, if we assume functional extensionality for `s_prop`.

Axiom `s_prop_extensionality` : $(p,q:s_prop)$
 $((s : st)(p\ s)==(q\ s))\rightarrow(p==q)$.

Theorem `s_log_extensionality` : $(p,q:s_prop)$
 $(\text{everywhere } (s_iff\ p\ q))\rightarrow(p==q)$.

4.3.4 Safety Assertions

The module `safety` contains the definitions of the temporal logic operators `co`, `unless`, `stable`, `invariant` and `invariant` for safety assertions. As explained before, we follow the New UNITY approach [Mis94], where `co` is chosen as the basic safety operator and the other operators are defined on top of it. This is not a major deviation from [CM88] but rather a more systematic way to structure the definitions.

We start with the definition of the `co` operator. Recall that for two predicates `p` and `q` the assertion $(co\ E\ p\ q)$ means that if `p` holds at some state then `q` holds at each possible successor state in `E` no matter which action occurs.

Definition `hoare` := $[e:act]\ [p,q:s_prop]$
 $((s,s':st)(\text{trans } e\ s\ s')\rightarrow(p\ s)\rightarrow(q\ s'))$.

Definition `co` := $[E:view]\ [p,q:s_prop]$
 $(e:act)(\text{In } act\ E\ e)\rightarrow(\text{hoare } e\ p\ q)$.

Subsequently, we define further operators in terms of `co`. Notice that all operators are relative to a view `E` specified as the first argument. The other arguments of the temporal operators are state predicates, usually denoted by `p`, `q` and `r`.

Remember that the assertion $(\text{unless } E\ p\ q)$ states that in `E`, `p` holds unless `q` holds, or more precisely, if `p` holds at some state and `q` does not hold then at the next step `p` remains true or `q` becomes true. It is important to see that progress does not have to take place, i.e. the assertion does not exclude the possibility that `p` remains true forever and `q` never holds.

Definition `unless` := `[E:view] [p,q:s_prop]`
`(co E (s_and p (s_not q)) (s_or p q)).`

The assertion `(stable E p)` asserts that `p` is stable, i.e. once `p` becomes true in `E` it continues to hold.

Definition `stable` := `[E:view] [p:s_prop] (co E p p).`

The stronger assertion `(ind_invariant E IC p)` states that `p` is an inductive invariant in `E` w.r.t. an initialization condition `IC`. In addition to stability of `p` it is required that `p` is implied by the `IC`.

Definition `ind_invariant` :=
`[E:view] [IC:s_prop] [p:s_prop]`
`(everywhere (s_imp IC p)) /\ (stable E p).`

We also introduce assertions of the form `(invariant E IC p)` which state that `p` is implied by some inductive invariant in `E` w.r.t. an initialization condition `IC`.

Definition `invariant` :=
`[E:view] [IC:s_prop] [p:s_prop]`
`(ExT [inv:s_prop] (ind_invariant E IC inv) /\`
`(everywhere (s_imp inv p))).`

Finally, we have the state predicate `(si E IC)`, called the strongest invariant [Lam90], which characterizes the set of reachable states.

Definition `si` :=
`[E:view] [IC:s_prop] [s:st]`
`(p:s_prop)(ind_invariant E IC p)->(p s).`

This concludes the list of temporal operators used to formulate safety assertions. The remainder of this module proves a variety of theorems about these operators and their mutual relationship. Most proof rules contained in [CM88] are among these theorems. Also many proof rules for `co` are proved here, among them also rules given in [Mis94].

We start with a combined left hand side strengthening and right hand side weakening rule for `co` assertions.

Theorem `co_implication` : `(E:view)(p,p',q,q':s_prop)`
`(everywhere (s_imp p p')) ->`
`(everywhere (s_imp q q')) ->`
`(co E p' q) -> (co E p q').`

Two `co` assertions can be combined into a new one by taking the conjunction of their left hand sides and their right hand sides, respectively.

```
Theorem co_conjunction : (E:view)(p,p',q,q':s_prop)
  (co E p q) -> (co E p' q') ->
  (co E (s_and p p') (s_and q q')).
```

The rule `co_conjunction` can be specialized for `stable` as follows.

```
Theorem co_stable_conjunction : (E:view)(p,q,r:s_prop)
  (stable E r) -> (co E p q) ->
  (co E (s_and p r) (s_and q r)).
```

The following rules are useful to derive *relativized assertions*, i.e. assertions which do not make statements about all states but only about a subset of states usually specified by some inductive invariant. Formally, we conceive relativized assertions as ordinary assertions with some additional restriction expressed by conjunction. For instance, `(co E (s_and r p) (s_and r q))` and `(stable E (s_and r p))` are relativations of `(co E p q)` and `(stable E p)` w.r.t. some inductive invariant `r`, respectively. The following two theorems show that, given a goal that is a relativized assertion of one of these kinds, we can prove this goal by exploiting `r` not only on the left hand side but also as an assumption on the right hand side if we know at least that `r` is stable.

```
Theorem rel_co : (E:view)(p,q,r:s_prop)
  (stable E r)->(co E (s_and r p) (s_imp r q))->
  (co E (s_and r p) (s_and r q)).
```

```
Theorem rel_stable : (E:view)(p,r:s_prop)
  (stable E r)->(co E (s_and r p) (s_imp r p))->
  (stable E (s_and r p)).
```

Rules of this kind for other forms of related assertions will be given later. In general, such rules allow us to use a state predicate `r`, which is typically an inductive invariant, as an assumption on both sides of the underlying implication rather than leading to proof obligation for `r` on the right hand side. The purpose of these rules is similar to that of the substitution axiom (cf. [CM88]), but allows us to use inductive invariants as assumptions in a more disciplined way. Recall that the substitution axiom allows us to use UNITY invariants everywhere, i.e. it assumes that they are true in all states, which leads to the problems discussed in Section 4.1. In fact, relativized assertions as they are introduced in this chapter

correspond closely to the relativized assertions introduced in [San91] (and corrected in [Pra94]) as one solution to the dilemma posed by the substitution axiom. However, instead of introducing new families of temporal operators indexed by inductive invariants, we prefer to use the original UNITY-style operators to express them.

We also have the following generalizations of the previous two proof rules from state predicates to dependent state predicates p and q . Relativized assertions of the form $(\text{co } E \text{ (ds_ex } r \text{ } p) \text{ (ds_ex } r \text{ } q))$ as in the conclusion of these rules are often needed if the propositions $(p \text{ } s)$ and $(q \text{ } s)$ depend on the proof of $(r \text{ } s)$. We first added dependent state predicates and corresponding relativized assertions in order to cope with problems related to the static character of the type theory.¹⁹ A typical example would be the use of a partial function which is only meaningful under the premise $(r \text{ } s)$. In CIC such a function is naturally represented as a total function that depends on a proof of $(r \text{ } s)$. More generally, we view dependent state predicates as a solution to the difficulties arising from the use of type theory rather than set-theory for formal program/system verification (cf. [LP97]). Such generalized proof rules for relativized assertions can be proved for all temporal operators, but for the sake of brevity we only state them here.

```
Theorem rel_co_dep : (E:view)(r:s_prop)
  (stable E r)->(p,q:(s:st)(r s)->Prop)
  (co E (ds_ex r p) (ds_all r q))->
  (co E (ds_ex r p) (ds_ex r q)).
```

```
Theorem rel_stable_dep : (E:view)(r:s_prop)
  (stable E r)->(p:(s:st)(r s)->Prop)
  (co E (ds_ex r p) (ds_all r p))->
  (stable E (ds_ex r p)).
```

For assertions involving dependent state predicates we often have rules to strengthen the condition the predicates depends on such as the following:

```
Theorem rel_co_dep_strengthen : (E:view)(r,r':s_prop)
  (stable E r)->
  (ev:(everywhere (s_imp r r'))))
  (p,q:(s:st)(r' s)->Prop)
  (co E (ds_ex r' p)
    (ds_ex r' q))->
  (co E (ds_ex r [s:st][prf:(r s)](p s (ev s prf)))
    (ds_ex r [s:st][prf:(r s)](q s (ev s prf))))).
```

¹⁹in the context of formal reasoning about Petri nets

```

Theorem rel_stable_dep_strengthen : (E:view)(r,r':s_prop)
  (stable E r)->
  (ev:(everywhere (s_imp r r'))))
  (p:(s:st)(r' s)->Prop)
  (stable E (ds_ex r' p))->
  (stable E (ds_ex r [s:st][prf:(r s)](p s (ev s prf))))).

```

Above we have introduced the strongest invariant ($\text{si } E \text{ IC}$), which is a state predicate characterizing the set of states reachable from some state satisfying IC . It is interesting to note that the use of si makes it unnecessary to deal with reachable states directly, just as there will be no need to assume and formalize a particular execution semantics. In the following some basic facts about the relationship between si , ind_inv and invariant will be given.

```

Theorem ind_inv_is_inv : (E:view)(IC:s_prop)(p:s_prop)
  (ind_invariant E IC p)->(invariant E IC p).

```

```

Theorem si_is_indinv : (E:view)(IC:s_prop)
  (ind_invariant E IC (si E IC)).

```

```

Theorem si_is_inv : (E:view)(IC:s_prop)
  (invariant E IC (si E IC)).

```

```

Theorem indinv_imp_si : (E:view)(IC:s_prop)(p:s_prop)
  (ind_invariant E IC p)->
  (everywhere (s_imp (si E IC) p)).

```

The next two results together provide a characterization of invariant in terms of the strongest invariant given by si . Since, $(\text{si } E \text{ IC})$ characterizes the set of states reachable w.r.t. IC in E , the result says that $(\text{invariant } E \text{ IC } p)$ holds iff p is true at all such reachable states.

```

Theorem inv_imp_si : (E:view)(IC:s_prop)(p:s_prop)
  (invariant E IC p)->
  (everywhere (s_imp (si E IC) p)).

```

```

Theorem si_imp_inv : (E:view)(IC:s_prop)(p:s_prop)
  (everywhere (s_imp (si E IC) p))->
  (invariant E IC p).

```

The following theorem specializes the theorem `rel_stable` given before to relativized inductive invariants. Here it is necessary to require that `r` is not only stable but is also an inductive invariant. The theorem is particularly useful to prove and strengthen invariants incrementally. Notice that `r` cannot only be exploited in the proof of the right hand side of the `co` premise but also to establish that `p` holds initially.

```
Theorem rel_inv : (E:view)(IC,p,r:s_prop)
  (ind_invariant E IC r)->
  (everywhere (s_imp (s_and IC r) p))->
  (co E (s_and r p) (s_imp r p))->
  (ind_invariant E IC (s_and r p)).
```

Using `co_stable_conjunction` and `co_implication` we can prove a conjunction rule for `stable` and `unless`, stating that a stable state predicate `r` can be added by conjunction to both sides of an existing `unless` assertion.

```
Theorem unless_stable_conjunction : (E:view)(p,q,r:s_prop)
  (stable E r) -> (unless E p q) ->
  (unless E (s_and p r) (s_and q r)).
```

Extending the collection of proof rules for relativized assertions `rel_co`, `rel_stable` and `rel_inv` we add corresponding rules for relativized `unless` assertions.

```
Theorem rel_unless : (E:view)(p,q,r:s_prop)
  (stable E r)->
  (co E (s_and3 r p (s_not q)) (s_imp r (s_or p q)))->
  (unless E (s_and r p) (s_and r q)).
```

```
Theorem rel_unless_dep : (E:view)(r:s_prop)
  (stable E r)->(p,q:(s:st)(r s)->Prop)
  (co E (ds_ex r (ds_and r p (ds_not r q)))
    (ds_all r (ds_or r p q)))->
  (unless E (ds_ex r p) (ds_ex r q)).
```

```
Theorem rel_unless_dep_strengthen : (E:view)(r,r':s_prop)
  (stable E r)->
  (ev:(everywhere (s_imp r r'))))
  (p,q:(s:st)(r' s)->Prop)
  (unless E (ds_ex r' p)
    (ds_ex r' q))->
  (unless E (ds_ex r (ds_cast r r' ev p))
    (ds_ex r (ds_cast r r' ev q))).
```

The following theorem `co_big_conjunction` provides a conjunction rule for an arbitrary family of `co` assertions indexed by a type `T` which may be infinite in general. Notice that `(s_all T P)` is the conjunction over the family `P` of state predicates indexed by `T`.

```
Theorem co_big_conjunction : (T:Type) (E:view) (P,Q:T->s_prop)
  ((t:T)(co E (P t) (Q t)))->
  (co E (s_all T P) (s_all T Q)).
```

There are rules for disjunction similar to those for the conjunction of `co` assertions. Again we have a finitary version and a general version for an arbitrary family of premises. Below `(s_ex T P)` is the disjunction over the family `P` of state predicates indexed by `T`.

```
Theorem co_big_disjunction : (T:Type) (E:view) (P,Q:T->s_prop)
  ((t:T)(co E (P t) (Q t)))->
  (co E (s_ex T P) (s_ex T Q)).
```

```
Theorem co_disjunction : (E:view) (p,p',q,q':s_prop)
  (co E p q) -> (co E p' q') ->
  (co E (s_or p p') (s_or q q')).
```

Remember that `s_false` and `s_true` are state propositions which are true and false for all states, respectively. Here we use them to state a few trivial properties of `co` and `unless`.

```
Theorem co_false : (E:view) (p:s_prop)
  (co E s_false p).
```

```
Theorem co_true : (E:view) (p:s_prop)
  (co E p s_true).
```

```
Theorem unless_true : (E:view) (p:s_prop)
  (unless E p s_true).
```

In the simplest case, `unless` assertions can be derived from implications.

```
Theorem unless_imp : (E:view) (p,q:s_prop)
  (everywhere (s_imp p q))->(unless E p q).
```

An equivalence between `(stable E p)` and `(unless E p s_false)` is proved by the following two implications. In [CM88] this equivalence is taken to define `stable`.

Theorem `stable_imp_unless` : (E:view)(p,q:s_prop)
 (stable E p)->(unless E p q).

Theorem `unless_imp_stable` : (E:view)(p:s_prop)
 (unless E p s_false)->(stable E p).

Using the general disjunction rule `co_big_disjunction` for `co` we can prove a similar rule for `unless`.

Theorem `unless_big_disjunction` :
 (E:view) (T:Type) (P,Q:T->s_prop)
 ((i:T)(unless E (P i) (Q i)))->
 (unless E (s_ex T P) (s_ex T Q)).

The following rule allows weakening of the right hand side of `unless`. It is known as the consequence weakening rule.

Theorem `unless_implication_r` : (E:view)(p,q,r:s_prop)
 (everywhere (s_imp q r)) ->
 (unless E p q) -> (unless E p r).

We also have a conjunction rule for `unless`, but notice that, in contrast to the conjunction rule for `co`, we have a disjunction of three state predicates on the right hand side depending on whether progress takes place in the second, in the first or in both premises.

Theorem `unless_conjunction` : (E:view)(p,p',q,q':s_prop)
 (unless E p q) -> (unless E p' q') ->
 (unless E (s_and p p')
 (s_or3 (s_and p q') (s_and p' q) (s_and q q')))).

Right hand side weakening of the previous conjunction rule yields the simple conjunction rule.

Theorem `unless_simple_conjunction` : (E:view)(p,p',q,q':s_prop)
 (unless E p q) -> (unless E p' q') ->
 (unless E (s_and p p') (s_or q q'))).

As a special case of `unless_simple_conjunction` we obtain the following binary conjunction rule for `unless`, which is generalized by induction to a finite family of premises in the theorem `unless_left_big_conjunction`.

```
Theorem unless_left_conjunction : (E:view)(p,p',q:s_prop)
  (unless E p q) -> (unless E p' q) ->
  (unless E (s_and p p') q).
```

```
Theorem unless_left_big_conjunction : (E:view)
  (b:nat)(Q:nat->s_prop)(r:s_prop)
  ((i:nat)(lt i b)->(unless E (Q i) r))->
  (unless E (s_all_nat b Q) r).
```

In analogy to `unless_simple_conjunction` there is a simple disjunction rule for `unless` formulated in the following theorem.

```
Theorem unless_simple_disjunction : (E:view)(p,p',q,q':s_prop)
  (unless E p q) -> (unless E p' q') ->
  (unless E (s_or p p') (s_or q q')).
```

Similar to `unless_conjunction` we can combine two `unless` assertions in a disjunctive fashion using the rule `unless_disjunction`. The proof is done by applying `co_disjunction` and `co_implication`.

```
Theorem unless_disjunction : (E:view)(p,p',q,q':s_prop)
  (unless E p q) -> (unless E p' q') ->
  (unless E (s_or p p')
    (s_or3
      (s_and (s_not p) q') (s_and (s_not p') q) (s_and q q')))).
```

It is interesting to note that the proof of `unless_simple_disjunction` is the only proof in the current version of the temporal logic library which relies on the use of the classical axiom `classic`. Without this axiom it would not be possible to make use of the left hand side of the `co` assertion in the conclusion which is of the form

$$\begin{aligned} & (s_and \ (s_or \ p \ p') \\ & \quad (s_not \ (s_or3 \ (s_and \ (s_not \ p) \ q') \\ & \quad \quad (s_and \ (s_not \ p') \ q) \\ & \quad \quad (s_and \ q \ q')))) \end{aligned}$$

if we expand the definition of `unless`. Consider a state satisfying this predicate. Intuitionistically, we can infer that in this state `q` and `q'` cannot hold simultaneously, but we do not know which one does not hold. In other words, we have `(s_and p (s_not q))` or `(s_and p' (s_not q'))` classically, but it does not hold intuitionistically, so that without the classical axiom it is undecidable which of the two `unless` premises should be applied.

Finally, we use `unless_disjunction` to prove the following cancelation rule: If we can leave a state satisfying `p` only via a state satisfying `q` and a state satisfying `q` only via a state satisfying `r`, it follows that a state satisfying `(s_or p q)` can only be left via `r`.

```
Theorem unless_cancelation : (E:view)(p,q,r:s_prop)
  (unless E p q) -> (unless E q r) ->
  (unless E (s_or p q) r).
```

4.3.5 Liveness Assertions

As in New UNITY the liveness assertions `ensures` and `leads_to` are built on top of `unless` and `transient` assertions. This is done in the module `liveness`, which is discussed below and also contains general proof rules for liveness assertions.

We begin by defining enabledness of an action `e` as a state predicate (`enabled e`) which is satisfied for those states `s` where the action `e` may occur.

```
Definition enabled := [e:act][s:st]
  (ExT ([s':st] (trans e s s'))).
```

We then lift (`enabled e`) to groups of actions giving rise to another state predicate (`group_enabled E`) which is satisfied iff the group `E` (i.e. at least one action from this group) is enabled.

```
Definition group_enabled := [E:(set act)][s:st]
  (ExT [e:act](In act E e) /\ (enabled e s)).
```

We next introduce `sco`, the strong version of `co`, which takes enabledness into account. The assertion (`sco E p q`) means not only that in `E` the successor state of a state satisfying `p` is a state satisfying `q`, but it also states that `E` seen as a group of actions is enabled in every state satisfying `p`.

```
Definition en := [E:(set act)][p:s_prop]
  (everywhere (s_imp p (group_enabled E))).
```

```
Definition sco := [E:(set act)][p,q:s_prop]
  (en E p) /\ (co E p q) .
```

These definitions have a number of simple consequences:

```
Theorem sco_false :
  (E:(set act))(q:s_prop)(sco E s_false q).
```

Theorem `en_implication` :
 $(E:(\text{set act}))(p,q:s_prop)$
 $(\text{everywhere } (s_imp\ p\ q)) \rightarrow (en\ E\ q) \rightarrow (en\ E\ p).$

Theorem `sco_implication` :
 $(E:(\text{set act}))(p,p',q,q':s_prop)$
 $(\text{everywhere } (s_imp\ p\ p')) \rightarrow$
 $(\text{everywhere } (s_imp\ q\ q')) \rightarrow$
 $(sco\ E\ p'\ q) \rightarrow (sco\ E\ p\ q').$

Theorem `co_sco_conjunction` : $(E:\text{view})$
 $(p,q:s_prop)(co\ E\ p\ q) \rightarrow$
 $(p',q':s_prop)(sco\ E\ p'\ q') \rightarrow$
 $(sco\ E\ (s_and\ p\ p')\ (s_and\ q\ q')).$

Theorem `stable_sco_conjunction` : $(E:\text{view})(E':\text{view})$
 $(Included\ act\ E'\ E) \rightarrow$
 $(r:s_prop)(stable\ E\ r) \rightarrow$
 $(p,q:s_prop)(sco\ E'\ p\ q) \rightarrow$
 $(sco\ E'\ (s_and\ r\ p)\ (s_and\ r\ q)).$

Now we are prepared to give the formal definition of `transient`, which states that the assertion $(\text{transient } E\ p)$ is satisfied iff the given view E includes a weakly fair group E' of actions such that for every state satisfying p each successor state obtained by executing an action from E' invalidates p .

Definition `transient` := $[E:\text{view}][p:s_prop]$
 $(\text{ExT } [E':(\text{set act})]$
 $(In\ (\text{set act})\ wf\ E') \wedge (Included\ act\ E'\ E) \wedge$
 $(sco\ E'\ p\ (s_not\ p))).$

Some simple theorems about `transient` are given next.

Theorem `transient_implication` : $(E:\text{view})(p,q:s_prop)$
 $(\text{transient } E\ p) \rightarrow (\text{everywhere } (s_imp\ q\ p)) \rightarrow$
 $(\text{transient } E\ q).$

Theorem `transient_false` : $(E:\text{view})(\text{transient } E\ s_false).$

The previous theorem `transient_false` is reduced to `sco_false` by exploiting our convention that the empty group is weakly fair, as expressed by the axiom

`wf_contains_empty_set` in the module `abstract`. In fact, this is the only place where this axiom is used.

The following rule allows us to combine a `stable` and a `transient` assertion. Being stable and being transient are contradictory requirements for a state predicate, except for the trivial case where it is never satisfied.

```
Theorem stable_and_transient : (E:view)(p:s_prop)
  (stable E p) -> (transient E p) ->
  (everywhere (s_not p)).
```

Again we have proof rules to derive relativized `sco` and `transient` assertions:

```
Theorem rel_sco : (E:view)(E':view)
  (Included act E' E)->
  (r:s_prop)(stable E r)->(p,q:s_prop)
  (sco E' (s_and r p) (s_imp r q))->
  (sco E' (s_and r p) (s_and r q)).
```

```
Theorem rel_transient : (E:view)(p,r:s_prop)
  (stable E r)->
  (ExT [E':view]
    (In (set act) wf E')/\(Included act E' E)/\
    (sco E' (s_and r p) (s_imp r (s_not p))))->
  (transient E (s_and r p)).
```

The `ensures` assertion is essentially `unless` equipped with liveness. Remember the definition of `(unless E p q)` as `(co E (s_and p (s_not q)) (s_or p q))` stating that `(s_and p (s_not q))` implies that `(s_or p q)` holds in the next state in `E`. To ensure liveness we just have to add `(transient E (s_and p (s_not q)))` expressing that `(s_and p (s_not q))` cannot hold permanently in `E`. This excludes the possibility that `p` holds forever and that `q` never becomes true.

```
Definition ensures := [E:view] [p,q:s_prop]
  (unless E p q) /\ (transient E (s_and p (s_not q))).
```

From `unless_true` and `transient_false` we obtain immediately:

```
Theorem ensures_true : (E:view)(p:s_prop)
  (ensures E p s_true).
```

In analogy to `unless_imp`, any implication gives rise to an `ensures` assertion.

Theorem `ensures_imp` : (E:view)
 (p,q:s_prop)(everywhere (s_imp p q)) -> (ensures E p q).

In an assertion (`ensures E p s_false`) the state predicate `p` can never be true, otherwise `s_false` would have to become true eventually which is impossible.

Theorem `ensures_false` : (E:view)
 (p:s_prop)(ensures E p s_false)->(everywhere (s_not p)).

In analogy to `unless_implication_r` we have a weakening rule for the right hand side of an `ensures` assertion.

Theorem `ensures_implication_r` : (E:view)(p,q,r:s_prop)
 (everywhere (s_imp q r)) ->
 (ensures E p q) -> (ensures E p r).

The following conjunction rule for `ensures` has a form similar to the `unless` conjunction rule. Notice, however, that only one premise has to be a liveness assertion in order to infer liveness.

Theorem `ensures_conjunction` : (E:view)(p,p',q,q':s_prop)
 (unless E p q) -> (ensures E p' q') ->
 (ensures E (s_and p p'))
 (s_or3 (s_and p q') (s_and p' q) (s_and q q'))).

Theorem `ensures_stable_conjunction` : (E:view)
 (p,q,r:s_prop)(stable E r)->(ensures E p q)->
 (ensures E (s_and p r) (s_and q r)).

Finally, we have the usual rule to derive relativized `ensures` assertions.

Theorem `rel_ensures` : (E:view)(p,q,r:s_prop)
 (stable E r)->
 (unless E (s_and r p) (s_imp r q))->
 (transient E (s_and r (s_and p (s_not q))))->
 (ensures E (s_and r p) (s_and r q)).

The `leads_to` operator is defined inductively on top of `ensures`. In fact, we define `leads_to` as the smallest predicate satisfying the three properties `leads_to_basis`, `leads_to_transitivity` and `leads_to_disjunction`. Remember that the assertion (`leads_to E p q`) should entail that if `p` holds in an arbitrary state in `E` then `q` will hold eventually in the future. This is obviously a weaker property

than `(ensures E p q)`, justifying the implication `leads_to_basis` in the inductive definition below. Furthermore, transitivity is clearly a property of `leads_to`, as required by the implication `leads_to_transitivity`. Finally, the implication `leads_to_disjunction` formalizes a disjunction property: If `(leads_to E (P i) q)` for a family `P` of state predicates, then we can conclude `(leads_to E (s_ex T P) q)`. The justification is straightforward: Given a state `s` satisfying the disjunction `(s_ex T P)` there must be some index `i` such that `(P i)` is true at `s`. Now we can use the premise `(leads_to E (P i) q)` to infer that `q` will hold eventually.

```

Inductive leads_to [E:view] : s_prop->s_prop->Prop :=
  leads_to_basis : (p,q:s_prop)
    (ensures E p q) -> (leads_to E p q)
| leads_to_transitivity : (p,q,r:s_prop)
  (leads_to E p q) -> (leads_to E q r) ->
  (leads_to E p r)
| leads_to_disjunction : (T:Type)(P:T->s_prop)(q:s_prop)
  ((i:T)(leads_to E (P i) q)) ->
  (leads_to E (s_ex T P) q).

```

The induction principle associated with this inductive definition of `leads_to` can be formulated as follows:

```

Theorem leads_to_induction :
  (IH:view->s_prop->s_prop->Prop)(E:view)
  ((p,q:s_prop)(ensures E p q)->(IH E p q))->
  ((p,q,r:s_prop)
    ((leads_to E p q)->(leads_to E q r)->
      (IH E p q)->(IH E q r)->(IH E p r)))->
  ((T:Type)(P:T->s_prop)(p,q:s_prop)
    ((i:T)(leads_to E (P i) q))->((i:T)(IH E (P i) q))->
      (IH E (s_ex T P) q))->
  ((p,q:s_prop)
    (leads_to E p q)->(IH E p q)).

```

We now continue with proof rules for `leads_to` assertions, starting with an immediate consequence of `ensures_true`:

```

Theorem leads_to_true : (E:view)
  (p:s_prop)(leads_to E p s_true).

```

An immediate consequence of `ensures_imp` is that every implication gives rise to a `leads_to` assertion. We also give two specializations below.

Theorem `leads_to_imp` : (E:view)
 (p,q:s_prop)(everywhere (s_imp p q)) -> (leads_to E p q).

Theorem `leads_to_refl` : (E:view)
 (p:s_prop)(leads_to E p p).

Theorem `leads_to_false` : (E:view)
 (p:s_prop)(leads_to E s_false p).

Using `leads_to_imp` and `leads_to_transitivity` we can prove that an assertion `leads_to` can be strengthened on the left hand side and weakened on the right hand side as stated in the following theorem.

Theorem `leads_to_implication` : (E:view)
 (p,p',q,q':s_prop)
 (everywhere (s_imp p p'))->
 (everywhere (s_imp q q'))->
 (leads_to E p' q)-> (leads_to E p q').

We also prove a general disjunction rule for arbitrary families of `leads_to` assertions. It is in fact a generalization of the `leads_to_disjunction` rule of the inductive definition.

Theorem `leads_to_big_disjunction` : (E:view)
 (T:Type)(P,Q:T->s_prop)
 ((i:T)(leads_to E (P i) (Q i)))->
 (leads_to E (s_ex T P) (s_ex T Q)).

As a special case, we obtain a binary disjunction rule.

Theorem `leads_to_small_disjunction` :
 (E:view)(p,q,p',q':s_prop)
 (leads_to E p p') -> (leads_to E q q')->
 (leads_to E (s_or p q) (s_or p' q')).

The following cancelation rule is very useful in practice. Assume that: (1) `p` leads to `q` or `b`, and (2) `b` leads to `r`. Then `p` leads either to a state satisfying `q`, or if this is not the case, it must lead to a state satisfying `b`, which in turn leads to `r`. So we can conclude that `p` leads to `q` or `r`.

Theorem `leads_to_cancelation` : (E:view)(p,q,b,r:s_prop)
 (leads_to E p (s_or q b))->(leads_to E b r)->
 (leads_to E p (s_or q r)).

A very powerful rule is the following progress-safety-progress rule `leads_to_psp`, which combines a liveness assertion and a safety assertion and allows us to infer a new liveness assertion. The formal metalogical proof is somewhat tedious, but the semantic justification is intuitive: Assume (1) `p` leads to `q` and that (2) `r`, once it is true, cannot become false before `b` becomes true. Now there are two possibilities in a state satisfying `p` and `r`: Either `b` remains false forever, or it becomes true eventually. In the first case `p` and `r` leads to `q` and `r`, since `r` remains true. In the second case `p` and `r` leads to `b` trivially.

The progress-safety-progress rule for `leads_to` will be proved by induction over the `leads_to` premise. We first prove the progress-safety-progress rule for `ensures`, which is the base case of the inductive definition of `leads_to`.

```
Theorem ensures_psp : (E:view)
  (p,q,r,b:s_prop)(ensures E p q)->(unless E r b)->
  (ensures E (s_and p r) (s_or (s_and q r) b)).
```

```
Theorem leads_to_psp : (E:view)
  (p,q,r,b:s_prop)(leads_to E p q)->(unless E r b)->
  (leads_to E (s_and p r) (s_or (s_and q r) b)).
```

Another rule to combine safety and liveness assertions is the following: A `leads_to` assertion can be strengthened on both sides by a state predicate which is known to be stable.

```
Theorem leads_to_stable_conjunction : (E:view)
  (p,q,r:s_prop)(stable E r)->(leads_to E p q)->
  (leads_to E (s_and p r) (s_and q r)).
```

Using `leads_to_stable_conjunction` we can prove `rel_leads_to`, which completes our list of proof rules for relativized assertions.

```
Theorem rel_leads_to : (E:view)(p,q,r:s_prop)
  (stable E r)->
  (leads_to E (s_and r p) (s_imp r q))->
  (leads_to E (s_and r p) (s_and r q)).
```

We now present an alternative characterization of `leads_to` by means of the operator `strong_leads_to` introduced by the inductive definition below. The only difference w.r.t. the inductive definition of `leads_to` is that in the context of the other requirements transitivity is expressed by

```
(ensures E p q)->(strong_leads_to E q r) ->(strong_leads_to E p r)
```

which in isolation is weaker than

$$(\text{leads_to } E \text{ p } q) \rightarrow (\text{leads_to } E \text{ q } r) \rightarrow (\text{leads_to } E \text{ p } r) | .$$

The advantage of `strong_leads_to` is that a stronger induction principle is available, due to the stronger premise (`ensures E p q`) instead of (`leads_to E p q`) in this implication.

```
Inductive strong_leads_to [E:view] : s_prop->s_prop->Prop :=
  strong_leads_to_basis : (p,q:s_prop)
    (ensures E p q) -> (strong_leads_to E p q)
| strong_leads_to_transitivity : (p,q,r:s_prop)
    (ensures E p q) -> (strong_leads_to E q r) ->
    (strong_leads_to E p r)
| strong_leads_to_disjunction : (T:Type)(P:T->s_prop)(q:s_prop)
    ((i:T)(strong_leads_to E (P i) q)) ->
    (strong_leads_to E (s_ex T P) q).
```

The equivalence between `strong_leads_to` and `leads_to` is stated by the following two implications.

```
Theorem strong_leads_to_imp_leads_to : (E:view)
  (p,q:s_prop)(strong_leads_to E p q) -> (leads_to E p q).
```

```
Theorem leads_to_imp_strong_leads_to : (E:view)
  (p,q:s_prop)(leads_to E p q) -> (strong_leads_to E p q).
```

As a consequence, the stronger induction principle of `strong_leads_to` is inherited by `leads_to`. It is formulated in the following theorem.

```
Theorem leads_to_strong_induction :
  (IH:view->s_prop->s_prop->Prop) (E:view)
  ((p,q:s_prop)(ensures E p q) -> (IH E p q)) ->
  ((p,q,r:s_prop)
    ((ensures E p q) -> (leads_to E q r) ->
      (IH E p q) -> (IH E q r) -> (IH E p r))) ->
  ((T:Type)(P:T->s_prop)(p,q:s_prop)
    ((i:T)(leads_to E (P i) q)) -> ((i:T)(IH E (P i) q)) ->
      (IH E (s_ex T P) q)) ->
  ((p,q:s_prop)
    (leads_to E p q) -> (IH E p q)).
```

With this induction principle we can prove statements of the form

$((p, q : s_prop) (leads_to\ E\ p\ q) \rightarrow (IH\ E\ p\ q)).$

Later, however, we will encounter a situation where we need to prove a statement with two `leads_to` premises, more precisely a statement of the form

$((p, q, p', q' : s_prop)$
 $(leads_to\ E\ p\ q) \rightarrow (leads_to\ E\ p'\ q') \rightarrow$
 $(IH\ E\ p\ q\ p'\ q')).$

To deal with goals like this it is in general convenient to introduce the following double induction principle. The proof is done by nested application of `leads_to_strong_induction`.

Theorem `leads_to_double_induction` :

$(IH : view \rightarrow s_prop \rightarrow s_prop \rightarrow s_prop \rightarrow Prop) (E : view)$
 $((p, q, p', q' : s_prop)$
 $(ensures\ E\ p\ q) \rightarrow (leads_to\ E\ p'\ q') \rightarrow$
 $(IH\ E\ p\ q\ p'\ q')) \rightarrow$
 $((p, q, p', q' : s_prop)$
 $(leads_to\ E\ p\ q) \rightarrow (ensures\ E\ p'\ q') \rightarrow$
 $(IH\ E\ p\ q\ p'\ q')) \rightarrow$
 $((p, q, r, p', q', r' : s_prop)$
 $((ensures\ E\ p\ q) \rightarrow (leads_to\ E\ q\ r) \rightarrow$
 $(ensures\ E\ p'\ q') \rightarrow (leads_to\ E\ q'\ r') \rightarrow$
 $(IH\ E\ p\ q\ p'\ q') \rightarrow$
 $(IH\ E\ p\ q\ q'\ r') \rightarrow (IH\ E\ q\ r\ p'\ q') \rightarrow$
 $(IH\ E\ q\ r\ q'\ r') \rightarrow$
 $(IH\ E\ p\ r\ q'\ r') \rightarrow (IH\ E\ q\ r\ p'\ r') \rightarrow$
 $(IH\ E\ p\ r\ p'\ r')) \rightarrow$
 $((T : Type) (P' : T \rightarrow s_prop) (p, q, p', q' : s_prop)$
 $(leads_to\ E\ p\ q) \rightarrow$
 $((i : T) (leads_to\ E\ (P'\ i)\ q')) \rightarrow$
 $((i : T) (IH\ E\ p\ q\ (P'\ i)\ q')) \rightarrow$
 $(IH\ E\ p\ q\ (s_ex\ T\ P')\ q')) \rightarrow$
 $((T : Type) (P : T \rightarrow s_prop) (p, q, p', q' : s_prop)$
 $((i : T) (leads_to\ E\ (P\ i)\ q)) \rightarrow$
 $(leads_to\ E\ p'\ q') \rightarrow$
 $((i : T) (IH\ E\ (P\ i)\ q\ p'\ q')) \rightarrow$
 $(IH\ E\ (s_ex\ T\ P)\ q\ p'\ q')) \rightarrow$
 $((p, q, p', q' : s_prop)$
 $(leads_to\ E\ p\ q) \rightarrow (leads_to\ E\ p'\ q') \rightarrow$
 $(IH\ E\ p\ q\ p'\ q')).$

The double induction principle is more general than necessary if the formula $(IH\ E\ p\ q\ p'\ q')$ is logically symmetric w.r.t. to replacement of (p,q) by (p',q') . Actually, this will be the case in our application, which is the proof of the completion theorem. In the presence of such a symmetry `leads_to_double_induction` can be simplified to `leads_to_symmetric_double_induction`, reducing the number of premises needed to obtain the same goal.

Theorem `leads_to_symmetric_double_induction` :

```
(IH:view->s_prop->s_prop->s_prop->s_prop->Prop)
(E:view)
((p,q,p',q':s_prop)(IH E p q p' q')->
 (IH E p' q' p q))->
((p,q,p',q':s_prop)
 (ensures E p q)->(leads_to E p' q')->
 (IH E p q p' q'))->
((p,q,r,p',q',r':s_prop)
 ((ensures E p q)->(leads_to E q r)->
 (ensures E p' q')->(leads_to E q' r')->
 (IH E p q p' q')->
 (IH E p q q' r')->(IH E q r p' q')->
 (IH E q r q' r')->
 (IH E p r q' r')->(IH E q r p' r')->
 (IH E p r p' r'))->
((T:Type)(P:T->s_prop)(p,q,p',q':s_prop)
 ((i:T)(leads_to E (P i) q))->
 (leads_to E p' q')->
 ((i:T)(IH E (P i) q p' q'))->
 (IH E (s_ex T P) q p' q'))->
((p,q,p',q':s_prop)
 (leads_to E p q)->(leads_to E p' q')->
 (IH E p q p' q')).
```

Our next major goal is to prove the general completion rule given later by the theorem `leads_to_completion`, a rule that is very useful in practice. We will proceed using various intermediate statements, but first we will give an informal explanation. Basically the completion rule combines several `leads_to` assertions into a single one in a conjunctive fashion. Of course, we cannot expect to derive the usual conjunction rule for a family of `leads_to` assertions in the strict sense, since their right hand sides do not necessarily become true simultaneously. However, under some additional restrictions we can approximate this idea: Assume a finite family of assertions $(leads_to\ E\ (P\ i)\ (Q\ i))$ and assume we know that $(unless\ E\ (Q\ i)\ r)$, i.e. the states satisfying $(Q\ i)$ can only be left via r . Then we can conclude that $(s_all\ T\ P)$ leads to $(s_all\ T\ Q)$

or r . The disjunction on the right hand side corresponds to the two possible cases: Either all $(Q\ i)$ will simultaneously become true at some time, or if this is not the case, at least one of the state predicates $(Q\ i)$ will become true and then false again. But in this case r will become **true**. Actually, the Theorem `leads_to_completion` given later is a little bit more liberal: it has weaker premises `(leads_to E (P i) (s_or (Q i) r))`, because a situation where r becomes true immediately satisfies the right hand side of our `leads_to` conclusion.

The proof in [CM88] is rather informal, since it relies on induction over the length of proofs which are themselves not formalized. The length of proofs remains a rather vague notion, in particular in view of the infinitary inductive definition of `leads_to`. Here we adopted a different approach: We will employ the double induction principle for `leads_to`, more precisely its symmetric specialization `leads_to_symmetric_double_induction`, which has been proved before. Using this induction principle we prove `leads_to_simple_conjunction`, and from this we obtain the binary completion theorem `leads_to_simple_completion`. Finally, we prove the general completion theorem `leads_to_completion` by another induction using an intermediate result `leads_to_completion_nat`.

First we prove the base case of `leads_to_simple_conjunction` where both `leads_to` premises are actually `ensures` assertions.

```
Lemma leads_to_simple_conjunction_basis : (E:view)
  (p,q,p',q',r:s_prop)
  (ensures E p q)->
  (leads_to E p' q')->
  (unless E q r)->
  (unless E q' r)->
  (leads_to E (s_and p p') (s_or (s_and q q') r)).
```

Next we prove `leads_to_simple_conjunction`. This theorem is obviously symmetric w.r.t. to replacement of (p,q) by (p',q') , which allows us to prove it using `leads_to_symmetric_double_induction`.

```
Theorem leads_to_simple_conjunction : (E:view)
  (p,q,p',q',r:s_prop)
  (leads_to E p q)->
  (leads_to E p' q')->
  (unless E q r)->
  (unless E q' r)->
  (leads_to E (s_and p p') (s_or (s_and q q') r)).
```

By application of `unless_simple_disjunction` and `leads_to_simple_`

conjunction to the previous result we obtain the binary version of the completion theorem which has two `leads_to` premises.

```
Theorem leads_to_simple_completion : (E:view)
  (p,q,p',q',r:s_prop)
  (leads_to E p (s_or q r))->
  (leads_to E p' (s_or q' r))->
  (unless E q r)->
  (unless E q' r)->
  (leads_to E (s_and p p') (s_or (s_and q q') r)).
```

The general completion theorem `leads_to_completion` which has an arbitrary finite family of `leads_to` premises can be reduced to the following special case, where the family is indexed by an initial interval of the natural numbers. The proof is done by induction over the bound `b`, using the previous result for the binary case.

```
Theorem leads_to_completion_nat : (E:view)
  (b:nat)(P,Q:nat->s_prop)(r:s_prop)
  ((i:nat)(lt i b)->(leads_to E (P i) (s_or (Q i) r)))->
  ((i:nat)(lt i b)->(unless E (Q i) r))->
  (leads_to E (s_all_nat b P) (s_or (s_all_nat b Q) r)).
```

Now the proof of the general completion theorem `leads_to_completion` follows by exploiting the definition of `(finite T)`, which expresses that `T` is of finite cardinality.

```
Theorem leads_to_completion : (E:view)
  (T:Type)(finite T)->(P,Q:T->s_prop)(r:s_prop)
  ((i:T)(leads_to E (P i) (s_or (Q i) r)))->
  ((i:T)(unless E (Q i) r))->
  (leads_to E (s_all T P) (s_or (s_all T Q) r)).
```

In addition to our previous characterization of `leads_to` by `strong_leads_to`, we now give another characterization by means of `leads_to_pre`, which is again defined inductively in a way very similar to the definition of `strong_leads_to`. The only difference is that `(strong_leads_to E)` is an inductively defined *binary relation* on state predicates, whereas `(leads_to_pre E r)` is an inductively defined *set* of state predicates. The intention is that `(leads_to_pre E r)` contains those state predicates which are `strong_leads_to` preconditions of some fixed state predicate `r`.

```

Inductive leads_to_pre [E:view;r:s_prop] : s_prop->Prop :=
  leads_to_pre_basis : (p:s_prop)
    (ensures E p r) -> (leads_to_pre E r p)
| leads_to_pre_transitivity : (p,q:s_prop)
    (ensures E p q) -> (leads_to_pre E r q) ->
    (leads_to_pre E r p)
| leads_to_pre_disjunction : (T:Type)(P:T->s_prop)
    ((i:T)(leads_to_pre E r (P i))) ->
    (leads_to_pre E r (s_ex T P)).

```

Indeed, $(\text{leads_to_pre } E \ q \ p)$ is equivalent to $(\text{strong_leads_to } E \ p \ q)$ and hence equivalent to $(\text{leads_to } E \ p \ q)$ as it is proved in the following two theorems. The main reason to set up different characterizations of `leads_to` is that they all give rise to different induction principles. Furthermore, the definition as an inductive set is structurally simpler than the definitions of `leads_to` and `strong_leads_to` given before. It shows that the definition of `strong_leads_to` can be cast into a form which is a parameterized inductive definition (`E` and `r` are parameters) of a set of state predicates.

```

Theorem leads_to_pre_imp_leads_to : (E:view)(p,q:s_prop)
  (leads_to_pre E q p) -> (leads_to E p q).

```

```

Theorem leads_to_imp_leads_to_pre : (E:view)(p,q:s_prop)
  (leads_to E p q) -> (leads_to_pre E q p).

```

Given a state predicate `p` that is satisfied for some state, the assertion $(\text{leads_to } E \ p \ \text{s_false})$ cannot hold; otherwise a contradiction arises, since `s_false` would have to become true eventually. The proof is done by induction on the `leads_to` premise, using the induction principle provided by `leads_to_pre`.

```

Theorem leads_to_impossible : (E:view)
  (p:s_prop)(leads_to E p s_false)->(everywhere (s_not p)).

```

Finally, we turn to another family of proof rules addressing the fact that induction is the most important proof technique for `leads_to` assertions. In contrast to structural induction over `leads_to` assertions, that we used before as a metalogical tool to derive a number of generic proof rules, we will now discuss object-level induction rules, which are typically employed to prove a particular fixed `leads_to` assertion in a concrete system. We begin with a general proof rule for well-founded induction, and then give a typical specialization for induction over the type `nat` of natural numbers. The general well-founded induction rule `leads_to_induction` assumes that the induction is carried out over a state

function `variant` mapping states into some domain, assuming a well-founded ordering `less` on this domain which is not necessarily total. For this purpose we use the predicate `well_founded` from the module `WfT` from the mathematical library. The goal of the subsequent proof rule is that `p` leads to `q`. To guarantee this assertion we use the following condition as a premise: If `p` holds and the value of `variant` equals `m` then there are two possibilities: either (1) we will eventually reach a state satisfying `q`, in this case the conclusion is immediately justified, or (2) we will eventually reach a state satisfying `p` again, but this time with the value of `variant` being decreased w.r.t. `m`. Obviously, we are concerned with a loop where choice (2) can be taken only finitely often, due to the well-foundedness condition. So choice (1) has to be taken eventually, and `q` will then be satisfied.

```
Theorem leads_to_induction : (E:view)
  (T:Type) (less:T->T->Prop) (variant:st->T)
  (WfT_well_founded T less)->
  (p,q:s_prop)
  ((m:T)
    (leads_to E
      (s_and p [s:st] (variant s) == m)
      (s_or (s_and p [s:st] (less (variant s) m)) q))))->
  (leads_to E p q).
```

Now we instantiate `less` by `lt`, the usual well-founded strict total order on `nat`, and obtain the following induction rule for natural numbers as a special case.

```
Theorem leads_to_nat_induction : (E:view)
  (variant:st->nat)
  (p,q:s_prop)
  ((m:nat)
    (leads_to E
      (s_and p [s:st] (variant s) == m)
      (s_or (s_and p [s:st] (lt (variant s) m)) q))))->
  (leads_to E p q).
```

Supplementing the previous induction principle, where the value decreases during the execution, we provide a reverse induction principle, where the variant increases. In this case we have to require some upper bound for the variant, which is specified by `maximum`. In the proof we define a decreasing variant `[s:st](minus maximum (variant s))` and use the previous induction rule for natural numbers.

```
Theorem leads_to_rev_nat_induction : (E:view)
  (variant:st->nat) (maximum:nat) (p,q:s_prop)
```

```

(everywhere (s_imp p [s:st] (le (variant s) maximum))) ->
((m:nat) (le m maximum) ->
  (leads_to E
    (s_and p [s:st] m = (variant s))
    (s_or (s_and p [s:st] (lt m (variant s))) q))) ->
(leads_to E p q).

```

4.3.6 System Composition

The elementary compositionality results formalized in the module `composition` to be discussed next can be used to achieve a certain degree of modularity and reusability in system verification. This is in particular of interest for large systems which either can be decomposed into meaningful subsystems and/or are build anyway from (reusable) components. Compositional verification means that in order to prove a system property we rely on the specifications of its components, abstracting from their internal structure as far as possible. The assumption that a component satisfies a specification may again be verified by a temporal logic proof, but may also be confirmed by other formal or informal methods as it is often the case in practice.

As we have already explained before, we are working in the context of a single transition system which represents the entire system, and we regard components of this system as particular views. The natural composition operation then becomes a simple union of these views, i.e. a union of sets. A distinguishing feature of a UNITY-style temporal logic is that almost all of its operators enjoy elementary compositionality properties which are presented in this section.

```

Theorem co_union : (E,E':view)(p,q:s_prop)
  (co E p q) -> (co E' p q) ->
  (co (Union act E E') p q).

```

```

Theorem stable_union : (E,E':view)(p:s_prop)
  (stable E p) -> (stable E' p) ->
  (stable (Union act E E') p).

```

```

Theorem indinv_stable_union :
  (E,E':view)(IC:s_prop)(p:s_prop)
  (ind_invariant E IC p) -> (stable E' p) ->
  (ind_invariant (Union act E E') IC p).

```

```

Theorem stable_indinv_union :
  (E,E':view)(IC:s_prop)(p:s_prop)
  (stable E' p) -> (ind_invariant E IC p) ->
  (ind_invariant (Union act E E') IC p).

```

Theorem `unless_union` : $(E, E' : \text{view})(p, q : \text{s_prop})$
 $(\text{unless } E \text{ } p \text{ } q) \rightarrow (\text{unless } E' \text{ } p \text{ } q) \rightarrow$
 $(\text{unless } (\text{Union act } E \text{ } E') \text{ } p \text{ } q).$

Theorem `unless_stable_union` : $(E, E' : \text{view})(p, q : \text{s_prop})$
 $(\text{unless } E \text{ } p \text{ } q) \rightarrow (\text{stable } E' \text{ } p) \rightarrow$
 $(\text{unless } (\text{Union act } E \text{ } E') \text{ } p \text{ } q).$

Theorem `en_union_r` : $(E, E' : \text{view})(p : \text{s_prop})$
 $(\text{en } E' \text{ } p) \rightarrow (\text{en } (\text{Union act } E \text{ } E') \text{ } p).$

Theorem `co_sco_union` : $(E, E' : \text{view})(p, q : \text{s_prop})$
 $(\text{co } E \text{ } p \text{ } q) \rightarrow (\text{sco } E' \text{ } p \text{ } q) \rightarrow$
 $(\text{sco } (\text{Union act } E \text{ } E') \text{ } p \text{ } q).$

Theorem `transient_union_r` : $(E, E' : \text{view})(p : \text{s_prop})$
 $(\text{transient } E' \text{ } p) \rightarrow$
 $(\text{transient } (\text{Union act } E \text{ } E') \text{ } p).$

Theorem `transient_union` : $(E, E' : \text{view})(p : \text{s_prop})$
 $((\text{transient } E \text{ } p) \ \backslash / \ (\text{transient } E' \text{ } p)) \rightarrow$
 $(\text{transient } (\text{Union act } E \text{ } E') \text{ } p).$

Theorem `ensures_unless_union` : $(E, E' : \text{view})(p, q : \text{s_prop})$
 $(\text{ensures } E \text{ } p \text{ } q) \rightarrow (\text{unless } E' \text{ } p \text{ } q) \rightarrow$
 $(\text{ensures } (\text{Union act } E \text{ } E') \text{ } p \text{ } q).$

Theorem `unless_ensures_union` : $(E, E' : \text{view})(p, q : \text{s_prop})$
 $(\text{unless } E \text{ } p \text{ } q) \rightarrow (\text{ensures } E' \text{ } p \text{ } q) \rightarrow$
 $(\text{ensures } (\text{Union act } E \text{ } E') \text{ } p \text{ } q).$

Theorem `ensures_stable_union` : $(E, E' : \text{view})(p, q : \text{s_prop})$
 $(\text{ensures } E \text{ } p \text{ } q) \rightarrow (\text{stable } E' \text{ } p) \rightarrow$
 $(\text{ensures } (\text{Union act } E \text{ } E') \text{ } p \text{ } q).$

Theorem `stable_ensures_union` : $(E, E' : \text{view})(p, q : \text{s_prop})$
 $(\text{stable } E \text{ } p) \rightarrow (\text{ensures } E' \text{ } p \text{ } q) \rightarrow$
 $(\text{ensures } (\text{Union act } E \text{ } E') \text{ } p \text{ } q).$

These proof rules support compositional reasoning with component specifications involving `co`, `unless`, `stable`, `ind_invariant`, `transient` and `ensures`. It is

remarkable that the only operators which do not enjoy composition rules similar to those above are `invariant` and `leads_to`.

Clearly, the lack of simple compositionality rules for `leads_to` is unsatisfying, but for *tightly coupled systems* with an unrestricted notion of composition, as we use it here, we cannot expect a rule similar to `ensures_union` due to the possible interference between components. We will return to this issue in Section 4.5, where we derive a formal compositionality theorem for so-called informative `leads_to` assertions that in some sense can be regarded as the least restrictive compositionality rule of this kind.

Compositional verification of *loosely coupled systems*, i.e. systems which are interacting in a well-defined and typically restricted way, is beyond the scope of this thesis, but we think that the verification library can be extended by more restricted composition rules. As shown in [CK97] and [CC99a] there are possibilities of compositional reasoning with `leads_to` in a rely-guarantee style. Conditional assertions, which can be easily expressed as implications in the metalogic, are a natural starting point to support such kind of reasoning without modifying the temporal logic itself. In fact, [CC99a] uses conditional assertions of a particular kind, which is why we consider a formalization of this approach in our general setting of labeled transition systems as an natural future extension of our temporal logic library.

4.3.7 A State Space with Typed Variables

While the state space in the previous section and in fact most of the modules of the temporal logic library are kept as abstract as possible, we describe here an instantiation of the abstract frame to a state space of a more specific kind. In what we call a *state space with typed variables* we define a state as a function assigning values to variables which are equipped with types in CIC. A state space with variables is not only used for UNITY programs [CM88] and in other approaches such as [MP92], but it can also serve as the state space of places/transition nets or high-level Petri nets if places are regarded as variables containing natural numbers or multisets, respectively. Our state space is strictly typed, in the sense that each variable has a well defined type and type correctness of the values of a variable is enforced by the type theory. Reasoning about such state spaces in an abstract setting becomes possible here due to the use of dependent types and universes in CIC.

The following should be understood as a modification of the module `abstract` given in the previous section. The modification is obtained by replacing the abstract declaration of `st` by a more concrete definition and by adding what we describe below. To emphasize that the present module is an instantiation of `abstract` we call it `var_abstract`.

We begin by declaring a type `var` of system variables. For a particular system this type contains the names of variables used. It is important not to confuse these variables with the variables of the temporal logic which will directly be represented as variables of CIC.

```
Variable var:Type.
```

Since we need an induction principle for variables we assume finiteness of `var` by requiring that every variable `v` has an index w.r.t. to some enumeration `ith_var` which is smaller than a constant `vars`.

```
Variable vars : nat.
Variable ith_var : nat->var.
Hypothesis var_finite : (v:var)
  (Ex [i:nat] (lt i vars) /\ (ith_var i)==v).
```

Moreover, in order to define standard operations such as assignment as type-theoretic functions, we assume that equality of variables is decidable, that is that there is an operation `var_eq_dec` such that for each two variables `v` and `v'`, `(var_eq_dec v v')` returns an object of a disjoint union type which is a tagged proof of `(v==v')` or `~(v==v')`.

```
Hypothesis var_eq_dec:(v,v':var){(v==v')}{~(v==v')}.
```

We furthermore introduce a type `type_name` of type names and an operation `type` that associates a CIC type with each type name. It is a good example of the use of universes, which allow us to use CIC types in system specifications and to deal with them as first class citizens.

```
Variable type_name:Type.
Variable type:type_name->Type.
```

Now by the operation `var_type_name` we associate a type name to each variable and we define `var_type` as a convenient abbreviation to refer to the type of a variable directly. We often do not distinguish type names and their associated types in informal explanations.

```
Variable var_type_name:var->type_name.
```

```
Definition var_type := [v:var] (type (var_type_name v)).
```

Finally, we introduce `st` as the type of states by means of a definition which replaces the declaration of `st` in `abstract`. A state associates a value of the variable's type to each variable.

Definition `st := (v:var)(var_type v)`.

Notice that the dependent function space allows us to define this state space with typed variables in a concise way, which precisely corresponds to the usual set-theoretic definition. In contrast to set theory, the fact that a term denotes a state can be verified by automatic typechecking, thereby avoiding the need for an explicit proof.

At the present state of the development the temporal logic library contains the following three modules to support the notion of a state space with typed variables: The module `var_state_ops`, which is explained in the next section, introduces equality and assignment operators and related theorems for a state space with typed variables. The module `var_state_pred` formalizes the (in)dependence of a state property from a variable, provides a notion of substitution of variables, and proves some elementary theorems. The module `var_elim` demonstrates its use by proving Misra's elimination theorem, which in fact presupposes a notion of state with variables.

```
Load var_state_ops.
Load var_state_pred.
Load var_elim.
```

4.3.8 Operations on States and Variables

Assuming a state space with typed variables as introduced in the previous section, the module `var_state_ops` to be discussed below introduces extensional equality on states and provides an assignment operation to modify typed variables in a given state. While the definition of such an operation in set theory is straightforward, this is not the case in a framework with static types such as CIC.

Equality of states is extensional, i.e. it can be reduced to equality of the associated values for all variables. The proposition (`val_eq n x n' x'`) defined below means that the value `x` of type `n` is equal to the value `x'` of type `n'`. Its definition uses dependent equality `eqT_dep` from the module `EqdepT`, which is also used in the proofs of the subsequent theorems.²⁰

²⁰One should be aware of the fact that the dependent equality module `EqdepT` is not a conservative extension of CIC.

Definition `val_eq` :=

```
[n:type_name] [x:(type n)] [n':type_name] [x':(type n')]
  (eqT_dep type_name type n x n' x').
```

Obviously, the types of `x` and `x'` which will be extracted from variables, say `v` and `v'`, will depend on the types associated to `v` and `v'`, respectively. That both of these types will always be logically equal in a term `(val_eq n x n' x')` is not visible to the typechecker, which can only check computational equality. Hence, instead of the standard polymorphic equality `==`, which requires that the argument types are computationally equal, we use the more flexible dependent equality `eqT_dep`, which only requires logical equality.

Of course, standard equality implies value equality, since dependent equality `eqT_dep` is (inductively) defined to be reflexive, if both sides are instantiated by the same type.

```
Theorem eq_imp_val_eq:(n:type_name)
  (x,y:(type n))(x==y)->(val_eq n x n y).
```

The theorem `val_eq_imp_eq` is the converse of the previous implication. It corresponds to a lemma `eqT_dep_eq` from the module `EqdepT` specialized to our setting.

```
Theorem val_eq_imp_eq:(n:type_name)
  (x,y:(type n))(val_eq n x n y)->(x==y).
```

In the presence of an equality of type names `n` and `n'` it should be possible to cast an element `x` of `n` to an equal element of `n'`. This is achieved by `(cast n n' e x)` which converts a value `x` of type `n` to a value of type `n'` given an equality proof `e:n==n'`. In fact, `cast` is a particular instance of `eqT_rec` as introduced in Section 4.3.1.

```
Definition cast := [n,n':type_name] [e:(n==n')] [x:(type n)]
  (eqT_rec type_name n type x n' e).
```

By instantiating a general axiom `eq_rec_eq` from the module `EqdepT` we obtain the theorem `cast_thm`, stating that type casting between identical types does not change the value.

```
Theorem cast_thm : (n:type_name)(e:(n==n))(x:(type n))
  ((cast n n e x)==x).
```

We also prove the following theorem stating that type casting between equal types `n` and `n'` respects value equality.

```
Theorem cast_resp_eq : (n, n' : type_name) (x : (type n)) (e : (n == n'))
  (val_eq n' (cast n n' e x) n x).
```

After these preparations we come to the main part of this module. First of all we assume extensionality for states `st`, i.e. two states are identified if their values coincide at all variables.²¹

```
Axiom st_extensionality : (s, s' : st)
  ((v : var) ((s v) == (s' v))) -> (s == s').
```

The next lemma expresses the fact that equal variables have equal types. In spite of its triviality we name it explicitly in order to use it below.

```
Lemma var_eq_to_type_name_eq : (v, v' : var)
  (v == v') -> (var_type_name v) == (var_type_name v').
```

Below the assignment operation is introduced as the only operation that can modify a state. Given a state `s`, a variable `v` and a new value `x` compatible with the type of `v`, the term `(assign s v x)` denotes the new state obtained from `s` by assigning the value of `x` to `v`.

```
Definition assign := [s : st] [v : var] [x : (var_type v)]
  [v' : var] <var_type v'> Case (var_eq_dec v v') of
  (*v == v'*) [p : (v == v')] (cast (var_type_name v) (var_type_name v')
    (var_eq_to_type_name_eq v v' p) x)
  (*~(v == v')*) [_] (s v')
end.
```

Observe that the operation `var_eq_dec` deciding equality of variables and the type casting operation `cast` are used in an essential way to make `assign` properly typed. As explained above, `cast` needs an equality proof for the types involved as a justification for type casting. This equality proof is obtained by applying the previous lemma `var_eq_to_type_name_eq` to the equality proof on variables `p : (v == v')` provided by `(var_eq_dec v v')`.

The remaining theorems of this module are basic properties of the assignment operation, which are needed for most applications. The extensionality axiom `st_extensionality` is a major ingredient of their proofs.

The first theorem states that assigning the contents of a variable to itself does not change the state.

²¹This nonconservative axiom could be avoided by defining an appropriate equivalence on `st`, or more generally by equipping `st` with a categorical structure. This however would complicate proofs and would require additional theorems stating that this additional structure is respected by operations on states.

Theorem `assign_unchanged` : $(s:st)(v:var)$
 $((assign\ s\ v\ (s\ v)) == s).$

The next theorem states that a variable v which has just received the value x by assignment will contain that value x .

Theorem `assign_changed` : $(s:st)(v:var)(x:(var_type\ v))$
 $((assign\ s\ v\ x)\ v) == x).$

On the other hand, modifying a variable v does not change the value of any variable v' distinct from v .

Theorem `assign_indep` : $(s:st)(v,v':var)(x:(var_type\ v)) \sim (v==v') \rightarrow$
 $((assign\ s\ v\ x)\ v') == (s\ v').$

After repeated assignment of a value to a variable v the contents of v is the value of the last assignment.

Theorem `assign_twice` : $(s:st)(v:var)(x,y:(var_type\ v))$
 $(assign\ (assign\ s\ v\ x)\ v\ y) == (assign\ s\ v\ y).$

And finally we prove that the order of assigning values to two distinct variables does not matter. In such a situation the two assignments commute.

Theorem `assign_commutates` : $(s:st)$
 $(v,v':var)(x:(var_type\ v))(y:(var_type\ v')) \sim (v==v') \rightarrow$
 $(assign\ (assign\ s\ v\ x)\ v'\ y) == (assign\ (assign\ s\ v'\ y)\ v\ x).$

4.3.9 State Predicates and Variables

Again assuming a underlying state space with typed variables, this section presents the module `var_depend`. This module formalizes dependence and independence of state predicates on variables and introduces a (semantical) notion of variable substitution in state predicates. These concepts are needed to express some proof rules that make explicit use of variables, an example being the elimination theorem of New UNITY [Mis94, Mis95] which is subject of Section 4.3.10.

First, we introduce some technical tools for reasoning inductively about system variables. In order to refer to variables in a uniform way which is independent of the set of variables of a concrete system and their names we use the enumeration of variables provided by `ith_var` and `vars`.

The next theorem allows to infer that a property holds for all variables if it holds for all variables up to the maximum index. It uses the enumeration `ith_var` declared in `abstract`. Subsequently it will be useful to reduce proofs of properties of variables to induction over natural numbers.

```
Theorem var_index_thm : (P:var->Prop)
  ((i:nat)(lt i vars)->(P (ith_var i)))->(v:var)(P v).
```

The following auxiliary operation is used to merge two states in the following sense: Given an index n and states s and s' the term `(equalize_vars n s s')` returns a new state which is identical to s with the exception that the contents of all variables up to index n is copied from s' to s .

```
Fixpoint equalize_vars [n:nat] : st->st->st :=
  [s,s':st] Case n of
    (* 0 *) s
    (* (S p) *) [p:nat] (assign (equalize_vars p s s')
                               (ith_var p) (s' (ith_var p)))
  end.
```

```
Lemma equalize_lemma : (s,s':st)(n:nat)(i:nat)(lt i n)->
  ((equalize_vars n s s') (ith_var i))==s' (ith_var i).
```

Applying `equalize_vars` to the maximal number of variable indices `vars` gives an operation for copying a state.

```
Theorem equalize_vars_char : (s,s':st)
  (equalize_vars vars s s')==s'.
```

When reasoning about systems temporal proof rules sometimes require preconditions stating the independence of certain state predicates from particular variables. To express a simple form of such conditions the following definition of independence will be useful. Given a state predicate p and a variable v , `(independent p v)` states that p does not depend on v .

```
Definition independent := [p:s_prop][v:var]
  (s:st)(x,y:(var_type v))
  (p (assign s v x)) -> (p (assign s v y)).
```

An immediate consequence of this definition is that if p is independent of v then the fact that p holds at s is preserved under arbitrary modifications of s at v .

```
Theorem independent_imp : (p:s_prop)(v:var)
  (independent p v)->
  (s:st)(p s)->(x:(var_type v))(p (assign s v x)).
```

Using induction and the previous theorem we can prove a rather technical lemma: If a state predicate p is independent of all variables (up to some index) and p holds at some state s we can modify all these variables, e.g. using their values from another state s' , without changing the validity of p .

```
Lemma independent_and_equalize : (p:s_prop)
  ((v:var) (independent p v))->
  (s,s':st)(i:nat)(le i vars)->
  (p s)->(p (equalize_vars i s s')).
```

Using this lemma it is easy to prove that if a state predicate is independent from all variables then it does not depend on the state at all.

```
Theorem fully_independent : (p:s_prop)
  ((v:var)(independent p v))->(s,s':st)(p s)->(p s').
```

In addition to the capability of expressing independence we would also like to express preconditions stating that a predicate may depend on a particular variable but definitely does not depend on further ones. This is achieved by (`may_depend_on p v`) given a state predicate p and a variable v .

```
Definition may_depend_on := [p:s_prop] [v:var]
  (m:(var_type v))(s,s':st)
  (p (assign s v m))->(p (assign s' v m)).
```

The obvious relation between `independent` and `may_depend_on` is verified in the following theorem. It p is independent of all variables except for v' it follows that p may only depend on v' . Observe that this does not exclude the case that p is independent of v' .

```
Theorem dependent : (v':var)(p:s_prop)
  ((v:var)~(v==v'))->(independent p v)) -> (may_depend_on p v').
```

Given a predicate p there is a natural way to define a new predicate independent of an arbitrary variable v . The new predicate `[s:st] (p (assign s v m))` behaves as p except for the fact that it does not use the contents of v but a constant m instead. We denote this predicate by (`subst p v v'`), since it can be seen as a semantic counterpart of variable substitution.

```
Definition subst := [p:s_prop] [v:var] [v':(var_type v)]
  [s:st](p (assign s v v')).
```

Obviously, $(\text{subst } p \ v \ m)$ is independent from v .

```
Theorem indep_subst : (p:s_prop)(v:var)(m:(var_type v))
  (independent (subst p v m) v).
```

We also prove that independence is preserved by `subst`.

```
Theorem indep_imp_indep_subst : (p:s_prop)
  (v,v':var)(m:(var_type v))
  (independent p v')->(independent (subst p v m) v').
```

4.3.10 Elimination Theorem

As an application of the state space with typed variables and its supporting modules `var_state_ops` and `var_state_pred` we present a formalization of the elimination theorem. This theorem has been introduced in the context of New UNITY in [Mis94, Mis95], but we give it here in the more general setting of transition systems. The elimination theorem will be used in the example of Section 4.4 for the verification of safety properties.

In fact, we have chosen the elimination theorem as an example of a generic proof rule which can be used e.g. for arbitrary programs. Formalizing such proof rules and reasoning about them becomes possible inside CIC, since the notion of an abstract state space with typed variables can be expressed inside the logic as explained before.

We need two lemmas for the proof of the elimination theorem. The first one essentially states that substituting v in p by its own value $(s \ v) == m$ yields a predicate equivalent to p . Notice the use of the existential quantifier `s_ex` to introduce an auxiliary variable m local to the expression.

```
Lemma elimination_1 : (v:var)(p:s_prop)(everywhere
  (s_iff
    (s_ex (var_type v)
      ([m:(var_type v)](s_and (subst p v m) [s:st](s v)==m)))
    p)).
```

The second lemma will be proved using the following theorem, which is useful in its own right: If a predicate p is independent of all variables then it must be stable.

```
Theorem indep_imp_stable : (E:view)(p:s_prop)
  ((v':var)(independent p v'))->(stable E p).
```

The second lemma states: Given a predicate p which may depend only on a variable v , the predicate $(\text{subst } p \ v \ m)$ substituting any constant m for v is stable. The proof involves `indep_imp_stable` and `indep_imp_indep_subst`.

```
Lemma elimination_2 : (E:view)(v:var)(p:s_prop)
  ((v':var)~(v'==v)->(independent p v')) ->
  (m:(var_type v))(stable E (subst p v m)).
```

Now we are prepared to prove the elimination theorem. First we convey the intuitive idea: Assume that we have a predicate p which depends on no other variable than v . We would like to conclude that if p holds at some state, then at each successor state there is some m such that $(\text{subst } p \ v \ m)$, meaning that p holds with m substituted for v . So far this simply states that if p was true in the past there must have been some value m for v when p was true. Notice, however, that we do not know anything else about m . Now assume we have the additional information that when the variable v contains m then the predicate $(q \ m)$ holds in the next state. Then we can strengthen our conclusion: If p holds at some state then at each successor state there is some m such that $(\text{subst } p \ v \ m)$ and also $(q \ m)$ holds.

```
Theorem elimination :
  (E:view)(v:var)(p:s_prop)(q:(var_type v)->s_prop)
  ((v':var)~(v'==v)->(independent p v')) ->
  ((m:(var_type v))(co E [s:st]((s v)==m) (q m))) ->
  (co E p (s_ex ? [m:(var_type v)]
    (s_and (subst p v m) (q m)))).
```

The proof uses `co_big_disjunction`, `co_conjunction` and the previous two lemmas.

4.4 A Toy Example in Program Verification

In this section we demonstrate the use of the temporal logic library for the verification of UNITY programs by means of a very simple example. We describe a formal specification and verification of the well-known “Common Meeting Time Problem”. The example will be reused in Section 4.5.1 to demonstrate the use of our compositionality results that we develop in Section 4.3.6.

What we will present in the following can be seen as a special instance of an embedding of UNITY programs into CIC via their transition system semantics. From the viewpoint of type theory the most natural concept that could be used to represent a UNITY statement is that of an operation which can be effectively

executed. To exploit the power and elegance of type theory for the verification of functional programs it seems advantageous to view a UNITY program as a collection of communicating functional components. Each of these components can be specified and verified within type theory using standard techniques. More precisely, we represent a UNITY program as a set of functions operating on a shared state space and communicating via its variables. In fact, such representations correspond to a subclass of transitions systems that we called *functional transition systems* in [Ste98].²²

To further motivate the representation of UNITY programs that we have chosen, we would like to point out another benefit of the use of a strongly normalizing type theory such as CIC: The idea that an operational statement in the program should represent an *atomic* execution step nicely corresponds to the fact that all operations in CIC are total and terminating. So, termination proofs are implicit at the level of atomic steps. They are done by type checking and therefore do not require manual intervention. In summary, we have termination at the level of individual actions (ensured by strong normalization of CIC), but for the entire program termination is not enforced. Another important issue often related to termination is determinism. Again, we have determinism at the level of statements (ensured by confluence of CIC), reflecting the fact that each single statement has a unique result, but for the entire program (and its components) determinism is not enforced.

It should be clear that, although we use UNITY programs as an example in this section, the functional transition system representation described above is not limited to programs in the UNITY programming language, but it can be used to represent programs in other languages as well, provided that atomic statements can be expressed as functions in CIC. In fact, functional transition systems are a specialization of general transition systems that allow us to express the central concept of executability using the type-theoretic notion of a computable function.

In the following we briefly recall the common meeting time problem and a solution in terms of a simple UNITY program. Modeling time by natural numbers the problem is to find the earliest possible common meeting time for two persons. We presuppose the existence of such a common meeting time and for each of the two persons we assume a function, say f and g , respectively, such that $f(t)$ and $g(t)$ is the earliest availability time not earlier than t . We are looking for a program that has a variable `time` and satisfies the following temporal specification: (1) *Partial correctness*, i.e. `time` is approximating the common meeting, i.e. it will never decrease and will never exceed the smallest common meeting time. (2) *Total correctness*, i.e. in addition to partial correctness, `time` will eventually

²²We also would like to point out that a limitation of the approach in [Ste98], namely the requirement that a UNITY program has only a finite number of statements and therefore can be represented as a list, is not enforced any more in the present state of the library due to the use of arbitrary transition systems.

contain the smallest common meeting time. A solution satisfying this informal specification is provided by the UNITY program CMT given below. Observe that the UNITY fairness requirement is essential for total correctness.

```

Program CMT
declare time : nat
initially time = 0
assign time := f(time)
      | time := g(time)
end

```

In the remainder of this section we will demonstrate the use of the temporal logic library to: (1) give a formal representation of the program, (2) formalize its informal specification, i.e. the assumptions and the temporal properties of partial and total correctness, and (3) prove formally that the program satisfies the specification.

4.4.1 A Concrete Frame

We replace the abstract frame `var_abstract` explained in Section 4.3.7 by a concrete frame which starts with the assumptions stated in the informal explanation of the problem and a specification of the program given above. In other words, the temporal logic library is instantiated for a particular class of programs parameterized by functions `f` and `g` satisfying the assumptions stated above.

Load `init`.

It follows from the informal problem specification that we can assume two functions `f` and `g` on natural numbers which are ascending and monotone.

```

Variables f,g : nat->nat.
Axiom f_is_ascending : (m:nat)(le m (f m)).
Axiom g_is_ascending : (m:nat)(le m (g m)).
Axiom f_is_monotone : (m,n:nat)(le m n)->(le (f m) (f n)).
Axiom g_is_monotone : (m,n:nat)(le m n)->(le (g m) (g n)).

```

The concept of a common meeting time is formulated as a common fixpoint of `f` and `g`.

```

Definition common := [t:nat] (f t)==t /\ (g t)==t.

```

Next we assume that a smallest common meeting time, which we denote by `cmt`, exists, as our informal specification states.

```
Variable cmt : nat.
Axiom cmt_exists : (common cmt).
Axiom cmt_smallest : (m:nat)(common m)->(le cmt m).
```

Subsequently, we describe the concrete embedding of the UNITY program `CMT`, seen as a transition system, into `CIC`. We carry out the transformation manually, but it should be obvious that this process can be easily mechanized.

The program contains exactly one program variable `time`. It is represented by a variable name `time`, which is the only element of the following inductive type `var`. For simple programs like ours the inductive type is a simple enumeration type, but there are also applications in which other kinds of inductive types could be exploited, e.g. to represent families of program variables.

```
Inductive var : Type := time : var.
```

We also provide the enumeration function `ith_var` of `var` as required by the abstract frame together with a proof of finiteness of `var`.

```
Definition vars := (1) .
Definition ith_var := [i:nat] time.
```

```
Theorem var_finite : (v:var)
  (ExT [i:nat] (lt i vars) /\ (ith_var i)==v).
```

The abstract frame also requires a decision function for equality on `var`, which here follows directly from the induction principle for enumeration types.

```
Theorem var_eq_dec : (v,v':var){(v==v')}{~(v==v')}.
```

In addition to a variable name we need names for all types occurring in the program. Again we have `type_name` as an enumeration type with a single type name `unat` intended to represent the `CIC` type `nat` of natural numbers as specified by the function `type`.

```
Inductive type_name : Type := unat : type_name.
```

```
Definition type := [n:type_name]
  Cases n of unat => nat end.
```

The declaration of program variables is translated into the function `var_type_name` which maps `time` to its associated type name `unat`.

```
Definition var_type_name := [v:var]
  Cases v of time => unat end.
```

```
Definition var_type := [v:var] (type (var_type_name v)).
```

Now, after all variables together with their types have been fixed we define the state space `st` as in the module `var_abstract`.

```
Definition st := (v:var)(var_type v).
```

```
Load var_state_ops.
```

After setting up the state space we have to represent the statements of the UNITY program. First, we define `act` as a enumeration type containing an action for each statement.

```
Inductive act : Type :=
  update_f : act | update_g : act.
```

Then these actions are related to the statements of the program by a transition function `effect`, which specifies the effect of each action as given by the semantics of its corresponding statement. It takes the form of a case analysis over all actions.

```
Definition effect := [t:act]
  Cases t of
    update_f => [s:st] (assign s time (f (s time)))
  | update_g => [s:st] (assign s time (g (s time)))
  end.
```

On the basis of this functional transition system representation the transition relation `trans` can be directly defined:

```
Definition trans := [t:act][s,s':st]
  (s' == (effect t s)) .
```

Now the type of views is introduced as in `var_abstract`, and finally the body of the UNITY program `CMT` is represented by its set of actions, which are `update_f` and `update_g` in our case.

Definition `view` := `act->Prop`.

Definition `CMT` :=

```
(Add act (Add act (Empty_set act) update_g) update_f) .
```

Finally, we have to specify the unconditional fairness of UNITY, which is a special case of weak group fairness. The set `wf` of weakly fair groups contains by convention the empty group and in addition a singleton group for each statement.

Definition `wf` :=

```
(Add (set act) (Add (set act) (Add (set act)
  (Empty_set (set act))
  (Empty_set act))
  (Singleton act update_f))
  (Singleton act update_g)).
```

Theorem `wf_contains_empty_set` :

```
(In (set act) wf (Empty_set act)).
```

On top of our concrete program specification we include the rest of the general library and also the more specific modules supporting the state space with typed variables. By including these modules in the context established above they are instantiated for our particular application.

```
Load state_pred.
```

```
Load safety.
```

```
Load liveness.
```

```
Load var_state_pred.
```

```
Load var_elim.
```

4.4.2 Partial Correctness

Our overall strategy is to establish first safety assertions for partial correctness, and then, on top of such safety assertions, liveness assertions, that are additionally needed for total correctness, are proved. The two main theorems we prove will correspond exactly to our informal specification given before.

We start with a lemma which states that if `time` has a value `m`, then at each successor state `time` will not increase above `(f m)` or it will not increase above `(g m)`. The proof is done by unfolding `co` and then verifying the assertions for both actions `update_f` and `update_g`.

```

Lemma CMT_S1 : (m:nat)
  (co CMT ([s:st] (eq nat (s time) m))
    ([s:st] (le (s time) (f m)) \/(le (s time) (g m)))).

```

Following [Mis94, Mis95] we obtain the next lemma CMT_S2 by applying the elimination theorem on the program variable `time`. To this end, we instantiate `x`, `p` and `q` in Theorem `elimination by time`, `[s:st](common n)->(le (s time) n)` and `[m:nat][s:st](le (s time) (f m))\/(le (s time) (g m))`, respectively. Independence of `p` from variables different from `x` is trivially satisfied, as `time` is the only program variable. The remaining `co` premise is given exactly by Lemma CMT_S1.

```

Lemma CMT_S2 : (n:nat)
  (co CMT [s:st]((common n)->(le (s time) n))
    (s_ex ? [m:nat]
      [s:st]((common n)->(le m n))/\
        ((le (s time) (f m))\/(le (s time) (g m))))).

```

The next lemma states that `time`, which is advanced by the program, cannot skip any common meeting time. The proof starts with the `co` assertion of the preceding lemma and uses right hand side weakening to obtain the `stable` assertion.

```

Lemma CMT_S3 : (n:nat)
  (stable CMT
    ([s:st] (common n) -> (le (s time) n))).

```

Since `([s:st] (common n) -> (le (s time) n))` holds initially, i.e. it is implied by `[s:st](s time)=0`, we can establish an inductive invariant.

```

Lemma CMT_S4 : (n:nat)
  (ind_invariant CMT [s:st](eq nat (s time) 0)
    ([s:st] (common n) -> (le (s time) n))).

```

Substituting `cmt` for `n` we obtain Theorem CMT_5 establishing the fact that `time` cannot increase beyond the smallest common meeting time `cmt`.

```

Theorem CMT_S5 : (ind_invariant CMT
  [s:st](s time)==0 [s:st](le (s time) cmt)).

```

Finally, we prove Theorem CMT_S6, stating that `time` can only increase but never decreases. The proof is done by unfolding `co` and then by considering each of the two actions `update_f` and `update_g`, using the axioms `f_is_ascending` and `g_is_ascending`, respectively.

```
Theorem CMT_S6 : (m:nat)
  (co CMT [s:st](eq nat m (s time))
    [s:st](le m (s time)))).
```

4.4.3 Total Correctness

In order to establish total correctness we have to prove the liveness assertion stating that the smallest common meeting time `cmt` is reached eventually if the program starts in a state satisfying the initialization condition.

Formally, we wish to prove

```
(leads_to CMT ([s:st](eq nat (s time) 0))
  ([s:st](eq nat (s time) cmt)))
```

as formulated in Theorem `CMT_L14`. To this end we employ (reverse) induction on natural numbers according to Theorem `leads_to_rev_nat_induction`. We have to supply a variant which increases up to some maximum value. An obvious choice is the value of the variable `time`, together with the smallest common meeting time `cmt` as its maximum value.

```
Definition variant := [s:st](s time).
```

Below we need the following lemma `CMT_S7`, a special case obtained from Lemma `CMT_S3` by instantiating `n` with `cmt`.

```
Lemma CMT_S7 : (stable CMT [s:st](le (variant s) cmt)).
```

In the following we prepare the application of the reverse induction rule `leads_to_rev_nat_induction`, which will be used later to prove Lemma `CMT_L13`, which in turn immediately implies the final liveness property `CMT_L14`. The `leads_to` assertion in the premise of the reverse induction rule is proved in Lemma `CMT_L12` directly, as an `ensures` assertion.

We begin with the proof of the safety part of this `ensures` assertion, which is the `unless` assertion given in Lemma `CMT_S8`: If `time` is not beyond `cmt`, then there are only two possibilities for the next state: Either we reach `cmt`, or `time` increases but not beyond `cmt`. The proof unfolds `unless` and applies `co_stable_conjunction` to `CMT_S7` and `CMT_S6`. Then `co_implication` is used to obtain the goal.

```
Lemma CMT_S8 : (m:nat)
  (unless CMT
```

```

(s_and [s:st](le (s time) cmt)
  [s:st](eq nat m (s time)))
(s_or
  (s_and [s:st](le (s time) cmt)
    [s:st](lt m (s time)))
  [s:st](eq nat (s time) cmt))).

```

The liveness part of the aforementioned `ensures` assertion will be provided by the `transient` assertion in Lemma CMT_L11, which in turn will be proved from the following lemma CMT_L10 via an intermediate lemma CMT_L9 which states: If `time` is not a common meeting time, then `time` will eventually change.

```

Lemma CMT_L9 : (m:nat) (transient CMT
  [s:st](not (common (s time))) /\ (eq nat m (s time))).

```

The proof is done by unfolding `transient` and distinguishing two cases: If `m` equals `(f m)` then `update_g` is responsible for progress, otherwise `update_f` is the witness of progress.

Using `transient_implication` and the fact that `cmt` is the smallest common meeting time we obtain Lemma CMT_L10.

```

Lemma CMT_L10 : (m:nat) (transient CMT
  [s:st]((lt (s time) cmt) /\ (eq nat m (s time)))).

```

Another application of `transient_implication` together with some case analysis on `time` finally yields CMT_L11, which is exactly what we need for the proof of Lemma CMT_L12.

```

Lemma CMT_L11 : (m:nat)(transient CMT
  (s_and (s_and [s:st](le (s time) cmt)
    [s:st](eq nat m (s time)))
  (s_not
    (s_or
      (s_and [s:st](le (s time) cmt)
        [s:st](lt m (s time)))
      [s:st](eq nat (s time) cmt)))))).

```

Now we can combine safety (Lemma CMT_S8) and liveness (Lemma CMT_L11) assertions to obtain the `ensures` assertion that can be directly used to prove the following Lemma via `leads_to_basis` (see the inductive definition of `leads_to`).

```

Lemma CMT_L12 : (m:nat)
  (leads_to CMT
    (s_and [s:st](le (s time) cmt)
      [s:st](eq nat m (variant s)))
    (s_or
      (s_and [s:st](le (s time) cmt)
        [s:st](lt m (variant s)))
      [s:st](eq nat (s time) cmt))).

```

Finally, we can prove CMT_L13 using the reverse induction rule `leads_to_rev_nat_induction`. We instantiate `maximum` by `cmt`, and `variant` by `variant` as defined above. The `leads_to` premise is exactly CMT_L12. Lemma CMT_L13 states that if `time` is not beyond the smallest meeting time `cmt`, then it will reach `cmt` sometime in the future.

```

Lemma CMT_L13 : (leads_to CMT
  ([s:st](le (s time) cmt))
  ([s:st](eq nat (s time) cmt))).

```

Of course, this left hand side of `leads_to` covers specifically the initialization condition of our original program. So, we obtain our final liveness result CMT_L14 by weakening the left hand side using `leads_to_implication`.

```

Theorem CMT_L14 : (leads_to CMT
  ([s:st](eq nat (s time) 0))
  ([s:st](eq nat (s time) cmt))).

```

Together, the theorems CMT_S5, CMT_S6, and CMT_L14 prove total correctness of the program as formulated in informal specification.

4.5 Compositionality via Intensionality

Although we have simple compositionality theorems for most temporal operators, there is no counterpart of a similar form for `leads_to`. In particular, we cannot just replace `ensures` by `leads_to` in the theorem `ensures_unless_union`. Let us take a closer look at this issue: Assume we have components `E` and `E'` and state predicates `p` and `q` such that `(leads_to E p q)` and `(leads_to E' p q)` holds. Then we cannot infer in general that `(leads_to (Union act E E') p q)` holds, since the component `E` may interfere with `E'` on the execution path from `p` to `q`, e.g. by modifying variables which the progress of `E'` relies on. Obviously, such an interference can lead to new possibilities of execution that do not

lead to q . The issue of interference in parallel programs is a well-known problem and the solution proposed in [OG76] (see also the textbook presentation [AO97]) is based on a notion of *interference-free proof schemes*. This notion is rich of details and syntactic in nature. It involves a concept of programs annotated with proofs that seems to be hard to formalize in a rigorous way. In this section we propose to use the abstract notion of proof offered by the type theory itself according to the propositions-as-types/proofs-as-objects interpretation to obtain a fully satisfactory formalization of interference freeness in the general context of labeled transition systems.

Recall that we are using the logical universe `Prop` of CIC in a classical way. We have assumed classical axioms such as the law of the excluded middle, logical extensionality and proof-irrelevance, which are satisfied under an appropriate classical interpretation. Since, we are also interested in treating proofs as informative objects for the `leads_to` fragment of the temporal logic rather than assuming proof irrelevance, we move the relevant parts to the universe `Type` and we exploit the duality of views offered by the propositions-as-types interpretation, i.e. we use both, the conventional view of types as data types and the logical view of types as formulae. In other words, we will now work with *two different logics* simultaneously: (1) the impredicative logic which resides in the standard propositional universe `Prop` and is axiomatized as a classical extensional logic with proof irrelevance; and (2) the predicative logic provided by the `Type` universe hierarchy which is intuitionistic, intensional and takes proofs seriously.

The first step is to migrate from uninformative `leads_to` assertions to informative `inf_leads_to` assertions by replacing `Prop` by `Type`. We obtain the following definition:

```

Inductive inf_leads_to [E:view] : s_prop->s_prop->Type :=
  inf_leads_to_basis : (p,q:s_prop)
    (ensures E p q) -> (inf_leads_to E p q)
| inf_leads_to_transitivity : (p,q,r:s_prop)
  (inf_leads_to E p q) -> (inf_leads_to E q r) ->
  (inf_leads_to E p r)
| inf_leads_to_disjunction : (T:Type)(P:T->s_prop)(q:s_prop)
  ((i:T)(inf_leads_to E (P i) q)) ->
  (inf_leads_to E (s_ex T P) q).

```

We have formally verified that all proofs given so far remain valid if we replace `leads_to` by `inf_leads_to`. This is a consequence of the fact that the `leads_to` fragment of UNITY has been developed in an entirely constructive way relative to the base fragment of the logic. The main purpose of the new definition is to give assertions (`inf_leads_to sys p q`) in addition to their logical interpretation the status of a standard inductive data type with informative elements

which are interpreted as proofs. It is also important for us that this type resides in `Type` and is therefore not subject to the restrictions that are enforced by CIC for elimination over the uninformative universe `Prop` [PM93, BBC⁺99]. In fact, these restrictions are essential for our use of `Prop` under a classical interpretation satisfying the law of the excluded middle and proof irrelevance.

Since `inf_leads_to` is more informative than `leads_to`, we have the following theorem, which can also be seen as a casting function from informative proofs to uninformative, i.e. abstract, proofs.

```
Theorem inf_leads_to_imp_leads_to : (E:view)
  (p,q:s_prop)(inf_leads_to E p q)->(leads_to E p q).
```

Of course, the converse implication cannot be proved, due to the impossibility of generating informative objects from uninformative objects in CIC, which is consistent with our proof irrelevance axiom.

Exploiting the fact that proofs are objects we now define the compositionality property for proofs of `inf_leads_to` assertions. We define the property recursively, using the standard elimination principle `inf_leads_to_rect` that comes with `inf_leads_to`.

```
Definition compositional := [E':view] [E:view]
  (inf_leads_to_rect E
    ([u,v:s_prop] [l:(inf_leads_to E u v)] Prop)
    (* inf_leads_to_basis *)
    [p,q:s_prop] [en:(ensures E p q)]
    (unless E' p q)
    (* inf_leads_to_transitivity *)
    [p,q,r:s_prop]
    [l1:(inf_leads_to E p q)] [comp1:Prop]
    [l2:(inf_leads_to E q r)] [comp2:Prop]
    (comp1 /\ comp2)
    (* inf_leads_to_disjunction *)
    [T:Type] [P:T->s_prop] [q:s_prop]
    [L:((i:T)(inf_leads_to E (P i) q))][Comp:(i:T)Prop]
    (i:T)(Comp i)).
```

Given a proof `prf:(inf_leads_to E p' q')` the compositionality property (`compositional E' E prf`) requires that each `ensures` step in `E` is matched by a corresponding `unless` step in `E'`, a condition that expresses freedom of interference at the finest level of granularity.

The compositionality theorem that covers this case is `unless_ensures_union`. The idea of the subsequent compositionality theorem for `inf_leads_to` is to

reduce compositionality at the higher level of `inf_leads_to` to compositionality at the lower level of `ensures` using the aforementioned theorem.

In fact, the compositionality theorem for `inf_leads_to` is easy to prove by induction over the proof `prf`:

```
Theorem inf_leads_to_union : (E,E':view) (p,q:s_prop)
  (prf:(inf_leads_to E p q))
  (compositional E' E p q prf)->
  (inf_leads_to (Union act E E') p q).
```

This theorem reduces the proof of an `inf_leads_to` assertion of a system with components `E` and `E'` to a corresponding proof for one component `prf:(inf_leads_to E p q)` that satisfies the compositionality property (`compositional E' E p q prf`). Since the proof of `prf` is typically derived using the standard proof rules for `inf_leads_to` that have been given before, the task of proving the compositionality property can be considerably simplified by using theorems that state how the compositionality property behaves under application of the standard proof rules. In fact, we have proved a theorem for each proof rule that states a minimal sufficient condition to ensure compositionality of the resulting `inf_leads_to` assertions. To convey the basic idea we give a selection of these theorems below. In many cases these theorems state that proof rules preserve the compositionality property. The first three theorems correspond directly to the three cases of the inductive definition of `compositional`.

```
Theorem inf_leads_to_basis_comp :
  (E',E:view) (p,q:s_prop)
  (unless E' p q)->
  (en:(ensures E p q))
  (compositional E' E ? ?
   (inf_leads_to_basis E ? ? en)).
```

```
Theorem inf_leads_to_transitivity_comp :
  (E',E:view) (p,q,r:s_prop)
  (lto1:(inf_leads_to E p q))
  (lto2:(inf_leads_to E q r))
  (compositional E' E ? ? lto1)->
  (compositional E' E ? ? lto2)->
  (compositional E' E ? ?
   (inf_leads_to_transitivity E ? ? ? lto1 lto2)).
```

```
Theorem inf_leads_to_disjunction_comp :
  (E',E:view) (T:Type) (P:T->s_prop) (p,q:s_prop)
```

```

(lto:((i:T)(inf_leads_to E (P i) q)))
((i:T)(compositional E' E ? ? (lto i)))->
(compositional E' E ? ?
 (inf_leads_to_disjunction E T P (s_ex T P) q lto)).

```

Theorem `inf_leads_to_small_disjunction_comp` :

```

(E',E:view)(p,q,p',q':s_prop)
(lto1:(inf_leads_to E p p'))
(lto2:(inf_leads_to E q q'))
(compositional E' E ? ? lto1)->
(compositional E' E ? ? lto2)->
(compositional E' E ? ?
 (inf_leads_to_small_disjunction E p q p' q' lto1 lto2)).

```

Theorem `inf_leads_to_cancelation_comp` :

```

(E',E:view)
(p,q,r,b:s_prop)
(lto1:(inf_leads_to E p (s_or q b)))
(lto2:(inf_leads_to E b r))
(compositional E' E ? ? lto1)->
(compositional E' E ? ? lto2)->
(compositional E' E ? ?
 (inf_leads_to_cancelation E p q b r lto1 lto2)).

```

Theorem `inf_leads_to_psp_comp` :

```

(E',E:view)
(p,q,r,b:s_prop)
(lto:(inf_leads_to E p q))
(un:(unless E r b))
(compositional E' E ? ? lto)->
(unless E' r b)->
(compositional E' E ? ?
 (inf_leads_to_psp E p q r b lto un)).

```

Theorem `inf_leads_to_stable_conjunction_comp` :

```

(E',E:view)
(p,q,r:s_prop)
(st:(stable E r))
(lto:(inf_leads_to E p q))
(stable E' r)->
(compositional E' E ? ? lto)->
(compositional E' E ? ?
 (inf_leads_to_stable_conjunction E p q r st lto)).

```

```

Theorem inf_leads_to_induction_comp :
  (E',E:view)
  (T:Type)(less:T->T->Prop)(variant:st->T)
  (wf:(WfTinf_well_founded T less))
  (p,q:s_prop)
  (lto:((m:T)(inf_leads_to E
    (s_and p [s:st] (variant s) == m)
    (s_or (s_and p [s:st] (less (variant s) m)) q))))
  ((m:T)(compositional E' E ? ? (lto m)))->
  (compositional E' E ? ?
  (inf_leads_to_induction E T less variant wf p q lto)).

```

4.5.1 Extending the Example

To illustrate the use of our compositionality results in a concrete setting we extend the common meeting time example, which involves two persons represented by functions f and g , by adding another person represented by a function h .

Variables $h : \text{nat} \rightarrow \text{nat}$.

```

Axiom h_is_ascending : (m:nat)(le m (h m)).
Axiom h_is_monotone : (m,n:nat)(le m n)->(le (h m) (h n)).
Axiom h_accepts_cmt : (h cmt) = cmt .

```

```

Theorem h_rem_below_cmt : (m:nat)(le m cmt)->(le (h m) cmt) .

```

As formulated above, we assume that the new person at least agrees with the common meeting time cmt , i.e. with the common meeting time of the first two persons. This ensures that adding the new person to the existing system does not affect the properties, especially Theorem `CMT_L14`, proved earlier, although the executions of the new system can be quite different.

While preserving the original component `CMT`, we represent the additional person by an action `update_h` in the new component `CMT'`. The definitions of `act` and `effect` are extended correspondingly. It is interesting to note that there is no need to assume weak fairness for `update_h`.

```

Inductive act : Type :=
  update_f : act | update_g : act | update_h : act.

```

```

Definition effect := [e:act]
  Cases e of

```

```

    update_f => [s:st] (assign s time (f (s time)))
  | update_g => [s:st] (assign s time (g (s time)))
  | update_h => [s:st] (assign s time (h (s time)))
end.

```

Definition CMT :=

```
(Add act (Add act (Empty_set act) update_g) update_f) .
```

Definition CMT' :=

```
(Add act (Empty_set act) update_h) .
```

Again, we have verified that after this extension all theorems remain valid. More importantly, we have reproved these theorems in their informative versions, that we need to apply our compositionality rule in the proof of Theorem CMT_L16 below.

The only nontrivial proof obligation produced by Theorem CMT_L16 below is given by the following lemma: For the new component CMT' we just have to prove an `unless` assertion, which states noninterference with the informative version of CMT_L12 (a `inf_leads_to` assertion that is actually an `ensures` assertion).

```

Lemma CMT_L15 : (m:nat)(unless CMT'
  (s_and [s:st](le (s time) cmt)
    [s:st]m==(s time))
  (s_or (s_and [s:st](le (s time) cmt)
    [s:st](lt m (s time)))
    [s:st](s time)==cmt)).

```

```

Lemma CMT_L15 : (m:nat)(unless CMT'
  (s_and [s:st](le (s time) cmt)
    [s:st]m=(s time))
  (s_or (s_and [s:st](le (s time) cmt)
    [s:st](lt m (s time)))
    [s:st](s time)=cmt)).

```

Now we can prove the expected liveness property of the composed system as formulated in the following theorem.

```

Theorem CMT_L16 : (inf_leads_to (Union act CMT CMT'))
  ([s:st](le (s time) cmt)) ([s:st]((s time)==cmt))).

```

```

Theorem CMT_L16 : (inf_leads_to (Union act CMT CMT'))
  ([s:st](le (s time) cmt)) ([s:st](eq nat (s time) cmt))).

```

The proof is done by applying the compositionality theorem `inf_leads_to_union`. The verification of the compositionality property is reduced to CMT_L15 by applying `inf_leads_to_induction_comp` and `inf_leads_to_basis_comp`.

4.6 Related Work

There are a number of approaches to formalize UNITY in different logics and proof assistants, namely using the Boyer-Moore prover [Gol90, Gol92a, Gol99], the Larch prover [Che95, Che98], HOL [APP94, ABN⁺95], COQ [Hey97, HC96], and Isabelle/ZF [Pau00].

The references [Gol90, Gol92a, Gol99] use quantifier-free first-order logic to develop the UNITY temporal logic on the basis of its execution semantics. In [Che95, Che98] a first-order logic is employed to give an axiomatization of UNITY. The last three approaches employ more expressive logics: [APP94] and [HC96] directly use the higher-order logics of HOL and CIC, respectively, and [Pau00] uses Isabelle/ZF, a formalization of Zermelo-Fraenkel set theory in the higher-order framework of Isabelle.

The development presented in [Hey97, HC96] is the only one that uses an expressive logical type theory such as CIC. Different from the other approaches [Hey97, HC96] gives system-based *and* execution-based definitions of the main temporal operators and formalizes correctness and completeness proofs. In fact, the emphasis is more on semantic reasoning rather than on a metalogical viewpoint, which we emphasized in this chapter, as indicated by the fact that structural induction principles and all proof rules that are derived from such principles are not treated in [Hey97]. An interesting aspect of [Hey97, HC96], however, is the treatment of compositionality using a concept of environments called contexts. The thesis [Hey97] furthermore presents a number of nontrivial applications of this approach, including a quite complex ATM (asynchronous transfer mode) protocol. For a related comparison between the first-order approach of [Che95, Che98] and the higher-order approach [Hey97, HC96] we refer to [CH97].

Our approach is guided by New UNITY [Mis94, Mis95]. We have developed a verified and comprehensive library of temporal logic proof rules maintaining a good compromise between effort and formal safety. Instead of formalizing a particular execution semantics as in [Hey97, HC96], such as the one presented informally at the beginning of this chapter, we follow the semantics-independent approach of [CM88]. In this sense our approach bears some similarity with [APP94], which can also be classified as an essentially metalogical approach. On the other hand [APP94] sticks very close to the original UNITY approach and does not consider any generalizations beyond UNITY programs. Due to the rather weak type system of HOL, [APP94] cannot express and reason about abstract state spaces with typed variables, as it is possible in CIC with dependent types and universes. The semantics-independent approach, which has a slight proof-theoretic flavour in the sense that the main liveness operator `leads to` is given by an inductive definition, does not only fit well with the role of our development as a case study in metalogical reasoning using CIC, but it offers also the possibility of exploiting the propositions-as-types interpretation in the context of compositionality. As

far as we know, the propositions-as-types interpretation has neither been applied in the context of UNITY nor for this particular purpose until now.

It seems to be interesting to discuss the relation to [HC96], which also employs CIC, in more detail. Apart from the fact that our formalization is guided by the New UNITY approach (in fact also [Pau00] has chosen New UNITY as the basis for his formalization) another more essential difference between [HC96] and our approach is that we interpret assertions not over reachable states but over all states as in [APP94]. Instead of introducing an explicit concept of environment as in [HC96], we do not distinguish between the system actions and environment actions in the definitions of the temporal operators. All temporal operators are parameterized, i.e., they can explicitly refer to an arbitrary part of the full transition system. Quite different from other approaches, our initialization conditions appear only as parameters of some temporal operators. Furthermore, we explicitly support reasoning on restricted parts of the state by means of relativized assertions which are introduced in a conservative way.

Concerning compositionality, we have not committed ourselves to a notion which is built into the system model, but instead we think of compositional reasoning as a systematic way to structure temporal logic proofs. A basic form of compositional reasoning is supported directly in the UNITY-style temporal logic, and we have extended this approach to liveness assertions in a very general way. A different form of compositional reasoning is rely-guarantee style reasoning, which can be seen as an application of reasoning with conditional assertions as it has already been used in [CM88]. Arbitrary assertions (including liveness assertions) can serve as rely premisses. This approach allows us to cover also rely-guarantee style reasoning using the new operator “guarantees” which has been introduced in [CC99a] and formalized in [Pau00]. A more restrictive form of rely-guarantee style reasoning is supported in [HC96] by the notion of context, which essentially corresponds to a safety assumption about the environment. Although we think that it is convenient that an important case of rely-guarantee style reasoning (similar to the approaches of [Lam93] and [CK97]) is directly supported in [HC96], it is also clear to us that the restriction to safety assumptions is too limiting for many applications, and the original approach which uses conditional assertions is more flexible.

In contrast to other approaches we have used labeled transition systems as the underlying model of our development thereby subsuming functional transition systems, UNITY-programs and other system models as special cases. We have also used a more flexible notion of fairness which is adequate for this general setting: Instead of enforcing unconditional fairness uniformly for all actions, we can specify weak fairness, which takes into account enabledness of actions, for an arbitrary subset of actions. In [HC96] (and [CK97]) uniform unconditional fairness of system actions is assumed, but fairness of environment actions is not enforced. All the other approaches assume unconditional fairness uniformly for

all actions/statements. Beyond this more general notion of weak fairness which in addition can be specified for individual actions, our formalization allows us to work with weak group fairness. All these generalizations reflect our goal of obtaining a verified temporal logic library that is as general as possible, concerning the kind of system models that are admitted, while preserving the spirit and the elegance of the original UNITY approach.

4.7 Final Remarks

Logical type theories are very attractive candidates for the specification and verification of functional programs, since they provide an expressive and uniform formalism with an internal notion of logic and computation. Their elegance and simplicity is mostly due to the use of the propositions-as-types/proofs-as-objects interpretation, where the concept of program and proof on the one side, and the notion of type and proposition on the other are unified. The notions of imperative programs and more generally of concurrent and nondeterministic systems, however, are not directly supported by such type theories, although the need for rigorous formalisms to deal with these computational models is clearly growing.

In this chapter we have developed a simple approach combining a logical type theory and a notion of concurrent system by means of an embedding. The embedding generalizes the notion of UNITY programs to arbitrary labeled transition systems, which are used here as a basic model of concurrent systems. We furthermore use CIC as a metalogic to represent and to reason about a UNITY-style temporal logic, proving all the important proof rules from the literature and obtaining further results in our very general setting. Labeled transition systems can represent UNITY programs and similar models as interacting functional components communication via a common state space. More importantly, the approach we propose is not limited to concurrent programs of this kind, but applies to other notions of programs that can be represented as functional transition systems. Furthermore, many other formalisms for concurrent systems, including Petri nets and rewriting logic, can be equipped with a semantics in terms of labeled transition systems which are not functional. The embedding enables us to benefit from the capabilities of CIC in the context of specification and verification of concurrent programs and systems. We do not only take advantage of the (meta)logical capabilities of CIC, but our approach also benefits from its expressive data types, its built-in notion of computation, and its metatheoretic properties such as strong normalization.

In particular, our development exploits the internal higher-order logic of CIC as a framework logic for a temporal logic in the style of (New) UNITY. This provides a very expressive basis, which is not only important for abstract metalogical reasoning, but it also allows us to carry out specification and verification task for

concrete systems in a higher-order framework supporting general mathematical theorem proving. Of course, the last point is of major importance as already demonstrated by our simple example, which requires straightforward mathematical reasoning about natural numbers. In more realistic examples we can employ structural inductive reasoning about complex structures, since inductive types are a strong feature of CIC.

Another interesting aspect of this formal development is that we are working with two different logics: (1) the impredicative logic which resides in the standard propositional universe `Prop` and is axiomatized as a classical extensional logic with proof irrelevance; and (2) the predicative logic provided by the `Type` universe hierarchy which is intuitionistic, intensional and takes proofs seriously. The former has been used for the formalization of the UNITY-style temporal logic equivalent to the classical set-theoretic version presented in Section 4.2, and the latter is used for an intensional refinement of its `leads_to` fragment in Section 4.5, which exploits the proposition-as-type and proofs-as-objects paradigm to prove a compositionality theorem for an informative version of `leads_to`.

As explained in the introduction, our focus in the context of this thesis is on the use of type theory for metalogical reasoning in a semantics-independent setting, and not on reasoning within the temporal logic about concrete systems. In order to evaluate the practical use of the temporal logic library much more work is needed. We clearly need nontrivial cases studies, preferably using different specification formalisms. We gained some experience applying the UNITY-style temporal logic to place/transition nets and high-level nets in [Steb], where we verified a simple distributed network algorithm. A possible next step would be reasoning about more general system specifications given in rewriting logic, of which Petri nets are a special case as explained in Chapter 3. Guided by the new experience the library should be enhanced and extended. Among the many other interesting directions we would like to mention especially the following:

- For some applications, for instance in the context of Petri nets, a stronger notion of fairness can be useful to allow more expressive system specifications. Indeed, it seems to be possible to add a notion of strong fairness and of strong group fairness. The interesting question is whether this can be realized without destroying the simplicity and elegance of the UNITY approach, which is essentially based on its inductive and proof-theoretic nature.
- Another interesting problem of more theoretical significance is the issue of completeless w.r.t. an execution semantics, which obviously depends on the notion of fairness in a nontrivial way. Such a result would generalize existing completeness results for UNITY, although we found that the proof of [Pac90, Pac92] itself cannot be easily adapted.

- Last but not least, the addition of general support for compositional reasoning in a rely-guarantee style is another interesting subject (also witnessed by [Pau99]) and unavoidable for real-world applications. This could be done on the basis of conditional assertions along the lines of [CC99a] as in [Pau00], but also the integration of ideas from [Mis94], [Mis01], [CK97], [Pnu85], [Sha98] and [Lam94] is certainly worth considering.

4.8 Acknowledgements

The research contained in this chapter has been partly conducted in the scope of the European Community project “Modelling and Analysis of Time Constrained and Hierarchical Systems” (CHRX-CT94-0452), and the application of the results of this chapter to colored nets is subject of the contribution [Stea]. A detailed verification case study based on this methodology has been presented at the “Advanced Summer School on System Engineering” [Steb], and I greatly appreciate all the feedback from the participants. I am grateful for the help of my collaborators in the EC project, especially, Berndt Farwer, Rainer Mackenthun, Rüdiger Valk, the participants of the Summer School, and the referees of [Stea] for their comments that also led to improvements of the theoretical foundation presented here. I am also very grateful for the feedback from the participants in a seminar on “Formal Verification of Distributed Algorithms” given at the University of Hamburg in 1998, which was based on the material of an earlier version of this chapter. I furthermore would like to thank the members of the COQ team, who not only developed a very advanced and usable proof assistant but also patiently answered all my questions. Many thanks also to Barbara Heyd for pointing out to me her excellent work [Hey97, HC96] on this subject. I am glad to having been given the opportunity of presenting parts of this work at the SRI logic lunch and to having enjoyed discussions on different occasions with Robert Constable, Jean-Christophe Filliatre, Jean-Pierre Jouannaud, Christoph Kreitz, José Meseguer, J. Strother Moore, Cesar Muñoz, Pavel Naumov, Hassan Saidi, Natarajan Shankar, and Carolyn Talcott concerning program verification, type theory and related issues. Last but not least, I appreciate the encouraging remarks of Jayadev Misra with whom I recently discussed the compositionality result and the role of the (New) UNITY temporal logic in the context of his Seuss methodology, which provides a very economic and elegant framework to verify object-oriented concurrent systems. Jayadev Misra also sent me the latest version of the unpublished manuscript [Mis94] on Seuss and New UNITY, which has in the meantime been published in [Mis01], and I am very grateful for that.

Chapter 5

First-Order Representations of Higher-Order Formalisms:

A Calculus of Names and Substitutions

In the present chapter we approach the general problem of representing higher-order languages, which usually have specialized higher-order constructs, in a less specialized first-order framework such as membership equational logic or rewriting logic.

A common feature of higher-order languages is that essential entities which operate on data receive a first-class status so that variables can range over these entities. For instance, in higher-order logics or higher-order functional programming languages, variables can range over functions; in object-oriented programming languages, variables can range over objects (which can contain methods); and in languages for mobile processes, variables can range over channels (which are used to communicate with other processes). In order to express the essential entities directly as terms in the language, higher-order languages typically provide a syntax for abstractions, such as abstractions for functions, methods or processes in the examples above. Abstractions are essentially binding constructs that bind the free variables in the abstracted term with the intention to be instantiated later by means of substitution.

Currently, there are two major approaches to representing an object language with binding constructs in first-order frameworks such as membership equational logic [Mes98, BJM00] or rewriting logic [Mes92]. We distinguish between representations with names and representations based on de Bruijn indices. Representations with names have the very desirable feature of being close to the object language; their major drawback is that they lack a canonical way to treat names and potential name clashes. Calculi based on de Bruijn indices have the advantage of a canonical representation, but they are more abstract, since information about names is not represented, so that the gap between the object language and its representation is considerable.

In addition to the representation of the object language, another important issue is capture-free substitution, the main operation that is needed for terms of the object language. For a definition of substitution on terms with binding constructs, it turns out to be useful to treat substitutions as first-class citizens, since substitutions usually have to be adjusted as they are propagated through the term they are applied to. Since substitutions receive the same formal status as terms, the calculi that deal with substitution in this way are called *explicit substitution calculi*.

Research in this area has led to a rich collection of calculi (overviews and comparisons can be found in [Les94, Ros96, Blo97, Muñ97, KR00]) with quite different properties and motivations. Most of this research is focused on the λ -calculus (notable exceptions are [BR96] and [Pag98]) and the use of explicit substitutions to express β -reduction in terms of a more primitive concept. Among the motivations for using explicit substitutions we can find the following: the need for a rigorous and simple explanation of capture-free substitution [BBLRD96], the quest

for a notion of computation that is more fine-grained and more implementation-oriented than standard β -reduction [Cur91, ACCL91], the interest in analysis of evaluation strategies and efficiency of computation [Fie90], the application of first-order techniques to higher-order languages [DHK99], and the use of algorithms that operate on incomplete terms such as higher-order unification [DHK95], type checking/inference [SM99], and proof synthesis [Muñ98, Muñ97].

Our primary motivation for proposing a new calculus of explicit substitutions in this paper is to obtain a first-order representation of terms with binding and capture-free substitution that is as close as possible to the standard named notation. Beyond that, we want this calculus to be executable, in the sense that it can be executed using a rewriting engine such as Maude [CDE⁺99a, CDE⁺00b], and furthermore we are interested in a general solution, in the sense that the theory does not restrict itself to a particular object language. More precisely, the objective of this chapter is to develop a calculus of names and explicit substitutions that takes names seriously, and completely removes the gap between the object language and its representation (often called representational distance) without losing the possibility of canonical representations. A solution that is closely related to de Bruijn's representation [dB72] but has been developed independently by Berkling [Ber76, BF82] is a unification of named and indexed notation. Despite its advantages, which have also been recognized more recently in [Rei98a], it is an unconventional representation that has not attracted much attention so far. Therefore, we devote the next section to an introduction and motivation of Berkling's representation, which will serve as a basis for the CINNI substitution calculus that we propose. CINNI is not only the first explicit substitution calculus based on Berkling's representation that we are aware of, but the CINNI approach also extends his representation for the λ -calculus to a wide range of object languages with binding constructs.

5.1 Indexed Names and Named Indices

Consider the standard treatment of binding constructs, say in the context of first-order logic, where α -equivalent terms, i.e. terms that can be transformed into each other by consistent renaming of bound variables, are identified, i.e. not distinguished for essential parts of the metatheory. An obvious first step towards a named representation is to give up this identification that we also refer to as α -equality. Unfortunately, this rather naive approach leads to the following difficulty that we refer to as *accidental hiding*.

Consider for instance the formula

$$\forall X . (A \wedge \forall Y . (B \Rightarrow \forall X . C(X)))$$

for distinct names X and Y . Assume the subformula $C(X)$ contains X free.

Then each free occurrence of X in $C(X)$ is *captured* by the inner \forall quantifier, so that the name bound by the outermost \forall quantifier is hidden from the viewpoint of $C(X)$. Indeed there is no way to refer to the outermost \forall quantifier within $C(X)$.

Hence, we are faced with the following problem: a calculus without α -equality is not only less abstract, which is an unavoidable consequence of giving up identification by α -conversion, but also, depending on the (accidental) choice of names, visibility of (bound) variables may be restricted. It is important to emphasize that visibility is not restricted in the original calculus with α -equality, since renaming can be performed *tacitly* at any time.

Clearly, the phenomenon of hiding that occurs in the example above is undesirable¹, because it is not present in the original calculus with α -equality. It is merely an accident caused by giving up identification by α -conversion without adding a compensating flexibility to the language.²

This suggests tackling this general problem by migrating to a more flexible syntax, where we express a binding constraint by annotating each name X with an index $i \in \mathbb{N}$, written X_i , that indicates how many X -binders should be skipped before we reach the one that X_i refers to. For instance, we write

$$\forall X . (A \wedge \forall Y . (B \Rightarrow \forall X . C(X_0)))$$

to express that X_0 is bound by the inner \forall , and

$$\forall X . (A \wedge \forall Y . (B \Rightarrow \forall X . C(X_1)))$$

meaning that X_1 is bound by the outermost \forall . To make the language a conservative extension of the traditional notation, we can identify X and X_0 . An equivalent representation has been introduced by Berkling [Ber76, BF82] in the context of the λ -calculus³. As indicated by the example above we will use Berkling's representation not (only) for the λ -calculus but as the core syntax of CINNI, the *Calculus of Indexed Names and Named Indices*, which is generic in the sense that it can be instantiated for a wide range of object languages.

Obviously, there is some similarity to a notation based on de Bruijn indices [dB72], but notice that there is an essential difference: the index m in the occurrence X_m is *not* the number of binders to be skipped; it states that we have to skip m binders *for the particular name X* , *not* counting binders for other names. Still a formal

¹Of course, in general hiding is important but it is not an issue of binding; it should be treated independently.

²A good example of how hiding causes difficulties in the context of type theories with dependent types (known as the the problem of closure under α -conversion) will be discussed in Chapter 6.

³An indexed variable X_i is represented in Berkling's representation as $\#^i X$ where $\#$ is the so-called unbinding operation.

relationship to de Bruijn's notation can be established: if we restrict ourselves to terms that contain only a single name X , then we can replace each X_i by the index i without loss of information, and we arrive at de Bruijn's purely indexed notation.⁴ In other words, if we restrict the available names to a single one, we obtain de Bruijn's notation as a very special case. In this sense, Berkling's representation can be formally seen as a proper generalization of de Bruijn's notation. Pragmatically, however, the relationship to de Bruijn's syntax plays only a minor role, since a typical user will exploit the dimension of names much more than the dimension of indices. Hence, in practice the notation can be used as a standard named notation, with the additional advantage that accidental hiding and weird renamings⁵ are avoided.

The pragmatic advantage of Berkling's notation is that it can be used to reduce the distance between the formal system and its implementation: it can be directly employed by the user who wants to think in terms of names, so that the need for a translation between an internal representation (e.g. using de Bruijn indices) and a user friendly syntax (e.g. using ordinary names) disappears completely.

Usually, this translation between an internal and an external representation is not considered to be a problem, and indeed, in the case of terms where all parts are known or accessible, solutions are straightforward. However, even in this case this gap is not desirable. Consider, for example, a tactic-based theorem prover where the user is confronted with an internal representation which reflects the theory only in a very indirect way. More seriously, the translation between internal and external representations becomes impossible, or at least requires certain restrictions, as soon as we use terms containing metavariables, holes or placeholders, which are useful for many applications, including unification algorithms and representation of incomplete proofs.

We conclude this section with an example which shows that standard notation as well as Berkling's notation is strictly more expressive than a de Bruijn index-based notation. Consider the following λ -calculus equation (we write $[X] M$ instead of $\lambda X.M$) with distinct names X, Y, Z and a single unknown metavariable "?":

$$[X] (([Y][Z] ?) 0) = [X][Z] ?$$

An obvious solution for this equation is $? = X$, but a corresponding solution cannot be expressed in the nameless version:

$$[\bullet] ((([\bullet][\bullet] ?) 0) = [\bullet][\bullet] ?$$

⁴With the slight difference that de Bruijn's indices start at 1 instead of at 0.

⁵See the discussion on weird renaming in the next section.

5.2 Explicit Substitutions

In the previous section we discussed Berklings’s first-order representation for expressions which contains the conventional named notation as well as de Bruijn’s indexed notation as special cases. The most important operation to be performed on such terms represented in this way is capture-free substitution. Therefore, we now present the CINNI substitution calculus, a first-order calculus that can be seen as an (operational) refinement of an external (i.e. metalevel) substitution function such as the one given in [BF82].

Strictly speaking, CINNI is a family of explicit substitution calculi, parameterized by the syntax of the language we want to represent. For a language \mathcal{L} given by its syntax we denote the corresponding instantiation of CINNI by $\text{CINNI}_{\mathcal{L}}$. The syntax defines the term constructors together with their binding constraints⁶ which are expressed by associating to each term constructor f a binary relation on $\{1, \dots, n\}$ as follows: If (i, j) is in the relation, then the i -th argument must be a name, the j -th argument must be a term, and we say that f binds argument i in argument j , that is, for each term $f(P_1, \dots, P_n)$ of the object language, the name P_i is bound in the subterm P_j . Note that each P_i is either a term or a name.

As an example we use the untyped λ -calculus to present the concrete instantiation CINNI_{λ} of the substitution calculus. CINNI_{λ} -terms are generated by the syntax

$$X_m \mid (M N) \mid [X] M$$

with the constraint that $[-]_-$ binds argument 1 in argument 2.

As a motivation for the substitution calculus given below, consider the following example of a β -reduction step in the traditional λ -calculus with distinct names X and Y , again taking names literally, i.e. not presupposing identification by α -conversion:

$$(([X][Y]X)Y) \rightarrow_{\beta} [Z]Y$$

Clearly, the bound variable Y must be renamed to Z , a name different from Y , to avoid capturing of the free variable Y . Unfortunately, there is no canonical choice if all names should be treated as being equal. We call this phenomenon *weird renaming* of bound variables. It is actually a combination of two undesirable effects: (1) names that have been carefully chosen by the user have to be changed, and (2) the enforced choice of a new name collides with the right of names to be treated as equal citizens. These effects are avoided in the CINNI calculus, which

⁶A similar way to formalize the syntax of a language with binding is used in binding structures [Tal93]. However, different from our approach bound variables do not carry names in binding structures but are represented using de Bruijn indices. See Section 5.5 for a discussion of further differences and similarities.

is specified by the first-order equational theory given below. Indeed, the only operation assumed on names is equality.

More formally, we assume that the syntax of the object language is given by a signature in membership equational logic which includes a sort of names (assumed to be nonempty), a sort of natural numbers (with zero 0 and successor +1 as the only operations), and a sort of object language terms.⁷ In addition, we have a constructor for *variables*, i.e. terms of the form X_m , as well as additional constructors for terms together with their binding constraints and optional structural equations (see below).

To present the actual calculus we need to extend the notion of term by explicit substitutions. To this end, we introduce a sort of substitutions together with the following operators: In addition to the two basic kinds of substitutions, namely *simple* substitutions $[X:=M]$ and *shift* substitutions \uparrow_X , a substitution S can be *lifted* using $\uparrow_X S$. The application $S M$ of an explicit substitution to a term is again a term. Now $\text{CINNI}_{\mathcal{L}}$ has the signature just described, and the following equations:

$$\begin{array}{ll}
[X:=M] X_0 = M & (\text{FVar}) \\
[X:=M] X_{m+1} = X_m & (\text{RVarEq}) \\
[X:=M] Y_n = Y_n \text{ if } X \neq Y & (\text{RVarNEq}) \\
\uparrow_X X_m = X_{m+1} & (\text{VarShiftEq}) \\
\uparrow_X Y_n = Y_n \text{ if } X \neq Y & (\text{VarShiftNEq}) \\
\uparrow_X S X_0 = X_0 & (\text{FVarLift}) \\
\uparrow_X S X_{m+1} = \uparrow_X (S X_m) & (\text{RVarLiftEq}) \\
\uparrow_X S Y_n = \uparrow_X (S Y_n) \text{ if } X \neq Y & (\text{RVarLiftNEq})
\end{array}$$

Furthermore, for each syntactic constructor f of \mathcal{L} we add a *syntax-specific equation*

$$S f(P_1, \dots, P_n) = f(\uparrow_{P_{j_{1,1}}} \dots \uparrow_{P_{j_{1,m_1}}} S P_1, \dots, \uparrow_{P_{j_{n,1}}} \dots \uparrow_{P_{j_{n,m_n}}} S P_n)$$

where $j_{i,1}, \dots, j_{i,m_i}$ are all the arguments (necessarily of the name sort) that f binds in argument i (necessarily of the term sort). If P_k is of name sort we identify $S P_k$ and P_k for every substitution S . This abuse of notation allows us to write the syntax-specific equations in the compact form given above.

Often the terms of the object language are not just freely generated by the syntactic constructors, but are subjected to additional *structural equations*. Admissible

⁷The restriction to a single sort of terms and hence to a single corresponding kind is not essential and only used to simplify the presentation. In fact, terms could be specified using multiple kinds by having a substitution application operator for each of them. This obvious generalization is demonstrated by the application in Section 5.4.2.

equations in this chapter are the laws of associativity, commutativity and identity for binary operators, and we assume throughout the chapter that structurally equivalent terms are identified. If f is a binary operator with identity e we add a condition to the syntax-specific equation above which ensures that none of the P_i is equal to e (cf. the specifications of the ζ -calculus and the π -calculus in Section 5.4). This ensures that the left hand sides of valid instances of syntax-specific equations do not overlap, and is also needed to ensure termination of the corresponding rewrite system.

The syntax-specific equations are the only equations that depend on the syntax of \mathcal{L} . For instance, CINNI_λ has the following syntax-specific equations:

$$\begin{aligned} S(MN) &= (SM)(SN) && \text{(App)} \\ S([X]M) &= [X](\uparrow_X S M) && \text{(Lambda)} \end{aligned}$$

The equations of CINNI can be justified by the following algebraic substitution semantics: a substitution S is interpreted as a function from variables to terms. Application of substitution is interpreted as function application. The substitutions $[X:=M]$, \uparrow_X and $\uparrow_X S$ are then uniquely defined by the equations above. Finally, substitutions are extended from variables to terms by the syntax-specific equations. Each time a substitution moves into a new scope it has to be adjusted using a lift substitution.

CINNI is not only an equational calculus with an algebraic semantics, but as usual for explicit substitution calculi it can be equipped with an operational semantics by regarding equations as rewrite rules. We refer to the resulting term rewrite system as the *CINNI rewrite system*, and following the computational system of membership equational logic we introduce the following relations: We define the binary relation \Rightarrow_S on terms such that $M \Rightarrow_S N$ holds iff CINNI has an equational axiom $\forall X. L = R$ if $\bar{\phi}$ and there is a kinded substitution θ for X such that $\theta(\bar{\phi})$ is derivable. The *reduction relation* induced by \Rightarrow_S , i.e. its context closure, is denoted by \rightarrow_S . The induced equivalence on terms is denoted by \equiv_S . Going back to CINNI_λ , the effect of lift substitutions can be best understood by considering special cases of the syntax-specific equation (Lambda). We have

$$\begin{aligned} [X:=M]([X]X_0) &\equiv_S [X](\uparrow_X [X:=M]X_0) \equiv_S ([X]X_0) && \text{(Lambda')} \\ [X:=M]([X]X_1) &\equiv_S [X](\uparrow_X [X:=M]X_1) \equiv_S [X]\uparrow_X M && \text{(Lambda'')} \end{aligned}$$

Here X_0 refers to X in the lambda abstraction and X_1 refers to X in the substitution.

We can now define the explicit substitution version of the β -rule by

$$([X]N)M \Rightarrow_B [X:=M]N.$$

Notice that weird renaming of bound variables as in the previous example is avoided with the new notion of β -reduction. For instance, we have

$$(([X][Y]X_0)Y_0) \rightarrow_{\text{SB}}^* ([Y]Y_1)$$

where \rightarrow_{SB} denotes the context closure of $\Rightarrow_{\text{S}} \cup \Rightarrow_{\text{B}}$. Notice also that we do not view application-specific computation rules as a part of the substitution calculus (which is CINNI_λ in this case). The substitution calculus only depends on the syntax of the object language.

As another application of substitution, consider *renaming of a bound variable* X by \bullet as in the following explicit substitution version of α -reduction

$$([X]N) \Rightarrow_{\text{A}} ([\bullet][X:=\bullet] \uparrow_{\bullet} N) \text{ if } X \neq \bullet$$

where \bullet is an arbitrary but fixed name. Using this rule and the rules for explicit substitutions, every CINNI_λ term can be reduced to a nameless α -normal form which is essentially its de Bruijn index representation. α -reduction is a new concept that becomes expressible due to the use of a unified syntax with indices and names.

Just as Berkling's notation contains de Bruijn's notation as a very special case, the instantiation of CINNI for the λ -calculus reduces to the calculus λv of explicit substitutions proposed in [Les94, LRD94, BBLRD96], but only in the degenerate case where we restrict the set of names to a singleton set. It is noteworthy that λv is the smallest known indexed substitution calculus enjoying good theoretical properties like confluence and preservation of strong normalization. It seems that its simplicity is inherited by CINNI although in practice the dimension of names will be much more important than the dimension of indices. Hence, we tend to think of CINNI more as a substitution calculus with names than as one with indices.

5.3 Metatheoretic Properties of CINNI

In this section we give a number of important metatheoretic properties concerning both the CINNI calculus in isolation, and the composition of CINNI with application-specific rules such as the explicit substitution version of the β -rule in the case of CINNI_λ . The present section generalizes the results of [Les94] for λv in two orthogonal dimensions.

The first dimension is the scope of applicability:

1. Instead of considering a fixed object language such as the λ -calculus, we consider an arbitrary object language given by a syntax \mathcal{L} with binders.

2. Instead of considering a fixed set of computation rules, such as β -reduction in the λ -calculus, we allow arbitrary computation rules R as long as they satisfy a certain well-formedness property.

The second dimension of generalization is concerned with the representation of the object language:

1. The de Bruijn index representation is generalized to a richer representation with indexed names.
2. The substitution calculus and its properties are generalized accordingly.

We first consider the CINNI rewrite system in isolation, instantiated for the syntax \mathcal{L} of an arbitrary object language. We denote this instance by $\text{CINNI}_{\mathcal{L}}$ and use Trm to denote all its terms. For all results we assume that \mathcal{L} has at least one name. In practice we think that \mathcal{L} should have a countably infinite set of names, but it is interesting to see that our proofs do not rely on this assumption. In fact all our results cover the case where \mathcal{L} has exactly one name and the term representation reduces to the well-known representation based on de Bruijn indices.

Definition 5.3.1 We define the following two functions on terms to express iterated shift and lift substitutions:

$$\begin{aligned} \uparrow_{\emptyset}(M) &= M & \uparrow_{\emptyset}S(M) &= S M \\ \uparrow_{\bar{Y},X}(M) &= \uparrow_{\bar{Y}}(\uparrow_X M) & \uparrow_{\bar{Y},X}S(M) &= \uparrow_{\bar{Y}}\uparrow_X S(M) \end{aligned}$$

Here and in the following we use variables $\bar{X}, \bar{Y}, \bar{Z}, \bar{U}, \bar{V}, \bar{W}$ to range over lists of names, \emptyset denotes the empty list and comma denotes concatenation. Furthermore, we use $|\bar{Y}|_X$ to denote the number of occurrences of X in \bar{Y} .

5.3.1 Strong Normalization

Confluence of CINNI can be easily established by transforming the rewrite system into an equivalent *orthogonal*, i.e. left-linear and nonoverlapping, rewrite system without conditions. This is done by replacing each possibly conditional equation by all its valid instances obtained by instantiating X and Y by concrete names. Notice that, if the set of names is infinite, the resulting system becomes infinite, too. This orthogonal version of CINNI will also be used in the proof of strong normalization (Theorem 5.3.3).

Theorem 5.3.2 (Confluence)

The relation \rightarrow_S is confluent.

Some mathematical evidence that CINNI_λ is a nontrivial generalization of $\lambda\nu$ and its metatheory seems to be given by the observation that the proof of strong normalization in [BBLRD96, LRD94], which makes use of elementary interpretations [Les92], cannot be applied to CINNI_λ . Indeed the problematic equations are (RVarLiftNEq) and the syntax-specific equations which make it unlikely that a proof of strong normalization can be obtained by a modified elementary interpretation. Furthermore, the syntax-specific equations seem to prevent us from giving a proof based on recursive path orderings. Hence, we pursue a different approach which makes use of the fact that in orthogonal term rewrite systems all maximal computations that do not erase redexes are essentially equivalent. In particular, computations that only perform innermost reductions are in a certain sense representative computations and can be used to prove strong normalization as already observed in [O'D77]. Indeed, the following theorem is proved by exhibiting an innermost-reduction strategy and showing that it always terminates.

Theorem 5.3.3 (Strong Normalization)

The relation \rightarrow_S is strongly normalizing.

Proof We first consider the rewrite system CINNI , i.e. $\text{CINNI}_\mathcal{L}$ without the syntax-specific equations for \mathcal{L} . It is strongly normalizing as it can be easily verified using the following polynomial interpretation:

$$\begin{aligned} \llbracket X_m \rrbracket &= m + 2 \\ \llbracket f(P_1, \dots, P_n) \rrbracket &= \llbracket P_1 \rrbracket + \dots + \llbracket P_n \rrbracket + 2 \quad \text{with } \llbracket X \rrbracket = 1 \\ \llbracket \uparrow x \rrbracket &= 3 \\ \llbracket [X := M] \rrbracket &= \llbracket M \rrbracket \\ \llbracket \uparrow_x S \rrbracket &= 4 \llbracket S \rrbracket \\ \llbracket [S M] \rrbracket &= \llbracket S \rrbracket \llbracket M \rrbracket \end{aligned}$$

To prove strong normalization for $\text{CINNI}_\mathcal{L}$ we use the theorem from [O'D77] according to which an orthogonal term rewriting system is strongly normalizing iff each term has a normal form reachable by innermost reduction. Hence, strong normalization of $\text{CINNI}_\mathcal{L}$ can be verified by exhibiting an innermost reduction strategy that always terminates. Given a term Q'_0 , the strategy repeatedly applies the following two steps starting with Step (2) until the termination condition (see Step (2)) is satisfied.

1. If Q_i contains $\text{CINNI}_\mathcal{L}$ -redexes (after executing Step (2)) they can only be redexes w.r.t. a syntax-specific equation. Pick an occurrence of such an innermost redex N_i in Q_i which is necessarily of the form $S f(P_1, \dots, P_n)$. We can write Q_i as $C_i[N_i]$ where C_i is a context. Now reduce N_i using only syntax-specific equations as far as possible and call the result N'_i . This gives

a term $C_i[N'_i]$ that we denote by Q'_i . Observe that, since each constructor f in \mathcal{L} has an associated syntax-specific equation, each maximal (w.r.t. subterm ordering) the substitution in Q'_i is of the form $\uparrow_{\bar{Z}} S$ and must be applied to a variable, i.e. it is part of an application of the form $\uparrow_{\bar{Z}} S X_m$.

2. Reduce Q'_i to normal form w.r.t. CINNI (i.e. $\text{CINNI}_{\mathcal{L}}$ without syntax-specific equations) by successively reducing innermost redexes and call the result Q_{i+1} . Notice that Q_{i+1} cannot contain substitution applications of the form $S X_m$. If Q_{i+1} does not contain any substitutions the strategy terminates.

We say that the sequential execution of Step (1) and Step (2) constitutes a round of the strategy. To define a well-founded order that decreases in each round we first define $\llbracket Q \rrbracket$ as the multiset of all substitutions contained in Q which can be either a substitution or a term. More precisely

$$\begin{aligned} \llbracket X \rrbracket &= \emptyset \\ \llbracket X_m \rrbracket &= \emptyset \\ \llbracket f(P_1, \dots, P_n) \rrbracket &= \llbracket P_1 \rrbracket \oplus \dots \oplus \llbracket P_n \rrbracket \\ \llbracket \uparrow_X \rrbracket &= \{\uparrow_X\} \\ \llbracket [X:=M] \rrbracket &= \{[X:=M]\} \oplus \llbracket M \rrbracket \\ \llbracket \uparrow_X S \rrbracket &= \{\uparrow_X S\} \oplus \llbracket S \rrbracket \\ \llbracket S M \rrbracket &= \{S\} \oplus \llbracket S \rrbracket \oplus \llbracket M \rrbracket \end{aligned}$$

where \oplus denotes multiset union. Then we define the complexity $C(Q)$ as the multiset of natural numbers $\bigoplus_{S \in \llbracket Q \rrbracket} \llbracket S \rrbracket$, where $\llbracket S \rrbracket$ is the cardinality of $\llbracket S \rrbracket$ as a multiset. We furthermore define the partial order $(\mathcal{MS}(\mathbb{N}), <_m)$ as the multiset extension [DM79] of $(\mathbb{N}, <)$. For subsequent rounds i and $i+1$ of the above strategy we verify $C(Q_i) >_m C(Q_{i+1})$ by distinguishing the following two cases after performing Step (1).

- S is of the form $\uparrow_{\bar{Y}} \uparrow_U$. In this case each maximal substitution in N'_i (as a subterm of Q'_i) is of the form $\uparrow_{\bar{Z}} \uparrow_{\bar{Y}} \uparrow_U X_m$. The subsequent step (2) will replace each of these subterms in N'_i by a term of the form $X_{m'}$, which is clearly of lower complexity than $\uparrow_{\bar{Z}} \uparrow_{\bar{Y}} \uparrow_U X_m$.
- S is of the form $\uparrow_{\bar{Y}} [U:=M]$. In this case each maximal substitution in N'_i (as a subterm of Q'_i) is of the form $\uparrow_{\bar{Z}} [U:=M] X_m$. The subsequent step (2) will replace each of these substitution applications by either a variable $X_{m'}$ (if $X \neq U$ or M is a variable) or a term of the form $\uparrow_{\bar{Z}'} M$ where \bar{Z}' is a sublist of \bar{Z} (if $X = U$ and M is not a variable), which are both terms of lower complexity than $\uparrow_{\bar{Z}} [U:=M] X_m$.

Well-foundedness of $(\mathcal{MS}(\mathbb{N}), <_m)$ is inherited from $(\mathbb{N}, <)$. Hence, we conclude that the above strategy terminates. \square

As a consequence of this theorem, each $\text{CINNI}_{\mathcal{L}}$ -term M and each $\text{CINNI}_{\mathcal{L}}$ -substitution S have a unique *substitution normal form*, which will be denoted by $NF_S(M)$ and $NF_S(S)$, respectively. Notice that $NF_S(M)$ is a *pure term*, i.e., it does not contain any substitutions; otherwise one of the rewrite rules could be applied.

5.3.2 Equational Properties

The following induction lemma provides a tool for proving certain equivalences of the form $S_l^L \dots S_1^L M \equiv_S S_r^R \dots S_1^R M$, and it has been used in the proofs of the subsequent lemmas which state basic equational properties of the CINNI-calculus.

Lemma 5.3.4 (Induction Lemma)

Let $S_1^L \dots S_l^L$ and $S_1^R \dots S_r^R$ be substitutions, and let Trm' be a subset of Trm which is subterm-closed. Define

$$\begin{aligned} L(M) &= S_l^L \dots S_1^L M, \\ R(M) &= S_r^R \dots S_1^R M, \\ \uparrow_{\bar{W}} L(M) &= \uparrow_{\bar{W}} S_l^L \dots \uparrow_{\bar{W}} S_1^L M, \\ \uparrow_{\bar{W}} R(M) &= \uparrow_{\bar{W}} S_r^R \dots \uparrow_{\bar{W}} S_1^R M. \end{aligned}$$

In order to prove the equivalence

$$\uparrow_{\bar{W}} L(M) \equiv_S \uparrow_{\bar{W}} R(M)$$

and in particular

$$L(M) \equiv_S R(M)$$

for all \bar{W} and terms $M \in \text{Trm}'$, it is sufficient to show that

$$\uparrow_{\bar{W}} L(X_k) \equiv_S \uparrow_{\bar{W}} R(X_k)$$

for all \bar{W} and terms $X_k \in \text{Trm}'$.

Proof Since each term N is equivalent to a pure term $NF_S(N)$ by Theorem 5.3.3 we can proceed by structural induction over pure terms M in Trm' . Our induction hypothesis is: $\forall \bar{W} . \uparrow_{\bar{W}} L(M) \equiv_S \uparrow_{\bar{W}} R(M)$.

Base Case: Given by the assumption of the lemma.

Induction Step: Assuming the induction hypothesis

$$\forall \bar{W} . \uparrow_{\bar{W}} L(P_i) \equiv_S \uparrow_{\bar{W}} R(P_i)$$

with $P_i \in Trm'$ for each $i \in \{1, \dots, n\}$ we have to prove

$$\uparrow_{\bar{W}} L(f(P_1, \dots, P_n)) \equiv_S \uparrow_{\bar{W}} R(f(P_1, \dots, P_n))$$

for $f(P_1, \dots, P_n) \in Trm'$.

In view of the syntax-specific equation for f it is sufficient to show that

$$\begin{aligned} & f((\uparrow_{P_{j_1,1}, \dots, P_{j_1, m_1}, \bar{W}} S_l^L \dots \uparrow_{P_{j_1,1}, \dots, P_{j_1, m_1}, \bar{W}} S_1^L P_1), \dots, \\ & (\uparrow_{P_{j_n,1}, \dots, P_{j_n, m_n}, \bar{W}} S_l^L \dots \uparrow_{P_{j_n,1}, \dots, P_{j_n, m_n}, \bar{W}} S_1^L P_n)) \equiv_S \\ & f((\uparrow_{P_{j_1,1}, \dots, P_{j_1, m_1}, \bar{W}} S_r^R \dots \uparrow_{P_{j_1,1}, \dots, P_{j_1, m_1}, \bar{W}} S_1^R P_1), \dots, \\ & (\uparrow_{P_{j_n,1}, \dots, P_{j_n, m_n}, \bar{W}} S_r^R \dots \uparrow_{P_{j_n,1}, \dots, P_{j_n, m_n}, \bar{W}} S_1^R P_n)) \end{aligned}$$

where the $j_{u,v}$ are defined as before (Section 5.2). The above equivalence follows immediately by applying the induction hypothesis n times. \square

In each of the following lemmas we explicitly state a strong equivalence, that can be established using the induction lemma, followed by some weaker consequences that are typically sufficient for most purposes. If not explicitly stated the induction lemma is instantiated with $Trm' = Trm$, i.e. to prove properties of all terms.

Lemma 5.3.5 (Shift Shift Reordering I)

1. $\uparrow_Z \uparrow_Y X_m \equiv_S \uparrow_Y \uparrow_Z X_m$,
2. $\uparrow_{\bar{Z}} \uparrow_{\bar{Y}} X_m \equiv_S \uparrow_{\bar{Y}} \uparrow_{\bar{Z}} X_m$.

Proof (1) is a special case of (2) which is obtained as follows: $\uparrow_{\bar{Z}} \uparrow_{\bar{Y}} X_m \equiv_S \uparrow_{\bar{Z}} X_{m+y} \equiv_S X_{m+y+z} \equiv_S \uparrow_{\bar{Y}} X_{m+z} \equiv_S \uparrow_{\bar{Y}} \uparrow_{\bar{Z}} X_m$ where $y = |\bar{Y}|_X$ and $z = |\bar{Z}|_X$. \square

Lemma 5.3.6 (Simplification I)

1. $\uparrow_{\bar{Y}} S X_m \equiv_S X_m$ if $m < |\bar{Y}|_X$,
2. $\uparrow_{\bar{Y}} S X_m \equiv_S \uparrow_{\bar{Y}} S X_{m-i}$ if $m \geq |\bar{Y}|_X = i$.

Proof

1. We proceed by induction over m .

Base case: Assume $|\bar{Y}|_X > m = 0$. Choose \bar{Z}, \bar{Z}', X with $\bar{Z}', X, \bar{Z} = \bar{Y}$ and $X \notin \bar{Z}'$. We have $\uparrow_{\bar{Y}} S X_0 \equiv_S \uparrow_{\bar{Z}', X, \bar{Z}} S X_0 \equiv_S \uparrow_{\bar{Z}} \uparrow_X \uparrow_{\bar{Z}} S X_0 \equiv_S \uparrow_{\bar{Z}'} \uparrow_X \uparrow_{\bar{Z}} S X_0 \equiv_S \uparrow_{\bar{Z}'} X_0 \equiv_S X_0$.

Induction step: Our induction hypothesis is: $\forall \bar{X}, \bar{Y} . \uparrow_{\bar{Y}} S X_m \equiv_S X_m$ if $|\bar{Y}|_X > m$. Now consider $\uparrow_{\bar{Y}} S X_{m+1}$ assuming $|\bar{Y}|_X > m + 1$. Choose \bar{Z}, \bar{Z}' , and X with $\bar{Z}', X, \bar{Z} = \bar{Y}$ and $X \notin \bar{Z}'$. We then have $\uparrow_{\bar{Y}} S X_{m+1} \equiv_S \uparrow_{\bar{Z}', X, \bar{Z}} S X_{m+1} \equiv_S \uparrow_{\bar{Z}'} \uparrow_{X, \bar{Z}} S X_{m+1} \equiv_S \uparrow_{\bar{Z}'} \uparrow_X \uparrow_{\bar{Z}} S X_m$. By the induction hypothesis we have $\uparrow_{\bar{Z}'} \uparrow_X \uparrow_{\bar{Z}} S X_m \equiv_S \uparrow_{\bar{Z}'} \uparrow_X X_m$ since $|\bar{Z}|_X > m$. Finally, $\uparrow_{\bar{Z}'} \uparrow_X X_m \equiv_S X_{m+1}$.

2. We proceed by induction over i .

Base Case: Clearly, $\uparrow_{\bar{Y}} S X_m \equiv_S \uparrow_{\bar{Y}} S X_m$ for $m \geq |\bar{Y}|_X = 0$.

Induction Step: Our induction hypothesis is: $\forall X, \bar{Y}, m . \uparrow_{\bar{Y}} S X_m \equiv_S \uparrow_{\bar{Y}} S X_{m-i}$ if $m \geq |\bar{Y}|_X = i$. Now consider $\uparrow_{\bar{Y}} S X_n$ assuming $n \geq |\bar{Y}|_X = i + 1$. Clearly, $n = m + 1$ for some m . Choose \bar{Z}, \bar{Z}' , and X with $\bar{Z}', X, \bar{Z} = \bar{Y}$ and $X \notin \bar{Z}'$. Then we have $\uparrow_{\bar{Y}} S X_n \equiv_S \uparrow_{\bar{Z}', X, \bar{Z}} S X_{m+1} \equiv_S \uparrow_{\bar{Z}'} \uparrow_X \uparrow_{\bar{Z}} S X_{m+1} \equiv_S \uparrow_{\bar{Z}'} \uparrow_X \uparrow_{\bar{Z}} S X_m$. By the induction hypothesis we have $\uparrow_{\bar{Z}'} \uparrow_X \uparrow_{\bar{Z}} S X_m \equiv_S \uparrow_{\bar{Z}'} \uparrow_X \uparrow_{\bar{Z}} S X_{m-i}$ since $m \geq |\bar{Z}|_X = i$. Finally, $\uparrow_{\bar{Z}'} \uparrow_X \uparrow_{\bar{Z}} S X_{m-i} \equiv_S \uparrow_{\bar{Y}} S X_{m-i}$. \square

Lemma 5.3.7 (Simplification II)

1. $\uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S X_m$ where $Z \neq X$,
2. $\uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_Z X_m$ if $m \geq |\bar{Y}|_X$,
3. $\uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_Z X_m$ if $Z \notin \bar{Y}$,
4. $\uparrow_{\bar{W}} \uparrow_{\bar{Y}} \uparrow_Z M \equiv_S \uparrow_{\bar{W}} \uparrow_Z M$ if $Z \notin \bar{Y}$,
5. $\uparrow_{\bar{Y}} \uparrow_Z M \equiv_S \uparrow_Z M$ if $Z \notin \bar{Y}$.

Proof

1. If $m < |\bar{Y}|_X$ this follows from Lemma 5.3.6.(1). If $m \geq |\bar{Y}|_X$ it follows from Lemma 5.3.6.(2) and $Z \neq X$.
2. Lemma 5.3.6.(2) gives $\uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_{\bar{Y}} \uparrow_Z X_{m-i}$ where $m \geq |\bar{Y}|_X = i$. We also have $\uparrow_{\bar{Y}} \uparrow_Z X_{m-i} \equiv_S \uparrow_Z \uparrow_{\bar{Y}} X_{m-i}$ by Lemma 5.3.5. Therefore, $\uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_Z \uparrow_{\bar{Y}} X_{m-i} \equiv_S \uparrow_Z X_m$.
3. Assume $Z \notin \bar{Y}$. If $X \notin \bar{Y}$ then $\uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_Z \uparrow_{\bar{Y}} X_m \equiv_S \uparrow_Z X_m$ using Lemma 5.3.5 and we are done. If $X \in \bar{Y}$ and consequently $X \neq Z$ we can apply (1) and we obtain $\uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S X_m \equiv_S \uparrow_Z X_m$.

4. Follows from

$$\uparrow_{\bar{W}} \uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_{\bar{W}} \uparrow_Z X_m \text{ if } Z \notin \bar{Y} \quad (*)$$

using the induction lemma. To prove (*) we proceed by case analysis:

- (a) Assume $m < |\bar{W}|_X$. Then $\uparrow_{\bar{W}} \uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S X_m$ clearly holds.
- (b) Assume $m \geq |\bar{W}|_X = i$. Then $\uparrow_{\bar{W}} \uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_{\bar{W}} \uparrow_{\bar{Y}} \uparrow_Z X_{m-i} \equiv_S \uparrow_{\bar{W}} X_{m-i} \equiv_S X_m$ using (3) and $Z \notin \bar{Y}$.

5. Special case of (4). □

Lemma 5.3.8 (Simplification III)

- 1. $\uparrow_{\bar{Y}} [Z:=N] X_m \equiv_S X_m$ where $Z \neq X$,
- 2. $\uparrow_{\bar{Y}} [Z:=N] \uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S X_m$,
- 3. $\uparrow_{\bar{Y}} [Z:=N] \uparrow_{\bar{Y}} \uparrow_Z M \equiv_S M$,
- 4. $[Z:=N] \uparrow_Z M \equiv_S M$.

Proof

- 1. If $m < |\bar{Y}|_X$ this follows from Lemma 5.3.6.(1). If $m \geq |\bar{Y}|_X$ it follows from Lemma 5.3.6.(2) and $Z \neq X$.
- 2. We consider two cases:
 - (a) Assume $m < |\bar{Y}|_X$. Then $\uparrow_{\bar{Y}} [Z:=N] \uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_{\bar{Y}} [Z:=N] X_m \equiv_S X_m$.
 - (b) Assume $m \geq |\bar{Y}|_X = i$. Then $\uparrow_{\bar{Y}} [Z:=N] \uparrow_{\bar{Y}} \uparrow_Z X_m \equiv_S \uparrow_{\bar{Y}} [Z:=N] \uparrow_Z X_m$. We distinguish two cases: If $X = Z$ then $\uparrow_{\bar{Y}} [Z:=N] \uparrow_Z X_m \equiv_S \uparrow_{\bar{Y}} [Z:=N] \uparrow_Z X_{m-i} \equiv_S \uparrow_{\bar{Y}} [Z:=N] X_{m-i+1} \equiv_S \uparrow_{\bar{Y}} X_{m-i} \equiv_S X_m$. If $X \neq Z$ then $\uparrow_{\bar{Y}} [Z:=N] \uparrow_Z X_{m-i} \equiv_S \uparrow_{\bar{Y}} [Z:=N] X_{m-i} \equiv_S \uparrow_{\bar{Y}} X_{m-i} \equiv_S X_m$.
- 3. Follows from (2) using the induction lemma.
- 4. Special case of (3). □

Lemma 5.3.9 (Shift Shift Reordering II)

1. $\uparrow_{\bar{W}} \uparrow_Z \uparrow_{\bar{W}} \uparrow_Y M \equiv_s \uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} \uparrow_Z M$,
2. $\uparrow_Z \uparrow_Y M \equiv_s \uparrow_Y \uparrow_Z M$.

Proof (2) is a special case of (1) which follows from

$$\uparrow_{\bar{W}} \uparrow_Z \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_s \uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} \uparrow_Z X_m \quad (*)$$

using the induction lemma.

To prove (*) it is sufficient to show that $\uparrow_{\bar{W}} \uparrow_Z \uparrow_{\bar{W}} \uparrow_Y X_m$ is equivalent to a term that is symmetric, i.e. invariant under exchange of Y and Z . We distinguish the cases:

1. Assume $m < |\bar{W}|_X$. Then $\uparrow_{\bar{W}} \uparrow_Z \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_s \uparrow_{\bar{W}} \uparrow_Z X_m \equiv_s X_m$ which is symmetric.
2. Assume $m \geq |\bar{W}|_X = i$. We have $\uparrow_{\bar{W}} \uparrow_Z \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_s \uparrow_{\bar{W}} \uparrow_Z \uparrow_{\bar{W}} \uparrow_Y X_{m-i} \equiv_s \uparrow_{\bar{W}} \uparrow_Z \uparrow_{\bar{W}} X_{m-i+y} \equiv_s \uparrow_{\bar{W}} \uparrow_Z X_{m+y} \equiv_s \uparrow_{\bar{W}} \uparrow_Z X_{m+y-i} \equiv_s \uparrow_{\bar{W}} X_{m+y-i+z} \equiv_s X_{m+y+z}$ where $y = |Y|_X$ and $z = |Z|_X$. Clearly, X_{m+y+z} is symmetric.

□

Lemma 5.3.10 (Lift Lift Reordering I)

$$\uparrow_Z \uparrow_Y S X_m \equiv_s \uparrow_Y \uparrow_Z S X_m.$$

Proof Trivial for the case where $X = Z$ and $X = Y$. Now we consider the left hand side $\uparrow_Z \uparrow_Y S X_m$ for the remaining cases:

1. $X = Z$ and $X \neq Y$. If $m = 0$ then $\uparrow_Z \uparrow_Y S X_m \equiv_s X_m$. If $m > 0$ then $\uparrow_Z \uparrow_Y S X_m \equiv_s \uparrow_Z \uparrow_Y S X_{m-1} \equiv_s \uparrow_Z \uparrow_Y S X_{m-1}$.
2. $X \neq Z$ and $X = Y$. If $m = 0$ then $\uparrow_Z \uparrow_Y S X_m \equiv_s \uparrow_Z \uparrow_Y S X_m \equiv_s \uparrow_Z X_m \equiv_s X_m$. If $m > 0$ then $\uparrow_Z \uparrow_Y S X_m \equiv_s \uparrow_Z \uparrow_Y S X_m \equiv_s \uparrow_Z \uparrow_Y S X_{m-1}$.
3. $X \neq Z$ and $X \neq Y$. Then $\uparrow_Z \uparrow_Y S X_m \equiv_s \uparrow_Z \uparrow_Y S X_m \equiv_s \uparrow_Z \uparrow_Y S X_m$.

The corresponding results for the right hand side are:

1. $X = Z$ and $X \neq Y$. If $m = 0$ then $\uparrow_Y \uparrow_Z S X_m \equiv_s \uparrow_Y \uparrow_Z S X_m \equiv_s \uparrow_Y X_m \equiv_s X_m$. If $m > 0$ then $\uparrow_Y \uparrow_Z S X_m \equiv_s \uparrow_Y \uparrow_Z S X_m \equiv_s \uparrow_Y \uparrow_Z S X_{m-1}$.

2. $X \neq Z$ and $X = Y$. If $m = 0$ then $\uparrow_Y \uparrow_Z S X_m \equiv_S X_m$. If $m > 0$ then $\uparrow_Y \uparrow_Z S X_m \equiv_S \uparrow_Y \uparrow_Z S X_{m-1} \equiv_S \uparrow_Y \uparrow_Z S X_{m-1}$.
3. $X \neq Z$ and $X \neq Y$. Then $\uparrow_Y \uparrow_Z S X_m \equiv_S \uparrow_Y \uparrow_Z S X_m \equiv_S \uparrow_Y \uparrow_Z S X_m$.

In each of these cases we observe the equivalence of left hand side and right hand side (using Lemma 5.3.9.(2) when needed). \square

Lemma 5.3.11 (Lift Lift Reordering II)

1. $\uparrow_{\bar{W}} \uparrow_Z \uparrow_Y S M \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_Z S M$,
2. $\uparrow_Z \uparrow_Y S M \equiv_S \uparrow_Y \uparrow_Z S M$.

Proof (2) is a special case of (1) which follows from

$$\uparrow_{\bar{W}} \uparrow_Z \uparrow_Y S X_m \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_Z S X_m \quad (*)$$

using the induction lemma. To prove (*) we consider two cases:

1. $m < |\bar{W}|_X$. Then we have $\uparrow_{\bar{W}} \uparrow_Z \uparrow_Y S X_m \equiv_S X_m$ and $\uparrow_{\bar{W}} \uparrow_Y \uparrow_Z S X_m \equiv_S X_m$, both by Lemma 5.3.6.(1).
2. $m \geq |\bar{W}|_X = i$. Then $\uparrow_{\bar{W}} \uparrow_Z \uparrow_Y S X_m \equiv_S \uparrow_{\bar{W}} \uparrow_Z \uparrow_Y S X_{m-i}$ and $\uparrow_{\bar{W}} \uparrow_Y \uparrow_Z S X_m \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_Z S X_{m-i}$. Now (*) follows using Lemma 5.3.10. \square

Lemma 5.3.12 (Special Shift Reordering)

1. $\uparrow_{\bar{W},Y} \uparrow_X \uparrow_{\bar{W}} \uparrow_Y M \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} \uparrow_X M$,
2. $\uparrow_Y \uparrow_X \uparrow_Y M \equiv_S \uparrow_Y \uparrow_X M$,
3. $\uparrow_{\bar{Y}} \uparrow_X \uparrow_{\bar{Y}} M \equiv_S \uparrow_{\bar{Y}} \uparrow_X M$.

Proof (3) is obtained by repeated application of (2) and (2) is a special case of (1) which follows from

$$\uparrow_{\bar{W},Y} \uparrow_X \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} \uparrow_X X_m \quad (*)$$

using the induction lemma. To prove (*) we consider the left hand side and the right hand side separately starting with the left hand side:

1. Assume $m < |\bar{W}|_X$. Clearly, $\uparrow_{\bar{W},Y} \uparrow_X \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_S X_m$.

2. Assume $m \geq |\bar{W}|_X = i$. Defining y as $|Y|_Z$ we have $\uparrow_{\bar{W},Y} \uparrow_X \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_S \uparrow_{\bar{W},Y} \uparrow_X \uparrow_{\bar{W}} \uparrow_Y X_{m-i} \equiv_S \uparrow_{\bar{W},Y} \uparrow_X \uparrow_{\bar{W}} X_{m-i+y} \equiv_S \uparrow_{\bar{W},Y} \uparrow_X X_{m+y} \equiv_S \uparrow_{\bar{W},Y} \uparrow_X X_{m+y-(i+y)} \equiv_S \uparrow_{\bar{W},Y} \uparrow_X X_{m-i}$ since $m+y \geq |\bar{W}, Y|_X = i+y$.

Now observe the equivalence with the right hand side in each of these cases:

1. Assume $m < |\bar{W}|_X$. Clearly, $\uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} \uparrow_X X_m \equiv_S X_m$.
2. Assume $m \geq |\bar{W}|_X = i$. Defining y as $|Y|_Z$ we have $\uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} \uparrow_X X_m \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} \uparrow_X X_{m-i} \equiv_S \uparrow_{\bar{W}} \uparrow_Y X_{m+1} \equiv_S \uparrow_{\bar{W}} \uparrow_Y X_{m+1-i} \equiv_S X_{m+y+1} \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_X X_{m-i} \equiv_S \uparrow_{\bar{W},Y} \uparrow_X X_{m-i} \equiv_S \uparrow_{\bar{W},Y} \uparrow_X X_{m+y}$ since $m+y \geq |\bar{W}, Y|_X = i+y$.

□

The previous lemma is used in a similar proof of the following generalization.

Lemma 5.3.13 (General Shift Reordering)

1. $\uparrow_{\bar{W},Y} S \uparrow_{\bar{W}} \uparrow_Y M \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} S M$,
2. $\uparrow_Y S \uparrow_Y M \equiv_S \uparrow_Y S M$,
3. $\uparrow_{\bar{Y}} S \uparrow_{\bar{Y}} M \equiv_S \uparrow_{\bar{Y}} S M$.

Proof (3) is obtained by repeated application of (2) and (2) is a special case of (1) which follows from

$$\uparrow_{\bar{W},Y} S \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} S X_m \quad (*)$$

using the induction lemma. To prove (*) we consider the left hand side and the right hand side separately starting with the left hand side:

1. Assume $m < |\bar{W}|_X$. Clearly, $\uparrow_{\bar{W},Y} S \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_S X_m$.
2. Assume $m \geq |\bar{W}|_X = i$. Defining y as $|Y|_Z$ we have $\uparrow_{\bar{W},Y} S \uparrow_{\bar{W}} \uparrow_Y X_m \equiv_S \uparrow_{\bar{W},Y} S \uparrow_{\bar{W}} \uparrow_Y X_{m-i} \equiv_S \uparrow_{\bar{W},Y} S \uparrow_{\bar{W}} X_{m-i+y} \equiv_S \uparrow_{\bar{W},Y} S X_{m+y} \equiv_S \uparrow_{\bar{W},Y} S X_{m+y-(i+y)} \equiv_S \uparrow_{\bar{W},Y} S X_{m-i}$ since $m+y \geq |\bar{W}, Y|_X = i+y$.

Now observe the equivalence with the right hand side in each of these cases:

1. Assume $m < |\bar{W}|_X$. Clearly, $\uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} S X_m \equiv_S X_m$.
2. Assume $m \geq |\bar{W}|_X = i$. Defining y as $|Y|_Z$ we have $\uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} S X_m \equiv_S \uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} S X_{m-i}$ and $\uparrow_{\bar{W}} \uparrow_Y \uparrow_{\bar{W}} S X_{m-i} \equiv_S \uparrow_{\bar{W}} \uparrow_Y S X_{m-i}$ by Lemma 5.3.12.(3). Furthermore, $\uparrow_{\bar{W}} \uparrow_Y S X_{m-i} \equiv_S \uparrow_{\bar{W},Y} S X_{m-i} \equiv_S \uparrow_{\bar{W},Y} S X_{m+y}$ since $m+y \geq |\bar{W}, Y|_X = i+y$.

□

Lemma 5.3.14 (Simple Substitution Reordering)

1. $\uparrow_{\bar{W}} S \uparrow_{\bar{W}} [Z:=N] M \equiv_S \uparrow_{\bar{W}} [Z:=S N] \uparrow_{\bar{W},Z} S M$,
2. $S [Z:=N] M \equiv_S [Z:=S N] \uparrow_Z S M$,
3. $[Y:=L] [Z:=N] M \equiv_S [Z:=[Y:=L] N] \uparrow_Z [Y:=L] M$.

Proof (2) and (3) are special cases of (1) which follows from

$$\uparrow_{\bar{W}} S \uparrow_{\bar{W}} [Z:=N] X_m \equiv_S \uparrow_{\bar{W}} [Z:=S N] \uparrow_{\bar{W},Z} S X_m \quad (*)$$

using the induction lemma. To prove (*) we consider the left hand side and the right hand side separately starting with the left hand side:

1. Assume $m < |\bar{W}|_X$. Clearly, $\uparrow_{\bar{W}} S \uparrow_{\bar{W}} [Z:=N] X_m \equiv_S X_m$.
2. Assume $m \geq |\bar{W}|_X = i$. We have $\uparrow_{\bar{W}} S \uparrow_{\bar{W}} [Z:=N] X_m \equiv_S \uparrow_{\bar{W}} S \uparrow_{\bar{W}} [Z:=N] X_{m-i}$. We distinguish three cases:
 - (a) Assume $X \neq Z$. Then $\uparrow_{\bar{W}} S \uparrow_{\bar{W}} [Z:=N] X_{m-i} \equiv_S \uparrow_{\bar{W}} S \uparrow_{\bar{W}} X_{m-i} \equiv_S \uparrow_{\bar{W}} S X_m \equiv_S \uparrow_{\bar{W}} S X_{m-i}$.
 - (b) Assume $X = Z$ and $m - i = 0$. Then $\uparrow_{\bar{W}} S \uparrow_{\bar{W}} [Z:=N] X_{m-i} \equiv_S \uparrow_{\bar{W}} S \uparrow_{\bar{W}} N \equiv_S \uparrow_{\bar{W}} S N$ using Lemma 5.3.13.
 - (c) Assume $X = Z$ and $m - i > 0$. Then $\uparrow_{\bar{W}} S \uparrow_{\bar{W}} [Z:=N] X_{m-i} \equiv_S \uparrow_{\bar{W}} S \uparrow_{\bar{W}} X_{m-i-1} \equiv_S \uparrow_{\bar{W}} S X_{m-i-1}$ using Lemma 5.3.13.

Now we consider the right hand side:

1. Assume $m < |\bar{W}|_X$. Clearly, $\uparrow_{\bar{W}} [Z:=S N] \uparrow_{\bar{W},Z} S X_m \equiv_S X_m$.
2. Assume $m \geq |\bar{W}|_X = i$. Then we have $\uparrow_{\bar{W}} [Z:=S N] \uparrow_{\bar{W},Z} S X_m \equiv_S \uparrow_{\bar{W}} [Z:=S N] \uparrow_{\bar{W}} \uparrow_Z S X_{m-i} \equiv_S \uparrow_{\bar{W}} [Z:=S N] \uparrow_Z S X_{m-i}$ using Lemma 5.3.13. We distinguish three cases:
 - (a) Assume $X \neq Z$. $\uparrow_{\bar{W}} [Z:=S N] \uparrow_Z S X_{m-i} \equiv_S \uparrow_{\bar{W}} [Z:=S N] \uparrow_Z S X_{m-i} \equiv_S \uparrow_{\bar{W}} S X_{m-i}$ using Lemma 5.3.8.(4).
 - (b) Assume $X = Z$ and $m - i = 0$. Clearly, $\uparrow_{\bar{W}} [Z:=S N] \uparrow_Z S X_{m-i} \equiv_S \uparrow_{\bar{W}} [Z:=S N] \uparrow_Z S X_0 \equiv_S \uparrow_{\bar{W}} [Z:=S N] X_0 = \uparrow_{\bar{W}} S N$.
 - (c) Assume $X = Z$ and $m - i > 0$. We have $\uparrow_{\bar{W}} [Z:=S N] \uparrow_Z S X_{m-i} \equiv_S \uparrow_{\bar{W}} [Z:=S N] \uparrow_Z S X_{m-i-1} \equiv_S \uparrow_{\bar{W}} S X_{m-i-1}$ using Lemma 5.3.8.(4). \square

The following lemma states that substitutions S only modifying variables which do not occur in a term M leave the term unchanged. Notice however that its assumption is not necessarily satisfiable if the set of names in our language \mathcal{L} is finite and all of them are used in M . Its proof is a good example of the use of the induction lemma in its general form.

Lemma 5.3.15 (Substitution Elimination)

Assume that Z does not occur in M and S is a substitution of the form $[Z:=N]$ or \uparrow_Z . Then we have

1. $\uparrow_{\bar{W}} S M \equiv_S M$,
2. $S M \equiv_S M$.

Proof

(2) reduces to (1) which follows from

$$\uparrow_{\bar{W}} S X_m \equiv_S X_m \tag{*}$$

assuming $X \neq Z$ using the induction lemma instantiated for the subset of terms Trm' which do not contain Z . To prove (*) we consider the following cases:

1. Assume $m < |\bar{W}|_X$. Then $\uparrow_{\bar{W}} S X_m \equiv_S X_m$ by Lemma 5.3.6.(1).
2. Assume $m \geq |\bar{W}|_X = i$. Then $\uparrow_{\bar{W}} S X_m \equiv_S \uparrow_{\bar{W}} S X_{m-i} \equiv_S \uparrow_{\bar{W}} X_{m-i} \equiv_S X_m$ by considering the two cases for S in the assumption of the lemma. □

5.3.3 Preservation of Confluence

In a typical application context the specification of $\text{CINNI}_{\mathcal{L}}$ is extended by *extra equations*⁸ such as β -reduction in the case of CINNI_{λ} . A natural question to ask is whether the resulting system remains confluent if confluence of the system without explicit substitutions has already been established. So let R be the set of (possibly conditional) equations of the form $\forall X . M = N$ if $\bar{\phi}$. Here M and N are terms possibly containing equational logic variables. In analogy to \Rightarrow_S we define the relation \Rightarrow_R on terms such that $M \Rightarrow_S N$ holds iff our specification

⁸Depending on the intended model-theoretic semantics of the specification (especially regarding the level of abstraction) we can also use extra rules (in the sense of rewriting logic) instead of extra equations. The operational treatment is entirely analogous. We would like to emphasize, however, that the treatment and the results of this chapter focus on extra rules of equational nature, i.e. defining a reduction relation that satisfies the standard context closure property of equality.

has an extra equation $\forall X . L = R$ if $\bar{\phi}$ and there is a kinded substitution θ for X such that $\theta(\bar{\phi})$ is derivable. We denote the context closure of \Rightarrow_R and $\Rightarrow_S \cup \Rightarrow_R$ on terms and substitutions by \rightarrow_R and \rightarrow_{SR} , respectively. We also define a relation \Rightarrow_R on pure, i.e. substitution-free, terms by $M \Rightarrow_R M'$ iff there is an M' such that $M \rightarrow_R M'$ and $M'' = NF_S(M')$. We use \Rightarrow_R as a reference system that operates at a level of abstraction without (observable) explicit substitutions.⁹

Definition 5.3.16 (Well-Formedness)

We say that the set R of equations is *well-formed* iff for each equation in R both of its sides are terms of the (object language) term sort, there are no substitution applications occurring in the left hand side, and for all terms M, N , and substitutions S ,

$$M \Rightarrow_R M' \text{ implies } \exists N' : NF_S(S M) \Rightarrow_R N' \equiv_S S M'$$

It is interesting to note that well-formedness excludes quite different kinds of ill-formed rules. Intuitively, well-formedness expresses compatibility between application of rules and application of substitutions modulo the equations of the substitution calculus. We will show in Corollary 5.3.22 that well-formedness constitutes a sufficient condition to ensure certain coherence properties in the sense of [Vir95, Vir] and [Mes93]. The advantage of our well-formedness condition is that it covers all practically relevant extensions of CINNI by extra equations and that it is usually easy to verify by equational reasoning. Indeed, for a concrete object language this condition can typically be verified using the lemmas given before, as we will show in the examples of Section 5.4.

Lemma 5.3.17 (Well-Formedness Lemma) Assume that R is well-formed. Then for all terms M, N , $q \in \mathbb{N}$, and substitutions S_1, \dots, S_q ,

$$M \Rightarrow_R M' \text{ implies } \exists N' : NF_S(S_q \dots S_1 M) \Rightarrow_R N' \equiv_S S_q \dots S_1 M'$$

Proof

We first prove this lemma for all $q \geq 1$ by induction over n .

Base Case: For $q = 1$ the lemma coincides with the definition of well-formedness.

Induction Step: Our induction hypothesis is: $M \Rightarrow_R M'$ implies $\exists N' : NF_S(S_q \dots S_1 M) \Rightarrow_R N' \equiv_S S_q \dots S_1 M'$ for all $M, M', n \in \mathbb{N}$, and

⁹The relation \Rightarrow_R can be seen as a reduction relation $\rightarrow_{R/S}$ in the terminology of normalized rewriting [Mar96]. In other words we could use the rewrite system defined by R to rewrite modulo the normalizing rewrite system S , but we show in the following that under natural conditions there is no need to impose any strategy (such as normalized rewriting), i.e., S and R can be combined into a single rewrite system which is equivalent to our reference system.

S_1, \dots, S_q . Assume M and M' with $M \Rightarrow_{\mathbf{R}} M'$. The induction hypothesis gives us an N' such that $NF_{\mathbf{S}}(S_q \dots S_1 M) \Rightarrow_{\mathbf{R}} N' \equiv_{\mathbf{S}} S_q \dots S_1 M'$. Applying well-formedness gives us an N'' with $NF_{\mathbf{S}}(S_{q+1} S_q \dots S_1 M) = NF_{\mathbf{S}}(S_{q+1} NF_{\mathbf{S}}(S_q \dots S_1 M)) \Rightarrow_{\mathbf{R}} N'' \equiv_{\mathbf{S}} S_{q+1} S_q \dots S_1 M'$.

Now it remains to prove the lemma for $q = 0$. Assume $M \Rightarrow_{\mathbf{R}} M'$. We have to show that $\exists N' : NF_{\mathbf{S}}(M) \Rightarrow_{\mathbf{R}} N' \equiv_{\mathbf{S}} M'$. We reduce this case to the case $q = 2$ as follows. Choose substitutions $S_1 = [X := X_0]$ and $S_2 = \uparrow_X$ for an arbitrary X (exploiting our general assumption that the set of names is nonempty). Using the lemma for $q = 2$ we obtain N' such that $NF_{\mathbf{S}}(S_2 S_1 M) \Rightarrow_{\mathbf{R}} N' \equiv_{\mathbf{S}} S_2 S_1 M'$ which for our choices is $NF_{\mathbf{S}}([X := X_0] \uparrow_X M) \Rightarrow_{\mathbf{R}} N' \equiv_{\mathbf{S}} [X := X_0] \uparrow_X M'$. Using Lemma 5.3.8.(4) this is equivalent to $NF_{\mathbf{S}}(M) \Rightarrow_{\mathbf{R}} N' \equiv_{\mathbf{S}} N' \equiv_{\mathbf{S}} M'$, which proves our lemma for $q = 0$. \square

As a side note we would like to point out that, unlike calculi like $\lambda\sigma$ [ACCL91], CINNI does not have an identity substitution i.e. a substitution S such that $S M \equiv_{\mathbf{S}} M$ for all M . Of course, adding such a substitution would not cause any difficulties. On the other hand, it is possible to construct an “individual identity substitution” S for each term M such that $S M \equiv_{\mathbf{S}} M$ holds, e.g. using Lemma 5.3.15 if new variables are available, or more generally by an arbitrary substitution which is “lifted sufficiently high” (exploiting Lemma 5.3.6). However, as we can see from the the previous proof, the effect of an identity substitution can be easily achieved using two substitutions $[X := X_0] \uparrow_X$.

Lemma 5.3.18 (Simplification Ordering) It follows from strong normalization (Theorem 5.3.3) that there is a smallest partial order \prec that contains $\rightarrow_{\mathbf{S}}$ and the subterm ordering \sqsubset . This \prec is a simplification ordering, i.e. a well-founded, context-closed partial order that contains the subterm ordering.

Proof Sketch. \sqsubset and $\rightarrow_{\mathbf{S}}$ are context-closed by definition. $\rightarrow_{\mathbf{S}}$ is well-founded by Theorem 5.3.3. \prec is the transitive closure of $(\sqsubset) \cup (\rightarrow_{\mathbf{S}})$ and is therefore context-closed. Since each infinite descending chain of \prec corresponds to an infinite descending chain of $\rightarrow_{\mathbf{S}}$ we conclude that \prec must be well-founded. \square

It is remarkable that the proof of the confluence theorem and in particular the proof of the subsequent projection lemma, both given in [LRD94, BBLRD96] for $\lambda\nu$, generalize to our setting without any difficulties.

Lemma 5.3.19 (Projection) Assume that R is well-formed. Then

1. $M \rightarrow_{\mathbf{R}} M'$ implies $NF_{\mathbf{S}}(M) \Rightarrow_{\mathbf{R}}^* NF_{\mathbf{S}}(M')$ and
2. $S \rightarrow_{\mathbf{R}} S'$ implies $NF_{\mathbf{S}}(S) \Rightarrow_{\mathbf{R}}^* NF_{\mathbf{S}}(S')$,

where $\rightarrow_{\mathbb{R}}$ and NF_S are lifted to substitutions in the natural way.

Proof We prove that $Q \rightarrow_{\mathbb{R}} Q'$ implies $NF_S(Q) \Rightarrow_{\mathbb{R}}^* NF_S(Q')$ for all Q which are terms or substitutions by well-founded induction over Q w.r.t. the simplification ordering \prec .

We first treat the case where Q is a substitution S , i.e., we can assume $S \rightarrow_{\mathbb{R}} S'$. In this case S and S' must be of the form $\uparrow_{\bar{Y}}[X:=N]$ and $\uparrow_{\bar{Y}}[X:=N']$, respectively, and we have $N \rightarrow_{\mathbb{R}} N'$. By induction hypothesis we obtain $NF_S(N) \Rightarrow_{\mathbb{R}}^* NF_S(N')$ and hence $NF_S(S) \Rightarrow_{\mathbb{R}}^* NF_S(S')$.

In the remainder of the proof we deal with the case where Q is a term, i.e., we can assume $M \rightarrow_{\mathbb{R}} M'$. Without loss of generality we can furthermore assume that $M = S_q \dots S_1 N$ for some N which is not a substitution application. Since the left hand sides of equations in R cannot contain substitutions, it follows that the redex must be inside N or inside one of the S_i . Hence M' can also be written as $M' = S'_q \dots S'_1 N'$ and we distinguish the following cases:

1. $N \rightarrow_{\mathbb{R}} N'$ and $S_i = S'_i$ for all $i \in \{1, \dots, q\}$.
 - (a) We first consider the case $N \Rightarrow_{\mathbb{R}} N'$, i.e., a rule is applied at the top of N . By the well-formedness lemma there is a Q' such that $NF_S(S_q \dots S_1 N) \Rightarrow_{\mathbb{R}} Q' \equiv_S (S_q \dots S_1 N')$. It follows that $NF_S(M) \Rightarrow_{\mathbb{R}} Q' \equiv_S NF_S(M')$, which implies $NF_S(M) \Rightarrow_{\mathbb{R}}^* NF_S(Q') = NF_S(M')$, and we are done.
 - (b) Otherwise, i.e. if $N \Rightarrow_{\mathbb{R}} N'$ does not hold, N is of the form $f(P_1, \dots, P_n)$, N' is of the form $f(P'_1, \dots, P'_n)$ and there is an $i \in \{1, \dots, n\}$ such that $f(P_1, \dots, P_n) \rightarrow_{\mathbb{R}} f(P'_1, \dots, P'_n)$ with $P_i \rightarrow_{\mathbb{R}} P'_i$ and $P_j = P'_j$ for all $j \neq i$. Notice that $M = S_q \dots S_1 f(P_1, \dots, P_n) \rightarrow_S^+ f(\uparrow_{\bar{X}_1} S_q \dots \uparrow_{\bar{X}_1} S_1 P_1, \dots, \uparrow_{\bar{X}_n} S_q \dots \uparrow_{\bar{X}_n} S_1 P_n)$ for suitable \bar{X}_i . Notice also that the latter contains $\uparrow_{\bar{X}_i} S_q \dots \uparrow_{\bar{X}_i} S_1 P_i$ as a subterm. Hence we have $NF_S(\uparrow_{\bar{X}_i} S_q \dots \uparrow_{\bar{X}_i} S_1 P_i) \Rightarrow_{\mathbb{R}}^* NF_S(\uparrow_{\bar{X}_i} S_q \dots \uparrow_{\bar{X}_i} S_1 P'_i)$ by the induction hypothesis. Therefore, $NF_S(M) = NF_S(S_q \dots S_1 f(P_1, \dots, P_n)) = f(NF_S(\uparrow_{\bar{X}_1} S_q \dots \uparrow_{\bar{X}_1} S_1 P_1), \dots, NF_S(\uparrow_{\bar{X}_n} S_q \dots \uparrow_{\bar{X}_n} S_1 P_n)) \Rightarrow_{\mathbb{R}}^* f(NF_S(\uparrow_{\bar{X}_1} S_q \dots \uparrow_{\bar{X}_1} S_1 P'_1), \dots, NF_S(\uparrow_{\bar{X}_n} S_q \dots \uparrow_{\bar{X}_n} S_1 P'_n)) = NF_S(S_q \dots S_1 f(P'_1, \dots, P'_n)) = NF_S(M')$.
2. $N = N'$, $S_1 \rightarrow_{\mathbb{R}} S'_1$, and $S_j = S'_j$ for all $j \in \{2, \dots, q\}$. So we have $S_1 \rightarrow_{\mathbb{R}} S'_1$, which implies that $S_1 = \uparrow_{\bar{Z}}[Y:=Q]$ and $S'_1 = \uparrow_{\bar{Z}}[Y:=Q']$ for some \bar{Z}, Y and $Q \rightarrow_{\mathbb{R}} Q'$. We proceed by case analysis on N :
 - (a) N is of the form X_m . Then we have three subcases:
 - i. $X = Y$ and $m = |\bar{Z}|_X = i$. In this case $S_q \dots S_2 \uparrow_{\bar{Z}}[Y:=Q] X_m \rightarrow_S^+ S_q \dots S_2 \uparrow_{\bar{Z}}[Y:=Q] X_0 \rightarrow_S S_q \dots S_2 \uparrow_{\bar{Z}} Q$ and $S_q \dots S_2 \uparrow_{\bar{Z}}$

$[Y:=Q'] X_m \rightarrow_S^+ S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q'] X_0 \rightarrow_S S_q \dots S_2 \uparrow_{\bar{Z}} Q'$. Furthermore, $S_q \dots S_2 \uparrow_{\bar{Z}} Q \rightarrow_R S_q \dots S_2 \uparrow_{\bar{Z}} Q'$ and by the induction hypothesis $NF_S(S_q \dots S_1 N) = NF_S(S_q \dots S_2 \uparrow_{\bar{Z}} Q) \cong_R^* NF_S(S_q \dots S_2 \uparrow_{\bar{Z}} Q') = NF_S(S_q \dots S_1 N')$.

- ii. $X = Y$ and $m > |\bar{Z}|_X = i$. In this case $S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q] X_m \rightarrow_S^+ S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q] X_{m-i} \rightarrow_S^+ S_q \dots S_2 X_{m-1}$ and $S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q'] X_m \rightarrow_S^+ S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q'] X_{m-i} \rightarrow_S^+ S_q \dots S_2 X_{m-1}$. We trivially have $NF_S(S_q \dots S_1 N) = NF_S(S_q \dots S_2 X_{m-1}) \cong_R^* NF_S(S_q \dots S_2 X_{m-1}) = NF_S(S_q \dots S_1 N')$.
- iii. $X = Y$ and $m < |\bar{Z}|_X$. In this case $S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q] X_m \rightarrow_S^+ S_q \dots S_2 X_m$ and $S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q'] X_m \rightarrow_S^+ S_q \dots S_2 X_m$. We trivially have $NF_S(S_q \dots S_1 N) = NF_S(S_q \dots S_2 X_m) \cong_R^* NF_S(S_q \dots S_2 X_m) = NF_S(S_q \dots S_1 N')$.
- iv. $X \neq Y$. In this case $S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q] X_m \rightarrow_S^+ S_q \dots S_2 X_m$ and $S_q \dots S_2 \uparrow_{\bar{Z}} [Y:=Q'] X_m \rightarrow_S^+ S_q \dots S_2 X_m$. We trivially have $NF_S(S_q \dots S_1 N) = NF_S(S_q \dots S_2 X_m) \cong_R^* NF_S(S_q \dots S_2 X_m) = NF_S(S_q \dots S_1 N')$.

- (b) N is of the form $f(P_1, \dots, P_n)$. Notice that $M = S_q \dots S_1 f(P_1, \dots, P_n) \rightarrow_S^+ f(\uparrow_{\bar{X}_1} S_q \dots \uparrow_{\bar{X}_1} S_1 P_1, \dots, \uparrow_{\bar{X}_n} S_q \dots \uparrow_{\bar{X}_n} S_1 P_n)$ for suitable \bar{X}_i . The latter contains $\uparrow_{\bar{X}_i} S_q \dots \uparrow_{\bar{X}_i} S_1 P_i$ as a subterm. Hence we have $NF_S(\uparrow_{\bar{X}_i} S_q \dots \uparrow_{\bar{X}_i} S_1 P_i) \cong_R^* NF_S(\uparrow_{\bar{X}_i} S'_q \dots \uparrow_{\bar{X}_i} S'_1 P_i)$ by the induction hypothesis. Therefore, $NF_S(M) = NF_S(S_q \dots S_1 f(P_1, \dots, P_n)) = f(NF_S(\uparrow_{\bar{X}_1} S_q \dots \uparrow_{\bar{X}_1} S_1 P_1), \dots, NF_S(\uparrow_{\bar{X}_n} S_q \dots \uparrow_{\bar{X}_n} S_1 P_n)) \cong_R^* f(NF_S(\uparrow_{\bar{X}_1} S'_q \dots \uparrow_{\bar{X}_1} S'_1 P_1), \dots, NF_S(\uparrow_{\bar{X}_n} S'_q \dots \uparrow_{\bar{X}_n} S'_1 P_n)) = NF_S(S'_q \dots S'_1 f(P_1, \dots, P_n)) = NF_S(M')$.

- 3. $N = N'$ and there is an $i \in \{2, \dots, q\}$ such that $S_i \rightarrow_R S'_i$ and $S_j = S'_j$ for all $j \neq i$. Clearly, there is an L such that $S_1 N \rightarrow_S^+ L$. Then $S_q \dots S_1 N \rightarrow_S^+ S_q \dots S_2 L$ and as a consequence $S_q \dots S_2 L$ is smaller than M w.r.t. \prec . Since $i > 1$ we have $S_q \dots S_2 L \rightarrow_R S'_q \dots S'_2 L$ and application of the induction hypothesis yields $NF_S(M) = NF_S(S_q \dots S_2 L) \cong_R^* NF_S(S'_q \dots S'_2 L) = NF_S(M')$.

□

Lemma 5.3.20 (Hardin's Interpretation Technique [ACCL91, Har89])

Let S and R be relations on some set T . Assume S is confluent and strongly normalizing and $NF_S(x)$ is the normal form of x w.r.t. S . Assume R_S is a relation on $NF_S(T)$ with $R_S \subseteq (S \cup R)^*$ and $x R y$ implies $NF_S(x) R_S^* NF_S(y)$. Then confluence of R_S implies confluence of $(S \cup R)$.

As a direct consequence of the previous two lemmas we obtain:

Theorem 5.3.21 (Preservation of Confluence)

If R is well-formed and \Rightarrow_R is confluent then \rightarrow_{SR} is confluent.

A noteworthy point is that confluence is *not* reduced to local confluence via Newman's Lemma. Therefore, the previous theorem can also be applied in cases where, like in the λ -calculus, \Rightarrow_R is not strongly normalizing.

A corollary of the projection lemma is the following coherence result, which states that well-formedness implies *weak coherence* in the sense of [Vir95, Vir] (condition (3) below), which is equivalent to a notion proposed independently in [Mes93] (condition (2) below) for the case where the underlying equational theory is not only confluent but also strongly normalizing.

Corollary 5.3.22

If R is well-formed then \rightarrow_R is *weakly coherent* as expressed by each of the following conditions, which are equivalent for confluent and strongly normalizing relations \rightarrow_S .

1. $Q \rightarrow_R Q'$ implies $\exists Q''' . NF_S(Q) \rightarrow_{SR}^* Q''' \equiv_S Q'$,
2. $Q \rightarrow_{SR}^* Q'$ implies $\exists Q''' . NF_S(Q) \rightarrow_{SR}^* Q''' \equiv_S Q'$, and
3. $Q \rightarrow_S^* \rightarrow_R Q'$ and $Q \rightarrow_S^* Q''$ implies that $\exists Q''' . Q'' \rightarrow_S^* \rightarrow_{SR}^* Q''' \equiv_S Q'$.

Proof Sketch. (1) is a direct consequence of Lemma 5.3.19. (1) implies (2), by induction over $Q \rightarrow_{SR}^* Q'$ using (1) for the nontrivial base case. (2) implies (3) by proving $Q'' \rightarrow_S^* NF_S(Q) \rightarrow_{SR}^* Q''' \equiv_S Q'$ under the assumptions of (3). Finally, (1) is a special case of (3) as witnessed by the choosing $Q'' = NF_S(Q)$ in (3) and the fact that $\rightarrow_S^* \rightarrow_{SR}^* = \rightarrow_{SR}^*$. \square

Notice that it is not possible to prove *strong coherence* in the sense of [Vir95, Vir], i.e. to replace \rightarrow_{SR}^* by \rightarrow_R in condition (3) of this theorem, since substitution can lead to a duplication of redexes. By strengthening the projection lemma, however, it would be possible to prove *strong coherence*¹⁰ of a parallel version of \rightarrow_R . More generally, we conjecture that by taking rewrite proofs into account and using a corresponding stronger well-formedness condition it is possible to prove coherence as defined in Section 2.4.5, but using a coarser proof equivalence that corresponds to permutation equivalence in the sense of [LM96]. Since exploring this possibility is beyond the scope of this chapter, we leave it as an interesting direction for future work (see Section 5.6).

¹⁰See also [Pag98] in which, different from the approach presented here, confluence relies on such a strong coherence result and local confluence of a notion of parallel reduction. See Section 5.5 for a brief discussion of [Pag98], which uses a substitution calculus based on de Bruijn indices.

5.4 Applications

In this section we illustrate the use of CINNI to obtain membership equational logic [Mes98, BJM00] specifications of the λ -calculus, Abadi and Cardelli's ζ -calculus [AC96] as well as a rewriting logic [Mes92] specification of Milner's π -calculus [Mil99]. These calculi are interesting, since they have different binding constructs and quite different equational theories. In addition, the π -calculus does not only have an equational theory that specifies process congruence, but there are also rewrite rules to specify the operational semantics in terms of a transition system equipped with an algebraic structure. Both process congruence equations and transition rules make use of substitutions. We show how appropriate instantiations of CINNI can be used in all three cases to obtain formal and executable first-order representations of these languages in Maude [CDE⁺99a, CDE⁺00b]. Furthermore, in each of these examples we give application-specific confluence results that can be obtained using the general results stated earlier. In each example the well-formedness condition can be verified using the lemmas in Section 5.3.2. Since we aim at a unique model in each case, the specifications we give in the following should all be interpreted under the initial semantics.

5.4.1 Higher-Order Functions: The Lambda-Calculus

For the representation of the untyped λ -calculus we use the predefined sort `Qid` to represent names. The following signature defines representations of variables and λ -terms as elements of the sorts `Var` and `Trm`, respectively:

```

sorts Var Trm .
op _{ _ } : Qid Nat -> Var .
subsort Var < Trm .
op __ : Trm Trm -> Trm .
op [_]_ : Qid Trm -> Trm .

vars n m : Nat .  vars X Y Z : Qid .  vars M N : Trm .

```

Here `X{m}` is the representation of a variable, i.e. an indexed name, while `(M N)` and `[X] M` represent application and abstraction, respectively.

The instantiation of CINNI to the syntax of λ -terms, that is CINNI_λ , is given below. `[X := M]`, `[shift X]`, and `[lift X S]` represent simple substitutions, shift substitutions, and lifted substitutions, respectively, and `__` is substitution application.

```

sort Subst .  var S : Subst .

```

```

op [_:=_] : Qid Trm -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Trm -> Trm .

eq ([X := M] (X{0})) = M .
eq ([X := M] (X{suc(m)})) = (X{m}) .
ceq ([X := M] (Y{n})) = (Y{n}) if X /= Y .
eq ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X /= Y .
eq ([lift X S] (X{0})) = (X{0}) .
eq ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X /= Y .

eq S (M N) = (S M) (S N) .
eq S ([X] M) = [X] ([lift X S] M) .

```

Now the explicit substitution version of the β -rule is given by the equation¹¹

```

eq (([X] M) N) = [X := N] M . *** (B)

```

As an example we define the standard combinators:

```

op I K S : -> Trm .
eq I = ['z] 'z{0} .
eq K = ['u] ['v] 'u{0} .
eq S = ['x] ['y] ['z] (('x{0} 'z{0})('y{0} 'z{0})) .

```

The fact that $(S K K) == I$ can be verified by reduction:

```

red (S K K) .
--- rewrites: 47
--- result Trm: ['z]'z{0}

```

¹¹By using an equation we abstract from the operational behavior in the model-theoretic semantics. Of course, it can also be specified as a rule in rewriting logic if we want to make its operational behavior explicit.

Theorem 5.4.1 In the above specification $\text{CINNI}_\lambda + B$, in which the set R of extra equations contains only B , the rewrite relation \rightarrow_{SR} is confluent.

Proof Sketch.

One first has to show confluence of \Rightarrow_{R} , e.g. by adapting and hence generalizing an existing proof based on the standard Tait/Martin-Löf technique [Bar84], such as [Nip01] or [Hue94] for the λ -calculus with de Bruijn indices to λ -calculus with indexed names. It seems to us that generalizing the formalization of [Nip01] in Isabelle [Pau94] would offer the best potential for automating most of the tedious steps of such a proof.

The next step is to establish well-formedness of R , i.e. of the equation B . To this end, it is sufficient to show that there is an N' with $NF_{\text{S}}(S (([X] M) N)) \Rightarrow_{\text{R}} N' \equiv_{\text{S}} S [X:=N]M$. Indeed, $NF_{\text{S}}(S (([X] M) N)) = NF_{\text{S}}((([X] (\uparrow_X S M) (S N)))) = (NF_{\text{S}}([X] (\uparrow_X S M)) NF_{\text{S}}(S N)) = (([X] NF_{\text{S}}(\uparrow_X S M)) NF_{\text{S}}(S N)) \Rightarrow_{\text{R}} [X:=NF_{\text{S}}(S N)] NF_{\text{S}}(\uparrow_X S M) \equiv_{\text{S}} [X:=(S N)] \uparrow_X S M \equiv_{\text{S}} S [X:=N] M$ where the last equivalence is precisely Lemma 5.3.14.(2).

Finally, we are ready to use preservation of confluence (Theorem 5.3.21) to infer confluence of \rightarrow_{SR} from confluence of \Rightarrow_{R} . □

Finally, we would like to briefly discuss the relation to the standard presentation of the λ -calculus, e.g. in the textbook style of [HS90], which can be formalized in membership equational logic [MOM96, CDE⁺00b] using an arbitrary function which provides names that are fresh relative to a given term. By employing such as function it is possible to define renaming, substitution, and to define α -equivalence. By furthermore specifying α -equivalence as an equation as [MOM96, CDE⁺00b] the standard practice of identifying terms modulo this equivalence is reflected.

In CINNI, on the other hand, we do not presuppose α -equality, although as explained in Section 5.2 we can enforce it using a simple equation, which can be even directed in a computationally useful way. In other words, the initial models obtained by these two approaches are isomorphic. The main point however is that even without such an equation CINNI is a meaningful calculus with good metatheoretical properties, allowing us to gain access to a refined world, which is usually hidden behind α -equality. For instance, in the λ -calculus with the standard named notation [HS90], the best confluence result that is possible is confluence modulo α -conversion, since weird renaming has to be compensated for by a notion of equivalence weaker than identity. The fact that the previous theorem states confluence literally in the presence of names, i.e. in its strongest conceivable form, is noteworthy and is made possible by the canonical treatment of names in the CINNI calculus.

5.4.2 Object-Orientation: The Sigma-Calculus

As another application of CINNI we give a first-order representation of the ζ -calculus [AC96]. Just as the λ -calculus can be seen as the most basic model for higher-order functional programming, the object-calculus provides a basic model for object-oriented programming. In contrast to [KL99] which presents a version of the ζ -calculus with explicit substitutions that is based on the standard named representation with α -equality, the following specification is first-order and immediately executable using Maude. Continuing [KL99] a first-order representation of the ζ -calculus with explicit substitutions based on de Bruijn indices has recently been given in [Bon99]. Such a representation also appears in [BKR01b] as an instance of a general approach (see Section 5.5).

In the ζ -calculus an object is seen as a set of attributes, where each attribute has a label and a method. The labels are required to be unique inside an object, since they are used to invoke and update the object's attributes. Below labels, attributes, sets of attributes, methods, objects, and object variables are represented as elements of sorts `Lab`, `Attr`, `Attrs`, `Meth`, `Obj`, and `Var`, respectively.

```

sorts Obj Attr Attrs Meth Lab Var .
var L : Lab .  vars O B O' : Obj .  vars M M' : Meth .
var A : Attr .  var AA : Attrs .  vars X Y Z : Qid .

op ==_ : Lab Meth -> Attr .

op emptyAttrs : -> Attrs .
subsort Attr < Attrs .
op _,_ : Attrs Attrs -> Attrs [assoc comm id: emptyAttrs].

op _{ } : Qid Nat -> Var .
subsorts Var < Obj .
op { } : Attrs -> Obj .
op method[_]_ : Qid Obj -> Meth .

op .._ : Obj Lab -> Obj .
op .._:=_ : Obj Lab Meth -> Obj .

```

Notice that in contrast to the λ -calculus we make use of structural equations such as associativity, commutativity and identity laws (expressed by operator attributes in Maude) in the definition of the syntax of the ζ -calculus.

The last two operators are method invocation and method update, written as `O . L` and `O . L := M`, respectively. In order to define these operations we need a notion of substitution. So we instantiate the CINNI calculus to obtain the following specification of CINNI_ζ :

```

sort Subst .  var S : Subst .  vars n m : Nat .

op [_:=_] : Qid Obj -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .

op __ : Subst Obj -> Obj .
op __ : Subst Meth -> Meth .
op __ : Subst Attr -> Attr .
op __ : Subst Attrs -> Attrs .

eq ([X := 0] (X{0})) = 0 .
eq ([X := 0] (X{suc(m)})) = (X{m}) .
ceq ([X := 0] (Y{n})) = (Y{n}) if X /= Y .
eq ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X /= Y .
eq ([lift X S] (X{0})) = (X{0}) .
eq ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X /= Y .

eq S (L = M) = (L = S M) .
eq S emptyAttrs = emptyAttrs .
ceq S (AA, AA') = (S AA), (S AA') if
  AA /= emptyAttrs and AA' /= emptyAttrs .
eq S ({AA}) = {S AA} .
eq S (method [X] B) = method [X] ([lift X S] B) .
eq S (O . L) = (S O) . L .
eq S (O . L := M) = (S O) . L := S M .

```

Notice that `__` is overloaded, i.e., we have a substitution application operator for each syntactic kind. As a slight optimization we eliminated the application of substitutions to labels. Another noteworthy point is that we have a condition in the syntax-specific equation for application of substitutions to attribute sets in order to avoid nontermination in the operational semantics.

Now method update (MU) and method invocation (MI) can be defined as follows:

```

eq {L = M, AA} . L := M' = {L = M', AA} . *** (MU)

eq {L = method [X] B, AA} . L =
  [X := {L = method [X] B, AA}] B .      *** (MI)

```

Theorem 5.4.2 In the above specification $\text{CINNI}_\zeta + \text{MU} + \text{MI}$, in which the

set R of extra equations consists of MU and MI, the relation \rightarrow_{SR} is confluent modulo the structural equations.

Proof Sketch.

We identify terms which are equivalent by virtue of structural equations.

Confluent modulo α -equality is stated in [AC96] for the ζ -calculus with the two equations MU and MI. The proof is not essentially different from the corresponding proof for the λ -calculus, since it is based on the same Tait/Martin-Löf technique. A similar proof can be conducted for an isomorphic representation of the ζ -calculus based on de Bruin indices, e.g. by using the approach of [Nip01] in Isabelle [Pau94], which in complete analogy to the case of λ -calculus can be adapted and generalized to a representation based on indexed names.

Similar to the proof of Theorem 5.4.1 we next establish well-formedness of R , which is done by considering the following two cases:

1. For the equation MU we have to show that there is an N' with $NF_S(S (\{L=M, AA\} . L := M')) \Rightarrow_R N' \equiv_S S \{L = M', AA\}$. Indeed, $NF_S(S (\{L=M, AA\} . L := M')) = NF_S(\{(S (L=M)), (S AA)\} . L := (S M')) = NF_S(\{L=(S M), (S AA)\} . L := (S M')) \Rightarrow_R \{L=(S M'), (S AA)\} \equiv_S S \{L=M', AA\}$.
2. For the equation MI we have to show that there is an N' with $NF_S(S \{L=\text{method}[X] B, AA\} . L) \Rightarrow_R N' \equiv_S S [X:=\{L=\text{method}[X] B, AA\}] B$. Indeed, $NF_S(S \{L=\text{method}[X] B, AA\} . L) = NF_S(\{(S (L=\text{method}[X] B)), (S AA)\} . L) = NF_S(\{L=(S \text{method}[X] B), (S AA)\} . L) = \{L=\text{method}[X] NF_S(\uparrow_X S B), NF_S(S AA)\} . L \Rightarrow_R [X:=\{L=\text{method}[X] NF_S(\uparrow_X S B), NF_S(S AA)\}] NF_S(\uparrow_X S B) \equiv_S [X:=\{S (L=\text{method}[X] B, AA)\}] (\uparrow_X S B) \equiv_S S [X:=\{L=\text{method}[X] B, AA\}] B$ using Lemma 5.3.14.(2).

Finally, we infer confluence of \rightarrow_{SR} from confluence of \Rightarrow_R by preservation of confluence (Theorem 5.3.21). □

5.4.3 Mobile Processes: The Pi-Calculus

Milner's π -calculus [Mil99] of communicating mobile processes is quite different from the λ - and the ζ -calculi. Here mobility refers to the fact that processes can exchange names of channels and can use them for subsequent communications so that the logical communication topology can evolve dynamically. In the λ -calculus and the ζ -calculus the user is mainly interested in the result of an evaluation. Since both calculi are confluent, such a result is unique if it exists, and can be found by reduction to normal form. In the π -calculus a term is a

collection of possibly interconnected processes, and as a particular case of a reactive system the overall dynamic behavior is relevant. Typically such systems are nonterminating and nondeterministic, and the states that such a system can reach should be clearly distinguished from each other, rather than being identified by equations. So instead of using just membership equational logic as in the λ -calculus and the ζ -calculus, the capabilities of rewriting logic to specify dynamic systems are exploited in the present example. Nevertheless, the equational part, which includes in particular the equationally defined notion of substitution and the process congruence of the π -calculus, will still play a major role.

The π -calculus distinguishes between channels and process terms, which we represent by elements of the sorts **Chan** and **Trm**, respectively. There is an associative and commutative parallel composition $P|Q$ defined on process terms, with the empty process **nil** as identity element. Given a process P , the term $\text{out } CX < CY > . P$ represents a process that sends the channel (name) CY via the channel CX and then continues like P . Notice that this is not a binding construct, whereas the construct $\text{in } CX [Y] P$ binds the name Y in P . It represents a process that receives a channel name via channel CX and then behaves like P with the channel variable Y (that is $Y\{0\}$) substituted by the received channel name. Another binding construct is $\text{new } [X] P$ which declares X to be a local channel w.r.t. P and is also called the hiding construct. The full π -calculus has additional constructs for choice and replication, but the fragment introduced here will be sufficient to explain the application of CINNI in this context.

Related approaches to representing the π -calculus in rewriting logic, that make use of a de Bruijn index based substitution calculi, are given in [Vir96a, Vir96b], [Hir99], and as an instance of the general approach [BKR01b] (see Section 5.5). The first reference is closely related to the presentation given below. It uses a π -calculus version of λv , so that the representation of syntax and the substitution subcalculus arise as a special case of CINNI in the sense explained earlier. On the other hand [Vir96a, Vir96b] covers the full π -calculus with choice and replication, and uses a representation of the operational semantics exploiting rules and strategies. Although other possibilities exist, a CINNI version of the full π -calculus can be obtained by a straightforward adaptation of these rules.

The syntax of the fragment of the π -calculus introduced above is given by the following specification:

```

sorts Chan Trm .
op _{ } : Qid Nat -> Chan .
op nil : -> Trm .
op _|_ : Trm Trm -> Trm [assoc comm id: nil] .
op new[_]_ : Qid Trm -> Trm .
op out_<_>._ : Chan Chan Trm -> Trm .
op in_[]_. : Chan Qid Trm -> Trm .

```

```
vars X Y Z : Qid . vars CX CY CZ : Chan . vars M N P Q : Trm .
```

The instantiation of CINNI for the π -calculus syntax, that is CINNI_π , is given next. Since the π -calculus variables can only range over channels, it is sufficient to have substitutions of variables by channels. As in the specification of the ζ -calculus, $_$ is overloaded, so that substitutions can operate on channels and on process terms.

```
sort Subst . var S : Subst . vars n m : Nat .
```

```
op [_:=_] : Qid Chan -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Chan -> Chan .
op __ : Subst Trm -> Trm .
```

```
eq ([X := CZ] (X{0})) = CZ .
eq ([X := CZ] (X{suc(m)})) = (X{m}) .
ceq ([X := CZ] (Y{n})) = (Y{n}) if X /= Y .
eq ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X /= Y .
eq ([lift X S] (X{0})) = (X{0}) .
eq ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X /= Y .
```

```
eq S nil = nil .
ceq S (M | N) = (S M) | (S N) if N /= nil and M /= nil .
eq S (out CX < CZ > . M) = out (S CX) < S CZ > . (S M) .
eq S (in CX [ Y ] . M) = in (S CX) [ Y ] . ([lift Y S] M) .
eq S (new [X] M) = new [X] ([lift X S] M) .
```

The process congruence is generated by the structural equations for `nil` and `_|_` given above and the following equations NEW1, NEW2 and NEW3 involving the hiding construct. NEW2 and NEW3 are constrained by conditions to avoid nontermination. To this end we presuppose a total order `_<_` on names.

```
eq new [X] nil = nil . *** (NEW1)
ceq (new [X] P) | Q = new [X] (P | [shift X] Q)
    if P /= nil and Q /= nil . *** (NEW2)
ceq new [X] new [Y] P = new [Y] new [X] P if Y < X . *** (NEW3)
```

This completes the part of the specification that defines the process congruence. As stated in the following theorem, which generalizes a corresponding proposition in [Vir96b] in the fragment we consider here, the specification is confluent.

Theorem 5.4.3 In the above specification $\text{CINNI}_\pi + \text{NEW1} + \text{NEW2} + \text{NEW3}$, in which the set R of extra equations consists of NEW1, NEW2, and NEW3, the relation \rightarrow_{SR} is confluent modulo the structural equations.

Proof

We identify terms which are equivalent by virtue of structural equations.

We first establish confluence of \Rightarrow_{R} by an analysis of critical pairs: There is one critical pair between NEW1 and NEW2, but it has a nonsatisfiable condition. Another critical pair exists between NEW2 and NEW3 as witnessed by $((\text{new}[X] \text{new}[Y] P) | Q) \rightarrow_{\text{R}} (\text{new}[X] (\text{new}[Y] P | \uparrow_X Q))$ and $((\text{new}[X] \text{new}[Y] P) | Q) \rightarrow_{\text{R}} ((\text{new}[Y] \text{new}[X] P) | Q)$ if $Y < X$. But this pair does not violate confluence of \Rightarrow_{R} since $(\text{new}[X] (\text{new}[Y] P | \uparrow_Y Q)) \rightarrow_{\text{R}} (\text{new}[X] \text{new}[Y] (P | \uparrow_Y \uparrow_X Q))$ and $((\text{new}[Y] \text{new}[X] P) | Q) \rightarrow_{\text{R}} (\text{new}[Y] ((\text{new}[X] P) | \uparrow_Y Q)) \rightarrow_{\text{R}} \text{new}[Y] \text{new}[X] (P | \uparrow_X \uparrow_Y Q) \rightarrow_{\text{R}} \text{new}[X] \text{new}[Y] (P | \uparrow_X \uparrow_Y Q)$ and the two results are equivalent in terms of \equiv_{S} by virtue of Lemma 5.3.9.(2). Since there are not further critical pairs, \Rightarrow_{R} is confluent.

Now we establish well-formedness of R by considering the following cases:

1. For NEW1 we have to verify that $NF_{\text{S}}(S \text{new}[X] \text{nil}) \Rightarrow_{\text{R}} N' \equiv_{\text{S}} S \text{nil}$ for some N' , which trivially holds since $NF_{\text{S}}(S \text{new}[X] \text{nil}) = NF_{\text{S}}(\text{new}[X] \uparrow_X S \text{nil}) = \text{new}[X] \text{nil} \Rightarrow_{\text{R}} \text{nil} \equiv_{\text{S}} S \text{nil}$.
2. For NEW2 we have to verify that $NF_{\text{S}}(S (\text{new}[X] P) | Q) \Rightarrow_{\text{R}} N' \equiv_{\text{S}} S \text{new}[X] (P | \uparrow_X Q)$ for some N' . Indeed, $NF_{\text{S}}(S (\text{new}[X] P) | Q) = NF_{\text{S}}((S (\text{new}[X] P)) | (S Q)) = NF_{\text{S}}((\text{new}[X] \uparrow_X S P) | (S Q)) = (\text{new}[X] NF_{\text{S}}(\uparrow_X S P)) | NF_{\text{S}}(S Q) \Rightarrow_{\text{R}} \text{new}[X] (NF_{\text{S}}(\uparrow_X S P) | \uparrow_X NF_{\text{S}}(S Q)) \equiv_{\text{S}} \text{new}[X] ((\uparrow_X S P) | (\uparrow_X S Q)) \equiv_{\text{S}} \text{new}[X] ((\uparrow_X S P) | (\uparrow_X S \uparrow_X Q)) \equiv_{\text{S}} \text{new}[X] \uparrow_X S (P | \uparrow_X Q) \equiv_{\text{S}} S \text{new}[X] (P | \uparrow_X Q)$ where we have employed Lemma 5.3.13.(2) once.
3. For NEW3 we have to verify that $NF_{\text{S}}(S \text{new}[X] \text{new}[Y] P) \Rightarrow_{\text{R}} N' \equiv_{\text{S}} S \text{new}[Y] \text{new}[X] P$ for some N' assuming $Y < X$. Indeed, $NF_{\text{S}}(S \text{new}[X] \text{new}[Y] P) = NF_{\text{S}}(\text{new}[X] \text{new}[Y] \uparrow_Y \uparrow_X S P) = \text{new}[X] \text{new}[Y] NF_{\text{S}}(\uparrow_Y \uparrow_X S P) \Rightarrow_{\text{R}} \text{new}[Y] \text{new}[X] NF_{\text{S}}(\uparrow_Y \uparrow_X S P) \equiv_{\text{S}} \text{new}[Y] \text{new}[X] \uparrow_Y \uparrow_X S P \equiv_{\text{S}} \text{new}[Y] \text{new}[X] \uparrow_X \uparrow_Y S P \equiv_{\text{S}} S \text{new}[Y] \text{new}[X] P$, where we have used Lemma 5.3.11.(2).

Finally, we infer confluence of \rightarrow_{SR} from confluence of \Rightarrow_{R} by preservation of confluence (Theorem 5.3.21). □

Communication of two parallel processes via a channel CX can be expressed by the following rule that models an atomic interaction in which $(\text{out } CX < CZ > . P)$ sends the channel name CZ to $(\text{in } CX [Y] . Q)$.

$$\text{r1 } [\text{COMM}] : (\text{out } CX < CZ > . P) \mid (\text{in } CX [Y] . Q) \Rightarrow \\ P \mid [Y := CZ] Q .$$

By restricting the application of the context closure rule of rewriting logic¹² to the process constructors $_|_$ and $\text{new}[_]_$ we make sure that, in conformance with the π -calculus, communication never takes place inside $(\text{out } CX < CZ > . P)$ or $(\text{in } CX [Y] . Q)$.

In [Vir96b] Viry states strong coherence [Vir95] of his specification of the full π -calculus based on explicit substitutions with de Bruijn indices. Coherence of our specification is trivially satisfied, since in view of the restricted context closure rule no applicable critical pairs exist.

5.4.4 Further Applications

A real-world application of CINNI in the context of a programming language for active networks [TSS⁺97] is subject of [MÖST02, ST02a, ST01] and briefly described in Appendix B. Here, CINNI is used to specify an abstract machine for the PLAN language [HKM⁺98b, HKM⁺98a, MHN99, HK99, KHMG99], which is an imperative language similar to ML with a notion of program mobility and other features. We feel that this application of CINNI is interesting, since it shows how a general technique based on CINNI can be used to specify an abstract machine which controls the evaluation of a program, an important requirement in the presence of potential side-effects. In other words, the use of CINNI is not restricted to purely functional languages, but can also be applied to specify the operational semantics of imperative programming languages. Furthermore, this application demonstrates how a generalized version of CINNI, which supports simultaneous explicit substitutions, can be defined on top the the simpler version.

Another application of CINNI that should be placed in the context of higher-order logic and type theory is presented in Chapter 6 (see also [SM99]), where CINNI is instantiated to the family of *pure type systems* [Ber88, Ter89]. Pure type systems generalize the λ -cube [Bar92] and are considered to be of key importance, because their generality and simplicity makes them an ideal basis for representing and implementing higher-order logics.

Last but not least, the CINNI calculus has been applied in the design and implementation of a proof assistant for OCC, the *open calculus of constructions*, an

¹²In Maude a nonstandard context closure rule is reflected operationally by using a suitable execution strategy which only rewrites under operators for which the rule is assumed.

extension of the calculus of constructions [CH88] that incorporates equational and rewriting logic as computational sublanguages. OCC supports conditional equations and conditional assertions together with an operational semantics based on conditional rewriting modulo equations. In fact, based on the CINNI calculus we have developed an experimental Maude prototype of OCC that makes use of Maude's reflective capabilities to evaluate higher-order equational specifications and programs with reasonable efficiency. We use this prototype in a detailed presentation of OCC which is the subject of Chapter 8.

5.5 Three Orthogonal Research Directions

To place our work in the context of research conducted by other authors we describe in the following what could be visualized as a (partial) *cube of explicit substitution calculi*, namely an informal classification of three orthogonal research directions concerned with explicit substitution calculi and their (potential) combinations (cf. Figure 5.1). We restrict our attention here to first-order calculi. We consider the minimal and well-investigated substitution calculus λv as a reference point and we distinguish three orthogonal directions of research:

1. Generalizing the object language by metavariables to represent not only closed but also open terms (enrichment).
2. Generalizing the fixed syntax and computation rules of the object language from the λ -calculus to languages with arbitrary syntax and computation rules (parameterization).
3. Generalizing the underlying representation based on de Bruijn's indices to Berkling's representation allowing for an explicit representation of names (enrichment).

Although we are not concerned with the direction (1) in this chapter, this was historically the first direction investigated. The idea of dealing with open terms, i.e. with terms having metavariables, is already present in $\lambda\sigma$ [ACCL91] and λ_{\uparrow} [CHL96], from which λv can be derived as a subcalculus [Les94]. Conversely, λv can be extended by a composition operator and corresponding rules to obtain λ_{\uparrow} . Our main motivation for using λv as a starting point is its minimality and its logical completeness, in the sense that standard properties of composition are inductive consequences. By definition, the terms of λv (and CINNI) do not contain metavariables, hence confluence means ground-confluence in this context. If terms with metavariables are considered, confluence does not hold anymore in the core calculus. The extension λ_{\uparrow} is confluent on open terms, but does not preserve strong normalization (even on closed terms), a property that has, however, been

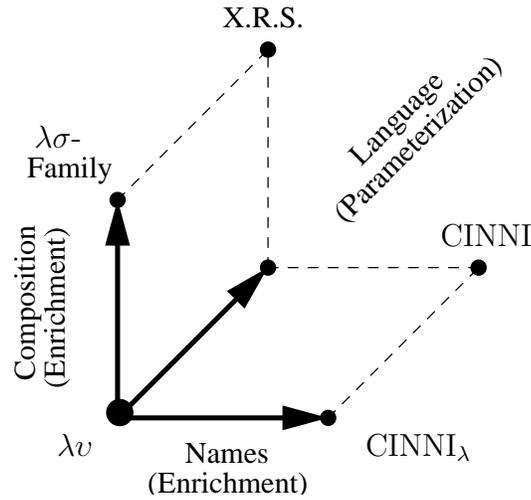


Figure 5.1: A partial cube of explicit substitution calculi

established for $\lambda\nu$ [LRD94, BBLRD96]. On the other hand, confluence on open terms is not needed for many important applications such as execution (see Section 5.4 and Appendix B) or type checking and inference (see Chapter 6). This motivates our choice of the minimal calculus $\lambda\nu$, which enjoys good metatheoretic properties on terms without metavariables, as a reference point in this chapter.

Direction (2) has been pursued first in the context of combinatory reduction systems [BR96], and later as a generalization of λ_\uparrow in [Pag98]. The work [BR96] uses combinatory reduction systems [Klo80], which emerged from Aczel’s general treatment of confluence in [Acz78]. Although binding and substitution are central concepts in combinatory reduction systems, they are not first-order and by abstracting from the choice of names they belong to a level of abstraction higher than the one we deal with in this chapter. The first-order approach presented in [Pag98], on the other hand, is closely related to our work, in the sense that it investigate a coherence condition similar to what we have called well-formedness. However, the author does not aim at *preservation* of confluence results as those presented in this chapter. Instead, he investigates a number of sufficient conditions to *ensure* confluence. Also, preservation of strong normalization, which we consider as an important future extension of our work, is impossible in [Pag98], since already λ_\uparrow does not have this property. Another important point is that [Pag98] does not consider explicit names. Loosely speaking, [Pag98] investigates the directions (1) and (2), whereas we deal with the directions (2) and (3). Of course, an interesting question is whether (1), (2) and (3) can still be combined in a reasonable way (possibly disregarding the problem of strong normalization).

Direction (3) appears to be a natural direction in the context of explicit substitution calculi that has not been explored so far. Instead of de Bruijn’s representation it makes use of Berkling’s representation [Ber76, BF82] as a basis of an

explicit substitution calculus. In view of the clear advantages, it is surprising that the CINNI calculus seems to be the first explicit substitution calculus based on this representation. It might appear that the generalization is rather straightforward, but this is definitely not true for the metatheory as indicated by our strong normalization result, which cannot be proved by just reusing the techniques of [BBLRD96, LRD94]. This indicates that CINNI is a nontrivial generalization that deserves a careful study. Furthermore, we think that the difference becomes even more challenging if other calculi such as $\lambda\sigma$ and λ_{η} are used as a starting point for the direction (3).

The view of $\lambda\nu$ as a minimal base calculus, especially for Direction (2), also nicely corresponds to a recent unified treatment [Kes00] of the confluence problem (for terms without metavariables, as in the present chapter) for all of the most well-known explicit substitution calculi based on de Bruijn indices. The main idea is to abstract a minimal set of properties which are needed for the proofs and use them to define a scheme for substitution calculi. The operators of $\lambda\nu$ which can be mapped to other calculi in different ways are central to this scheme. Again this line of research is orthogonal to our work, in the sense that it uses de Bruijn indices instead of indexed names, and in the sense that it considers the λ -calculus (including its extension by different η -rules) instead of being parameterized by the object language. It is needless to say that a unification of both lines of research would be highly desirable and that it is closely connected to the question how other substitution calculi can be generalized to indexed names.

The desire to obtain a general first-order theory applicable to a wide range of object languages in the sense of Direction (2) has motivated the development of a notion of *higher-order rewriting*¹³ in de Bruijn representation [BKR01a], which later has been refined by using explicit substitution calculi in [BKR01b] (in a similar style as the work [Kes00]). Another initial motivation for this very systematic approach was the unclear relation between the work of [Pag98] and other approaches to higher-order rewriting. In contrast to unconditional higher-order rewriting our approach is to use conditional first-order rewriting as the starting point. This is different from [Pag98], who only considered first-order equations without conditions. It is noteworthy that in our treatment of CINNI we have not imposed any restrictions on the conditions allowed in extra equations, even if we have used membership equational logic in the examples of this chapter. Examples with conditions that cannot be expressed in membership equational logic but are expressible in OCC will be given in Chapter 8 (see Section 8.2.4). Again we consider the comparison and the potential unification of our approach and [BKR01b] as an important subject of future work,¹⁴ since higher-order rewriting is becom-

¹³We use this term in the broad sense, rather than referring to a particular formalism.

¹⁴It might also be possible to use the CINNI approach to replace the two formalisms for higher-order rewriting presented in [BKR01b], one based on names and one based on de Bruijn indices, by a single unified one.

ing increasingly popular, and the direct use of a named notation, rather than a translation between internal and external representation, offers some advantages.

Before we conclude the discussion of related work we would like to point out that we have restricted our attention to (first-order) explicit substitution calculi in this chapter, but there are several other approaches which are not based on explicit substitutions, such as binding structures [Tal93] and higher-order abstract syntax [PE88], that are both worth mentioning.

Higher-order abstract syntax [PE88] uses a typed λ -calculus as a metalanguage to represent an object language via a shallow embedding that maps the binding constructs of the object language to the binding construct of the metalanguage, i.e. λ -abstraction. This is interesting in the sense that several object languages can be represented in terms of a single one, although it does not solve the core problem of binding but merely delegates it to the metalevel. This approach has the usual advantages of a shallow embedding, namely that the machinery of the metalanguage, is reused for the object language, but its disadvantages include its limited flexibility, the need for a higher-order metalanguage, and certain semantic difficulties (which appear in the context of inductive theorem proving) that are connected with a too tight coupling between object- and metalanguage. Compared with CINNI, such an approach necessarily abstracts from names, a feature that is inherited from the λ -calculus of the metalanguage. We further discuss this approach, which has been successfully used in the context of higher-order logical frameworks [Pfe91, Pfe96], higher-order logic programming languages [NM98] and theorem provers [DFH95, DH94], in Chapter 8 (see Section 8.2.4), where we show how higher-order abstract syntax can be used to represent object languages in the higher-order equational logic of OCC. In the same section we show how CINNI can be used in the context of OCC in a similar way, but without the potential problems caused by a higher-order representation.

The other line of work, that is quite different from all these approaches, introduces *binding structures* [Tal93], which can be seen as abstract data structures encapsulating two important concepts, namely object-variable substitution and hole filling (which can be regarded as metavariable substitution). Furthermore, binding structures have a well-developed theory which is based on a general principle to define homomorphisms such as substitution, unbinding, filling, etc. An obvious difference to our approach is that hole-filling and, more generally, metavariable substitution is not part of CINNI, although it can be easily defined on top of CINNI when needed. In fact, metavariable substitution coincides with the trivial notion of textual substitution (see the applications of CINNI in Chapter 6 and Appendix B). Another difference is that binding structures abstract from the names of bound variables using de Bruijn indices. More recently, a named version of binding structures has been proposed in [Mas99] (see also [Tal91] for an earlier approach to binding structures with names). From the viewpoint of CINNI, a unification of these approaches seems possible. For instance, one could

use CINNI as an abstract term representation with object-variable substitutions and add metavariables for the representation of holes. In this setting it should be possible to develop a metatheory generalizing that of [Tal93] and [Mas99], including a similar definition principle for homomorphisms on CINNI terms, which essentially corresponds to an elimination principle in the sense of type theory (see Chapter 8).

5.6 Final Remarks

The main contribution of this chapter is the introduction of a generic first-order calculus of explicit substitutions that combines the advantages of both named and indexed notation in a natural way. The fact that our approach contains $\lambda\nu$ as a special case is of great help for developing the metatheory, since most of the statements and proofs of [LRD94, BBLRD96] can be fruitfully (but nontrivially) generalized to our setting.

An important direction for future research is the issue of preservation of strong normalization. We conjecture that under a very liberal condition, which is practically not much more restrictive than our present well-formedness condition but ensures preservation of redexes in a certain sense, it is possible to generalize and modularize the proof of preservation of strong normalization given in [BBLRD96] in a way that allows us to deduce preservation of strong normalization for CINNI representations of object languages different from the λ -calculus.

Another challenge is to extend CINNI by adding a notion of composition in the style of the $\lambda\sigma$ -family, although it appears that composition is not compatible with preservation of strong normalization even for indexed-based calculi [Mel95]. On the other hand, the more recent approach of λs_e [KR97] shows how confluence on open terms can be obtained without explicit composition, and it seems that a unification of CINNI with the λs -family [KR00] seems to be possible and worth investigating.¹⁵ Unfortunately, according to a recent result [Gui00] preservation of strong normalization does not hold for λs_e either. Although there are two other calculi enjoying the properties of confluence (on terms with metavariables), simulation of 1-step β -reduction, and preservation of strong normalization, [GGL00, GL99, DG01], they both rely on nontrivial/nonstandard representations of λ -terms, which makes these solutions not particularly attractive in view of our main objective of reducing the gap between object languages and their formal representation.

To conclude, we would like to point out that the CINNI approach is a very natural application of the main idea behind rewriting logic, namely that we investigate the

¹⁵Indeed, the λs -style, the $\lambda\sigma$ -style calculi are closely related as shown for instance by the bridge calculus $\lambda\omega$ [KR00], which is also interesting in this context.

notion of rewriting modulo an underlying equational theory which in this case is the CINNI calculus. This suggests a further possible direction that is worthwhile to pursue, namely to study the explicit notion of proof offered by rewriting logic with CINNI as the underlying equational theory and to add suitable generic axioms that lead to a complete axiomatization of permutation equivalence in the sense of [LM96], where such axioms are proposed for $\lambda\sigma$ and for combinatory reduction systems. In such a setting it is possible to study a strong form of confluence which can be expressed in terms of a sequential composition of proofs which satisfies a simple commutation property corresponding to the diagram of confluence (see [LM96] for details).

5.7 Acknowledgements.

The work presented in this chapter has been supported by DARPA through Rome Laboratories Contract F30602-C-0312, by DARPA and NASA through Contract NAS2-98073, by Office of Naval Research Contract N00014-99-C-0198, and by National Science Foundation Grant CCR-9900334. The research reported in Appendix B has furthermore been supported by the DARPA active network program. I would like to thank José Meseguer for his advice, including his early suggestion to represent higher-order languages in rewriting logic using explicit substitutions, Cesar Muñoz for many discussions on calculi of explicit substitutions, especially in connection with type theory and proof synthesis, Carolyn Tallcott for discussions on binding structures, substitutions, and fruitful cooperation in the context of the DARPA program, as well as Narciso Martí-Oliet for many useful suggestions and his constructive criticism. I am grateful for the feedback from participants of a project/seminar “Rewriting as a Meta-Paradigm” given at the University of Hamburg in 2000, where CINNI was applied in connection with Maude to represent different object languages. I also appreciate the comments of several referees for the Rewriting Logic Workshop 2001 in Japan, which helped to improve the paper [Ste00a]. Furthermore, I am indebted to an anonymous referee for pointing out that the representation used by CINNI for pure terms is equivalent to Berklings’ representation, and for providing me with the corresponding references. Last but not least, I also received encouraging feedback from Delia Kesner who provided us with a long list of additional references and an explanation of their connections that I tried to reflect in my treatment, and from Alejandro Ríos, Eduardo Bonelli, and Ariel Arbiser, who together carefully studied the extended version of [Ste00a] and gave me lots of constructive comments and detailed suggestions, that I also incorporated into the present chapter. I am in particular grateful for several corrections and their suggestion to complete the earlier presentation by adding Lemmas 5.3.12 and 5.3.15. I highly appreciate all this feedback and realize that there are a number of challenging research

directions, especially connected with previous work by these and other authors, that I at least partly hope to be able to address in future work.

Chapter 6

Rewriting Logic as a Logical Framework:

Representing Pure Type Systems

This chapter is a detailed study on the ease and naturalness with which a family of higher-order formal systems, namely *pure type systems (PTSs)* [Ber88, Ter89], can be represented in the first-order logical framework of rewriting logic [Mes92, MOM94, MOM96]. PTSs generalize the λ -cube [Bar92], which already contains important calculi like $\lambda \rightarrow$ [Chu40], the systems F [Gir72, Rey74] and $F\omega$ [Gir72], a system λP close to the logical framework LF [HHP87], and their combination, the calculus of constructions (CC) [CH88]. PTSs are considered to be of key importance, since their generality and simplicity makes them an ideal basis for representing higher-order logics, either via the propositions-as-types interpretation [Geu93], or via their use as a higher-order logical framework in the spirit of LF [HHP87, Gar92] or Isabelle [Pau94].

Exploiting the fact that RWL and its MEL sublogic [BJM97] have initial and free models, we can define the representation of PTSs as a *parameterized theory* in the framework logic; that is, we define in a single parametric way all the representations for the infinite family of PTSs. Furthermore, the representational versatility of RWL, and of MEL, are also exercised by considering four different representations of PTSs at different levels of abstraction, from a more abstract textbook version in which terms are identified up to α -conversion, to a more concrete version with a calculus of names and explicit substitutions, and with a type checking inference system that can in fact be used as a reasonably efficient implementation of PTSs by executing the representation in the Maude language [CDE⁺99a, CDE⁺00b].

This case study complements earlier work [MOM94, MOM96], showing that rewriting logic has good properties as a logical framework to represent a wide range of logics, including linear logic, Horn logic with equality, first-order logic, modal logics, sequent-based presentations of logics, and so on. In particular, representations for the λ -calculus, and for binders and quantifiers have already been studied in [MOM96], but this is the first systematic study on the representation of *typed* higher-order systems. One property shared by all the above representations, including all those discussed in this chapter, is that what might be called the *representational distance* between the logic being formalized and its rewriting logic representation is virtually zero. That is, both the syntax and the inference system of the object logic are directly and faithfully mirrored by the representation. This is an important advantage both in terms of understandability of the representations, and in making the use of encoding and decoding functions unnecessary in a so-called adequacy proof.

Besides the directness and naturalness with which logics can be represented in a framework logic, another important quality of a logical framework is the *scope* of its applicability; that is, the class of logics for which faithful representations preserving relevant structure can be defined. Typically, we want representations that both preserve and reflect provability; that is, something is a theorem in the original logic if and only if its translation can be proved in the framework's

representation of the logic. Such mappings go under different names and differ in their generality; in higher-order logical frameworks representations are typically required to be *adequate* mappings [Gar92], and in the theory of general logics more liberal, namely *conservative* mappings of entailment systems [Mes89a], are studied. In Chapter 2 we have further generalized conservative mappings to the notion of a sound and complete total *correspondence of sentences* between two entailment systems. In fact, all the representations of PTSs that we consider are correspondences of this kind. Sound and complete total correspondences are systematically used not only to state the correctness of the representations of PTSs at different levels of abstraction, but also to relate those different levels of abstraction, showing that the more concrete representations correctly implement their more abstract counterparts.

A systematic way of comparing the scopes of two logical frameworks \mathcal{F} and \mathcal{G} is to exhibit a sound and complete total correspondence $\mathcal{F} \rightsquigarrow \mathcal{G}$, representing \mathcal{F} in \mathcal{G} . In view of this quite general concept, it is important to add that the *representational distance*, which we informally define as the complexity of this correspondence, is an important measure of the quality of the representation. Since such correspondences form a category, and therefore compose, this then shows that the scope of \mathcal{G} is *at least as general* as that of \mathcal{F} . Since PTSs include the system λP , close to the logical framework LF, and the calculus of constructions CC, the results in this chapter indicate that the scope of rewriting logic is at least as general as that of those logics. Furthermore, since there are no adequate mappings from linear logic to LF in the sense of [Gar92], but there is a conservative mapping of logics from linear logic to rewriting logic [MOM96], this seems to indicate that the LF methodology together with its rather restrictive notion of adequate mapping is more specialized than the rewriting logic approach.

In this chapter we will be concerned with PTSs as formal systems represented inside informal set theory, or inside another formal system such as rewriting logic or its membership equational sublogic. For formal systems in general, and for PTSs in particular, there is not a single canonical presentation. Instead each presentation is tailored for specific purposes. For example, there are different formulations of PTSs with different sets of rules, but the same sets, or related sets, of derivable sentences. Furthermore, presentations can be more or less abstract, e.g. concerning the treatment of names, or concerning the degree of operationality. It is needless to say that the use of some general terminology is highly desirable in this situation to deal with these issues in a systematic way. To this end, we follow the general logics methodology [Mes89a] to use an abstract logical metatheory, which is concerned with formal systems *and* their relationships, together with a particular formal system as a logical framework, namely rewriting logic. Regarding general logics terminology, we furthermore found that the notion of correspondences between sentences introduced in Chapter 2 that generalizes the idea of maps of entailment systems is a simple a useful tool to structure the

results of the present chapter.

6.1 Overview and Main Results

In Section 6.2 we show how the definition of PTSs can be formalized in MEL. The approach we use is not only less specialized than the one used in a higher-order logical framework like LF [HHP87] or Isabelle [Pau94], but it has also more explanatory power, since we explain higher-order calculi in terms of a first-order system with a simpler semantics, and our representations have initial (or, more generally, free extension) models supporting metalogical inductive reasoning about the PTSs thus represented.

In order to make the specification of PTSs more concrete, we introduce in Section 6.2.3 the notion of *uniform pure type systems (UPTSs)* [Ste99], that do not abstract from the treatment of names but use CINNI [Ste00a], the first-order calculus of names and substitutions that we introduced in Chapter 5. UPTSs solve the problem of closure under α -conversion, that has been discussed by Pollack in [Pol93], in a simple and elegant way. Again, a MEL specification of UPTSs is given that directly formalizes the informal definition.

As an intermediate step we employ *optimized UPTSs (OUPTSs)* which are introduced in Section 6.2.4. OUPSTs have an explicit judgement for well-typed contexts, and can be seen as a refinement of UPTSs towards a more efficient implementation of type checking.

Last but not least, we describe how the meta-operational view of an important class of OUPSTs, namely type checking and type inference, can be expressed as a transition system and can likewise be formalized in rewriting logic. The result of this formalization is an executable specification of *rewriting-based OUPSTs (ROUPSTs)* that is sound w.r.t. the logical specification given before in a very obvious way.

Formally, these different presentations of PTSs are families of unary entailment systems parameterized by *PTS signatures*. We use the notation \mathbf{PTS}_S , \mathbf{UPTS}_S , \mathbf{OUPTS}_S and \mathbf{ROUPST}_S to denote the entailment systems of PTSs, UPTSs, OUPSTs, and ROUPSTs, respectively, associated with a PTS signature S .

For appropriate PTS signatures S we obtain a chain of sound and complete total correspondences (see Section 2.3 for the definition of this concept)

$$\mathbf{PTS}_S \curvearrowright \mathbf{UPTS}_S \curvearrowright \mathbf{OUPTS}_S \curvearrowright \mathbf{ROUPST}_S.$$

Actually, we have two different kinds of connections between the first two entailment systems, leading to two different correspondences of the form $\mathbf{PTS}_S \curvearrowright \mathbf{UPTS}_S$. By composing three correspondences of the form above we finally arrive

at a sound and complete total correspondence

$$\mathbf{PTS}_S \curvearrowright \mathbf{ROUPTS}_S$$

which shows the equivalence of the high-level specification of PTSs with the implementation of a type checker.

The deductive system of RWL induces a unary entailment system **RWL** with sentences of the form $\mathcal{R} \vdash \phi$, where \mathcal{R} is a rewrite theory and ϕ is an equation, a membership or a rewrite. In this chapter we abstract from rewrite proofs, so that we use the term rewrite to refer to abstract rewrites of the form $M \rightarrow M'$ and we define $\mathcal{R} \vdash M \rightarrow M'$ to be derivable iff $\mathcal{R} \vdash P : M \rightarrow M'$ is derivable for some rewrite proof P in the deductive system of RWL. Likewise, MEL induces a unary entailment system **MEL** obtained by restricting \mathcal{R} to MEL theories and ϕ to equations or memberships.

The entailment systems **PTS_S**, **UPTS_S**, **OUPS_S** and **ROUPTS_S** can be easily specified in membership equational logic or in rewriting logic. Specifically, we have the following sound and complete total correspondences:

$$\mathbf{PTS}_S \curvearrowright \mathbf{MEL}$$

$$\mathbf{UPTS}_S \curvearrowright \mathbf{MEL}$$

$$\mathbf{OUPS}_S \curvearrowright \mathbf{MEL}$$

$$\mathbf{ROUPTS}_S \curvearrowright \mathbf{RWL}$$

In all cases the *representational distance* between the formal system and its representation is practically zero, that is, both the syntax and the inference system of each version of PTSs have direct and faithful representations in the framework logic.

The first correspondence is the representation of PTSs in MEL given in Section 6.2. Let $\overline{\mathbf{PTS}}_S$ be the MEL specification of **PTS_S**. Then, for all PTS judgements ϕ of **PTS_S** and possible representations ϕ' of ϕ in MEL, the sentence $\overline{\mathbf{PTS}}_S \vdash \phi'$ is derivable in MEL iff the judgement ϕ is derivable in **PTS_S**. This defines a sound and complete total correspondence of the form $\mathbf{PTS}_S \curvearrowright \mathbf{MEL}$. We are concerned with a correspondence rather than a function, due to the fact that PTSs abstract from names, but in the MEL representation names are part of the description of terms, although by adding appropriate equations an equivalent abstraction can be achieved in MEL at the semantic level.

In the remaining three systems **UPTS_S**, **OUPS_S**, and **ROUPTS_S** we do not abstract from names. Hence, the three associated representational correspondences actually take the form of functions, i.e., with each judgement of the type system we can associate a unique sentence in MEL or RWL, respectively. For the presentation of PTSs we follow [vBJMP93], which can be seen as an informal presentation of the machine-checked formalization [MP93].

6.2 The Metalogical View of PTSs

A *PTS signature* is a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where \mathcal{S} is a set of *sorts*, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is the set of *axioms*, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is the set of *rules*. The sorts of a PTS signature are used as types of types and are therefore often referred to as *universes*. We use S to range over PTS signatures, and for the following we fix an arbitrary PTS signature S .

In PTSs there is no a priori distinction between terms and types. *PTS terms* are defined by the following syntax with binders:

$$X \mid M \ N \mid [X : A]M \mid \{X : A\}M \mid s$$

Here, and in the following, s ranges over \mathcal{S} ; M, N, A, B, T range over terms; and X ranges over names. We should add that in $[X : A]M$ and $\{X : A\}M$ the name X is bound in M , and we assume that α -convertible terms, i.e. terms that are equal up to renaming of bound variables, are identified.

Formally this identification can be achieved by different means: the definition of PTS terms as equivalence classes modulo α -equivalence, or a representation based on de Bruijn indices are two possibilities. For the following it is important to keep in mind that the choice of particular names for bound variables is part of the informal notation (for readability) but is not reflected in PTS terms.

A *PTS context* is a list of *declarations*, each of the form $X : A$. A declaration $X : A$ *declares* a name X of type A . A context is *simple* if it declares each identifier at most once. In the following, Γ ranges over PTS contexts.

PTS typing judgements are of the form $\Gamma \vdash M : T$, and *derivability*, i.e. the set of *derivable typing judgements*, is defined by the formal system given by the following inference rules:

$$\frac{}{\emptyset \vdash s_1 : s_2} \quad (s_1, s_2) \in \mathcal{A} \quad (\text{Ax})$$

$$\frac{\Gamma \vdash A : s}{\Gamma, X : A \vdash X : A} \quad X \notin \Gamma \quad (\text{Start})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, X : B \vdash M : A} \quad X \notin \Gamma \quad (\text{Weak})$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash B : s_2}{\Gamma \vdash \{X : A\}B : s_3} \quad (s_1, s_2, s_3) \in \mathcal{R} \quad (\text{Pi})$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash M : B \quad \Gamma, X : A \vdash B : s_2}{\Gamma \vdash [X : A]M : \{X : A\}B} \quad (s_1, s_2, s_3) \in \mathcal{R} \quad (\text{Lda})$$

$$\frac{\Gamma \vdash M : \{X : A\}B \quad \Gamma \vdash N : A}{\Gamma \vdash (M N) : [X := A]B} \quad (\text{App})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad A \equiv_{\beta} B \quad (\text{Conv})$$

Here we write $X \notin \Gamma$ iff there is no $X : A \in \Gamma$ for any A , and we denote by $[X := N]M$ the standard (capture-free) substitution of all free occurrences of X in M by N . In the last rule, \equiv_{β} is the usual notion of β -convertibility, which contains α -convertibility (this is trivially satisfied in this presentation). Observe that the side conditions ensure that we can only derive *simple judgements*, i.e. judgements with simple contexts. We say that T is a *type* in the context Γ iff $T \in \mathcal{S}$ or $\Gamma \vdash T : s$ for some $s \in \mathcal{S}$. Furthermore, M is said to be an *element* of type T in the context Γ iff $\Gamma \vdash M : T$, in which case we also say that M is *well-typed* in Γ .

As an example, we can instantiate PTSs by

$$\begin{aligned} \mathcal{S} &= \{\text{Prop}, \text{Type}\}, \\ \mathcal{A} &= \{(\text{Prop}, \text{Type})\}, \\ \mathcal{R} &= \{(\text{Prop}, \text{Prop}, \text{Prop}), \\ &\quad (\text{Prop}, \text{Type}, \text{Type}), \\ &\quad (\text{Type}, \text{Prop}, \text{Prop}), \\ &\quad (\text{Type}, \text{Type}, \text{Type})\} \end{aligned}$$

to obtain the calculus of constructions (cf. Section 2.5.1).

This presentation of PTSs is rather abstract for two reasons: firstly, we are working modulo α -conversion, i.e., we identify α -convertible terms, and secondly, we are concerned with an inductive definition of a *set* of derivable judgements, but *not* with an *algorithm* to verify derivability of a given judgement.

Mathematically the abstract presentation has an important benefit: It allows us to reason about PTSs metalogically, without assuming anything about the concrete realization of names. This leads to very general results [Bar92, vBJ93] and frees proofs from unnecessary technical details.

Closure under α -conversion is the property that derivability of $\Gamma \vdash M : A$ and $M \equiv_{\alpha} M'$ implies derivability of $\Gamma \vdash M' : A$. Of course, this property trivially holds for PTSs as presented above, since \equiv_{α} is the identity. To state a stronger property we extend α -conversion \equiv_{α} from terms to judgements such that $\Gamma \vdash M : A \equiv_{\alpha} \Gamma' \vdash M' : A'$ iff $\Gamma' \vdash M' : A'$ and $\Gamma \vdash M : A$ are equal up to consistent renaming of variables. Then we have the following

Lemma 6.2.1 (Strong Closure under α -Conversion for PTSs)

Let M, A, M', A' be PTS terms and Γ, Γ' be PTS contexts. If the PTS judgement

$\Gamma \vdash M : A$ is derivable in \mathbf{PTS}_S and $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$, then $\Gamma' \vdash M' : A'$ is derivable in \mathbf{PTS}_S .

Proof Sketch. By induction over derivations of $\Gamma \vdash M : A$. □

The previous Lemma is equivalent to the statement that the following rule is admissible in PTSs:

$$\frac{\Gamma \vdash M : A}{\Gamma' \vdash M' : A'} \quad \text{if } \Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A' \quad (\text{Rename})$$

6.2.1 PTSs in Membership Equational Logic

In the following specifications, given in Maude syntax, we use the algebraic semantics of MEL for representing PTSs exactly as given above; a more operational version suited for use as an implementation is discussed in Section 6.3.2.

First, notice that we plan to describe not a single type system but the *infinite family* of PTSs parameterized by PTS signatures which define sorts, axioms and rules. All such PTS signatures can be formalized as models of a single parameter theory that can be specified in Maude as follows:

```
fth PTS-SIG is
  sorts Sorts Axioms Axioms? Rules Rules? .
  subsort Axioms < Axioms? .
  subsort Rules < Rules? .
  op (_,_) : Sorts Sorts -> Axioms? .
  op (_,_,_) : Sorts Sorts Sorts -> Rules? .
endfth
```

As an example, the PTS signature of CC is given by the following functional module:

```
fmod CC-SIG is

  sorts Sorts Axioms Axioms? Rules Rules? .
  subsort Axioms < Axioms? .
  subsort Rules < Rules? .
  op (_,_) : Sorts Sorts -> Axioms? .
  op (_,_,_) : Sorts Sorts Sorts -> Rules? .

  op Prop : -> Sorts .
  op Type : -> Sorts .
```

```

mb (Prop,Type) : Axioms .
mb (Prop,Prop,Prop) : Rules .
mb (Prop,Type,Type) : Rules .
mb (Type,Prop,Prop) : Rules .
mb (Type,Type,Type) : Rules .

endfm

```

PTSs can then be specified as a functional module parameterized by the theory PTS-SIG. Since functional modules have an initial (in this case free) model semantics, this formalization of PTSs is in fact a parameterized inductive definition that captures in a precise model-theoretic way the inductive character of PTS rules.

```
fmod PTS[S :: PTS-SIG] is
```

First we define the sort `Trm` of terms as an algebraic data type. Notice that we distinguish between a sort of names `Qid`, that are used in places where a variable is *declared*, and a sort of variables `Var`, that are used to *refer* to an already declared variable.

```

sorts Var Trm .
subsort Qid < Var .
subsort Var < Trm .
subsort Sorts < Trm .
op __ : Trm Trm -> Trm .
op [_:_]_ : Qid Trm Trm -> Trm .
op {_:}_ : Qid Trm Trm -> Trm .

vars s s1 s2 s3 : Sorts .
vars X Y Z : Qid .
vars A B M N O P Q R T A' B' M' N' T' : Trm .

```

The usual deterministic version of capture-free substitution can be naturally defined in MEL as demonstrated in [MOM96]. An important point is that we do not want to restrict ourselves to a particular choice of fresh names, since this would make the specification overly concrete. This can be accomplished by leaving unspecified the deterministic function for choosing fresh variables such that the actual function varies with the choice of the model; for details we refer to [MOM96]. Here we only give the signature for set membership, free variables and the substitution function:

```

op _in_ : Qid QidSet -> Bool .
op FV : Trm -> QidSet .
op [_:=_]_ : Qid Trm Trm -> Trm .

```

We can use the substitution operator $[_:=_]_$ to semantically identify terms that are α -convertible (we refer to the induced equality as α -equality) by means of the following equations.

```

ceq [X : A] M = [Y : A] ([X := Y] M) if not(Y in FV(M)) .
ceq {X : A} M = {Y : A} ([X := Y] M) if not(Y in FV(M)) .

```

We next define the binary relation of β -convertibility, which is used in the *Conv* rule of PTSs. The following (conditional) memberships, together with the initiality condition, define β -conversion as the smallest congruence (w.r.t. the term constructors) containing β -reduction.

```

sorts Convertible Convertible? .
subsort Convertible < Convertible? .

op _<->_ : Trm Trm -> Convertible? .

mb M <-> M : Convertible .

cmb M <-> N : Convertible
  if N <-> M : Convertible .

cmb P <-> R : Convertible if
  P <-> Q : Convertible and Q <-> R : Convertible .

cmb (M N) <-> (M' N') : Convertible if
  M <-> M' : Convertible and N <-> N' : Convertible .

cmb ([X : A] M) <-> ([X : A'] M') : Convertible if
  A <-> A' : Convertible and M <-> M' : Convertible .

cmb ({X : A} B) <-> ({X : A'} B') : Convertible if
  A <-> A' : Convertible and B <-> B' : Convertible .

mb (([X : A] M) N) <-> ([X := N] M) : Convertible .

```

The judgements of PTSs are of the form $\Gamma \vdash M : A$. We next define the syntax of contexts and judgements. Also, we define the function `_in_` used in the side conditions of some PTS rules.

```

sorts Context Judgement .
op emptyContext : -> Context .
op _:_ : Qid Trm -> Context .
op _,_ : Context Context -> Context [assoc id : emptyContext] .

var G : Context .

op _|_:_ : Context Trm Trm -> Judgement .

op _in_ : Qid Context -> Bool .
eq X in emptyContext = false .
eq X in (G,(Y : A)) = (X in G) or (X == Y) .

```

We are now ready to define the inference rules. Semantically, the inference rules define an inductive subset of *derivable judgements*. The derivability predicate is usually implicit in informal reasoning, where $\Gamma \vdash M : A$ refers either to the judgement itself or to the fact that it is derivable.

```

sort Derivable .
subsort Derivable < Judgement .

cmb (emptyContext |- s1 : s2) : Derivable if (s1,s2) : Axioms .

cmb (G,(X : A) |- X : A) : Derivable if
  (G |- A : s) : Derivable /\ not(X in G) .

cmb (G,(X : B) |- M : A) : Derivable if
  (G |- M : A) : Derivable /\
  (G |- B : s) : Derivable /\ not(X in G) .

cmb (G |- {X : A} B : s3) : Derivable if
  (G |- A : s1) : Derivable /\
  (G,(X : A) |- B : s2) : Derivable /\ (s1,s2,s3) : Rules .

cmb (G |- [X : A] M : {X : A} B) : Derivable if
  (G |- A : s1) : Derivable /\
  (G,(X : A) |- M : B) : Derivable /\
  (G,(X : A) |- B : s2) : Derivable /\ (s1,s2,s3) : Rules .

cmb (G |- (M N) : [X := A] B) : Derivable if
  (G |- M : {X : A} B) : Derivable /\
  (G |- N : A) : Derivable .

```

```

cmb (G |- M : B) : Derivable if
  (G |- M : A) : Derivable /\
  (G |- B : s) : Derivable /\ A <-> B : Convertible .

endfm

```

In this formalization we have avoided any arbitrary encoding of syntax with binders that would require nontrivial justifications. Also, we have seen that the first-order framework is sufficiently powerful to represent PTSs without making any commitments. In particular, there was no need to change the syntax nor the rules of PTSs to obtain a faithful representation.

6.2.2 Taking Names Seriously

Although the abstract treatment of names in PTSs leads to a general metatheory that can be used as a high-level theoretical basis for quite different implementations of PTSs, there is a price to pay, in that an abstract view necessarily limits the expressivity of the theory. In the case of PTSs, properties involving names cannot be expressed. Indeed, we often need a more concrete representation with more specialized results to deal, for example, with the implementation of a formal system, or with tools that use the formal system in an essential way. Also in the context of reasoning about a formal system, a more concrete specification that is computationally meaningful can have considerable advantages for the partial automation of metatheoretic proofs in logics with computational sublanguages.

However, as soon as *we take names seriously, i.e., we give up the identification of α -convertible terms*, and interpret the inference rules literally, we encounter at least two problems first discussed in [Pol93] under the title “closure under α -conversion”.¹

The *first problem* is that the set of derivable judgements is not closed under α -conversion. For instance, adapting an example given for $\lambda \rightarrow$ in [Pol93], we cannot derive a judgement of the form

$$A : \text{Prop}, P : \{Z : A\}\text{Prop} \vdash [X : A][X : P X]A : \{X : A\}\{X : P X\}\text{Prop},$$

say in CC, although the α -equivalent version

$$A : \text{Prop}, P : \{Z : A\}\text{Prop} \vdash [X : A][Y : P X]A : \{X : A\}\{Y : P X\}\text{Prop},$$

where some bound variables are distinct can be derived.

¹The problem with closure under α -conversion also remains unsolved in [Mag94], where a system with dependent types is presented that does not enjoy this property.

A *second difficulty* pointed out in [Pol93] is that we want to derive

$$A : \text{Prop}, P : \{Z : A\}\text{Prop} \vdash [X : A][X : P X]X : \{X : A\}\{Y : P X\}(P X),$$

but we should *not* be able to derive

$$A : \text{Prop}, P : \{Z : A\}\text{Prop} \vdash [X : A][X : P X]X : \{X : A\}\{X : P X\}(P X).$$

However, we cannot derive the first judgement, since the name X in the conclusion of the `Lda` rule is the same on both sides of the colon.

To tackle the first problem, Pollack proposed a type system \vdash_{tt} , a variation of $\lambda \rightarrow$. It uses a more liberal notion of context that allows multiple declarations of the same name, the one most recently introduced being visible inside the judgement. Unfortunately, he did not pursue this direction further because of the second difficulty, which appears in the context of PTSs with dependent types but is not present in $\lambda \rightarrow$. Concerning \vdash_{tt} , he remarks “I don’t think we can do the same for PTS.”

The solution finally discussed in [Pol93] is the solution employed in the *constructive engine* [Hue89] used in proof assistants such as LEGO [Pol94] and COQ [BBC⁺99]. The idea is to use a hybrid naming scheme which employs distinct names for *global variables* declared in the context of a judgement, and a de Bruijn representation of terms with bound *local variables*. Clearly, PTSs based on such a hybrid naming scheme are a correct implementation of (abstract) PTSs as described above. More precisely, PTSs using the hybrid naming scheme can be seen as particular models of the MEL specification of PTSs in the sense that the corresponding model is isomorphic to the one given by the appropriately instantiated functional module PTS. Nevertheless, an approach which maintains a distinction between global and local variables appears not to be very uniform, complicating formal metatheoretic proofs and type checking. Of course, scaling up Pollack’s \vdash_{tt} to PTSs would be much more satisfying, and this is the direction we pursue in the following.

A closer look at the second difficulty shows that in an intermediate state of typechecking we would have a context

$$A : \text{Prop}, P : \{Z : A\}\text{Prop}, X : A, X : (P X)$$

and we would like to express that in this context the type of X is $(P X)$ where the latter X refers to the X declared by $X : A$ which is hidden by $X : (P X)$.

Obviously, this is an instance of *accidental hiding*, a problem which is avoided by the use of Berkling’s representation in the CINNI calculus. Indeed, employing indexed names the type of X in the context above just becomes $(P X_1)$. Therefore, our next goal is to obtain a version of PTSs based on CINNI which not only avoids this problem but also avoids *weird renaming* which takes place in PTS to keep variables in a context distinct.

6.2.3 Uniform Pure Type Systems

The application of CINNI to PTSs can be seen as Pollack's \vdash_{lt} scaled up to PTSs. In contrast to the hybrid approach to PTSs adopted in the constructive engine [Hue89] and in the PTS formalization given in [MP93] based on an idea from [Coq91], both distinguishing between global and local variables, we use indexed names *uniformly*. This suggests defining *uniform pure type systems (UPTSs)* by modifying PTSs in three steps:

First, PTS terms are generalized to UPTS terms in the way explained before, i.e., *UPTS terms* are defined by the first-order CINNI syntax:

$$X_m \mid (M N) \mid [X : A]M \mid \{X : A\}M \mid s$$

As a second step, we adapt the syntax-dependent part of the CINNI calculus to UPTS terms:

$$\begin{aligned} S s &= s \\ S (MN) &= (SM)(SN) \\ S ([X : A]M) &= [X : (S A)](\uparrow_X S M) \\ S (\{X : A\}M) &= \{X : (S A)\}(\uparrow_X S M) \end{aligned}$$

The third and final step is to define the derivable typing judgements. Since we do not want to identify α -convertible terms, this is a fundamental change in the formal system. However, a careful inspection of the typing rules *under the new reading* shows that only minor changes in the rules **Start** and **Weak** are needed. The new rules are:

$$\frac{\Gamma \vdash A : s}{\Gamma, X : A \vdash X_0 : \uparrow_X A} \quad (\text{Start})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, X : B \vdash \uparrow_X M : \uparrow_X A} \quad (\text{Weak})$$

It might appear that the UPTSs we have defined above are a specialization of PTSs, since we have committed ourselves to a particular representation of names. But this is not the full truth, because on the other hand we have described a generalization of PTSs where multiple declarations of the same name are admitted in a well-typed context. Notice that in both rules above we have dropped the side condition $X \notin \Gamma$, which means that we have completely eliminated the need for these side conditions in UPTSs. We would also like to point out, that, in particular, we have not touched the **Lda** rule: the only place where α -conversion comes into play is in the **Conv** rule, where \equiv_β subsumes α - and β -conversion, just as in the original PTSs.

Finally, we describe how these changes are reflected in the MEL specification, that is how UPTSs can be represented by modifying the previous specification.

First, instead of using names as variables we use indexed names. So we replace `subsort Qid < Var` by

```
op [_{ }] : Qid Nat -> Var .
```

Second, instead of conventional substitution `[_:=_]_`, we use CINNI for UPTS terms:

```
sort Subst .
```

```
op [_:=_] : Qid Trm -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Trm -> Trm .
```

```
var S : Subst .
vars n m : Nat .
```

```
eq ([X := M] (X{0})) = M .
eq ([X := M] (X{suc(m)})) = (X{m}) .
ceq ([X := M] (Y{n})) = (Y{n}) if X /= Y .
```

```
eq ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X /= Y .
```

```
eq ([lift X S] (X{0})) = (X{0}) .
eq ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X /= Y .
```

```
eq (S s) = s .
eq (S (M N)) = ((S M) (S N)) .
eq S ([X : A] M) = [X : (S A)] ([lift X S] M) .
eq S ({X : A} M) = {X : (S A)} ([lift X S] M) .
```

Third, conversion now explicitly contains α -conversion, something that was implicit in the equality of the previous specification:

```
mb [X : A] M <->
  [Y : A] ([X := Y{0}] [shift Y] M) : Convertible .

mb {X : A} M <->
  {Y : A} ([X := Y{0}] [shift Y] M) : Convertible .
```

Finally, the new versions of **Start** and **Weak** are:

$$\text{cmb } (\mathbb{G}, (X : A) \vdash X\{0\} : [\text{shift } X] A) : \text{Derivable if} \\ (\mathbb{G} \vdash A : s) : \text{Derivable .}$$

$$\text{cmb } (\mathbb{G}, (X : B) \vdash [\text{shift } X] M : [\text{shift } X] A) : \text{Derivable if} \\ (\mathbb{G} \vdash M : A) : \text{Derivable } \wedge \\ (\mathbb{G} \vdash B : s) : \text{Derivable .}$$

Again, we can see that the representational distance between the mathematical presentation of UPTSs and their MEL specification is practically zero. In particular, the equational nature of the CINNI substitution calculus is directly captured by the MEL specification.

UPTSs are more liberal than PTS, since a derivable judgement $\Gamma \vdash M : A$ may contain multiple declarations of the same name in Γ . However, the set of derivable judgements $\Gamma \vdash M : A$ of PTS can be recovered as the set of derivable UPTS judgements $\Gamma \vdash_1 M : A$ generated by adding the following rule:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash_1 M : A} \quad \text{if } \Gamma \text{ is simple} \quad (\text{Simple})$$

The representation of judgements $\Gamma \vdash_1 M : A$ together with this rule in MEL is straightforward, and we omit it here and in all the following formalizations for the sake of brevity.

To state the following results we proceed as for PTSs: We extend α -conversion \equiv_α from terms to judgements so that $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$ iff $\Gamma' \vdash M' : A'$ and $\Gamma \vdash M : A$ are equal up to consistent renaming of declared *and* bound variables. Then we have the following

Lemma 6.2.2 (Strong Closure under α -Conversion for UPTSs)

Let M, A, M', A' be UPTS terms and Γ, Γ' be UPTS contexts. If the UPTS judgement $\Gamma \vdash M : A$ is derivable in \mathbf{UPTS}_S and $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$, then $\Gamma' \vdash M' : A'$ is derivable in \mathbf{UPTS}_S .

Proof Sketch. By induction over derivations of $\Gamma \vdash M : A$. □

It is noteworthy that a weak form of this lemma using α -conversion on terms instead of judgements, i.e. the special case where $\Gamma = \Gamma'$, cannot be proved directly by induction. The induction would fail for the rules **Pi** and **Lda**, since a declared variable X becomes a local variable.

As for PTSs the previous lemma is equivalent to the admissibility of the following rule in UPTSs:

$$\frac{\Gamma \vdash M : A}{\Gamma' \vdash M' : A'} \quad \text{if } \Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A' \quad (\text{Rename})$$

Using the terminology introduced in Section 2.3 for entailment systems, each of the following two propositions establishes a sound and complete total correspondence of the form $\mathbf{PTS}_S \curvearrowright \mathbf{UPTS}_S$, where S is an arbitrary PTS signature.

Proposition 6.2.3 (Soundness and Completeness of UPTSs I)

For all PTS terms M, A and PTS contexts Γ , if the PTS judgement $\Gamma \vdash_1 M : A$ is derivable in \mathbf{UPTS}_S then $\Gamma \vdash M : A$ is derivable in \mathbf{PTS}_S and vice versa.²

Proof Sketch.

First observe that each PTS rule is a UPTS rule if we restrict ourselves to simple judgements. In particular, the side conditions $X \in \Gamma$ in the PTS rules **Start** and **Weak** imply that the shift substitution in the corresponding UPTS rules can be eliminated.

(\Rightarrow) Given a UPTS derivation of a simple judgement $\Gamma \vdash M : A$, each occurrence of a UPTS inference rule can be replaced as follows: First the original premises are converted into suitable simple PTS form by virtue of **Rename**. Then the corresponding inference rule for PTSs is applied (which is also a UPTS rule according to the observation above). Finally, the conclusion in simple PTS form is converted back to the original conclusion in UPTS form, again using **Rename**. After transforming the entire derivation in this way all intermediate UPTS judgements which are not PTS judgements, i.e. the original premises and original conclusions, can be removed, and the result is still a UPTS derivation. Also, the resulting derivation corresponds to a derivation in PTSs extended by the admissible rule **Rename**.

(\Leftarrow) According to the observation above, each application of a PTS rule can be seen as an application of the corresponding UPTS rule. Furthermore, each implicit α -conversion step that is possible in PTSs can be simulated by **Rename**, which is an admissible rule in UPTSs. □

In other words, UPTSs are conservative over PTSs. A slightly weaker but more comprehensive correspondence of the form $\mathbf{PTS}_S \curvearrowright \mathbf{UPTS}_S$ can be given modulo renaming of variables:

Proposition 6.2.4 (Soundness and Completeness of UPTSs II)

For all UPTS terms M, A , PTS terms M', A' , UPTS contexts Γ and simple PTS contexts Γ' with $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$, if the UPTS judgement $\Gamma \vdash M : A$ is derivable in \mathbf{UPTS}_S then $\Gamma' \vdash M' : A'$ is derivable in \mathbf{PTS}_S and vice versa.

Proof Sketch.

(\Rightarrow) Let $\Gamma \vdash M : A$ be derivable in \mathbf{UPTS}_S and let $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$. By Proposition 6.2.2 (strong α -closure) $\Gamma' \vdash M' : A'$ and therefore $\Gamma' \vdash_1 M' : A'$.

²Here we make use of the convention, introduced in Section 5.1, that ordinary terms (here PTS terms) can be seen as CINNI terms (here UPTS terms).

A' are derivable in \mathbf{UPTS}_S . So by Proposition 6.2.3 $\Gamma' \vdash M' : A'$ is derivable in \mathbf{PTS}_S .

(\Leftarrow) Let $\Gamma' \vdash M' : A'$ be derivable in \mathbf{PTS}_S and let $\Gamma' \vdash M' : A' \equiv_\alpha \Gamma \vdash M : A$. By Proposition 6.2.3, $\Gamma' \vdash M' : A'$ is derivable in \mathbf{UPTS}_S , and by Proposition 6.2.2 (strong α -closure) $\Gamma \vdash M : A$ is derivable in \mathbf{UPTS}_S too. \square

The last proposition implies that, concerning judgements of the form $\Gamma \vdash M : A$, PTSs and UPTSs are equivalent modulo α -conversion. Hence all (metatheoretic) results about PTSs [GN91] apply to UPTSs after appropriate renaming.

Another consequence of the last proposition is that the new form of judgement $\Gamma \vdash_1 M : A$ is not necessary to ensure soundness, and could therefore be dropped. Sometimes, however, focussing on judgements of the form $\Gamma \vdash_1 M : A$ instead of the more general form $\Gamma \vdash M : A$ is more convenient, e.g. to formulate the weakening/thinning lemma [GN91, vBJMP93], since simple contexts can be treated as sets of declarations.

6.2.4 A Conservative Optimization

The presentations of PTSs and UPTSs given above maintain a good economy in the number of rules and are therefore well-suited for metatheoretic (inductive) reasoning. The judgement $\Gamma \vdash M : A$ implicitly subsumes another judgement $\Gamma \Vdash$, stating that Γ is a well-typed context. Since in practice checking contexts is as important as checking types, we switch to a conservative extension of UPTSs (similar to an optimization for PTSs mentioned in [vBJMP93]) that is not biased towards any of the two forms of judgement. From a practical point of view, the addition of a separate judgement for well-typed contexts can be seen as an optimization which avoids rechecking contexts in each subderivation. We will refer to this optimized version as *optimized UPTSs (OUPTSs)* and the entailment system will be denoted by \mathbf{OUPTS} . The only modifications we need are described below. In addition to the *main typing judgement*, which is written now as $\Gamma \Vdash M : A$ (stating that M is an element of the type T in Γ), we use *context typing judgements* of the form $\Gamma \Vdash$ meaning that Γ is a well-typed context, and *relative typing judgements* of the form $\Gamma \vdash M : A$ meaning that M is an element of type A if Γ is well-typed. Furthermore, we add the following rules:

$$\frac{}{\boxed{\Gamma} \Vdash} \quad (\text{Ctx1})$$

$$\frac{\Gamma \Vdash \quad \Gamma \vdash A : s}{\Gamma, X : A \Vdash} \quad (\text{Ctx2})$$

$$\frac{}{\Gamma \vdash X_m : \text{lookup}(\Gamma, X_m)} \quad \text{if } \text{lookup}(\Gamma, X_m) \text{ is defined} \quad (\text{Lookup})$$

$$\frac{\Gamma \Vdash \Gamma \vdash M : A}{\Gamma \Vdash M : A} \quad (\text{Main})$$

where $lookup(\Gamma, X_m)$ is a partial function defined by:

$$\begin{aligned} lookup((\Gamma, X : A), X_0) &= \uparrow_X A \\ lookup((\Gamma, X : A), X_{m+1}) &= \uparrow_X lookup(\Gamma, X_m) \\ lookup((\Gamma, X : A), Y_m) &= \uparrow_X lookup(\Gamma, Y_m) \text{ if } X \neq Y \end{aligned}$$

Then we replace **Ax** and **Simple** by:

$$\frac{}{\Gamma \vdash s_1 : s_2} \quad (s_1, s_2) \in \mathcal{A} \quad (\text{Ax})$$

$$\frac{\Gamma \Vdash M : A}{\Gamma \Vdash_1 M : A} \quad \text{if no variable is declared in } \Gamma \text{ more than once.} \quad (\text{Simple})$$

respectively, and we remove the rules **Start** and **Weak**, since they are admissible rules in the new system. The system we have just obtained is similar to the system $\vdash_{vtyp}, \vdash_{vctx}$ presented in [vBJMP93], but here we are concerned with UPTSs instead of PTSs. Another minor difference is that we make use of an explicit lookup function. As before, we do not need any freshness side conditions thanks to CINNI.

Again, the representation in MEL is quite direct. It nicely illustrates the mixed specification style using equations and memberships, and also the representation of partial functions such as $lookup$.

```

sort Trm? .
subsort Trm < Trm? .

op lookup : Context Var -> Trm? .
eq lookup(G, (X : A), X{0}) = [shift X] A .
eq lookup(G, (X : A), X{suc(m)}) = [shift X] lookup(G, X{m}) .
ceq lookup(G, (X : A), Y{m}) = lookup(G, Y{m}) if (X /= Y) .

op _||- : Context -> Judgement .
op _|-_:_ : Context Trm Trm -> Judgement .
op _||-_:_ : Context Trm Trm -> Judgement .

mb (emptyContext ||-) : Derivable .

cmb (G, (X : A) ||-) : Derivable if
  (G ||-) : Derivable /\ (G |- A : s) : Derivable .

```

```

cmb (G |- X{m} : lookup(G,X{m})) : Derivable if
  lookup(G,X{m}) : Trm .

cmb (G ||- M : A) : Derivable if
  (G ||-) : Derivable /\ (G |- M : A) : Derivable .

cmb (G |- s1 : s2) : Derivable if (s1,s2) : Axioms .

```

OUPTSs are equivalent to UPTSs, i.e., there is a sound and complete total correspondence of the kind $\mathbf{UPTS}_S \curvearrowright \mathbf{OUPTS}_S$ for arbitrary PTS signatures S , in the following sense:

Proposition 6.2.5 (Soundness and Completeness of OUPTSs)

Let M, A be UPTS terms, and let Γ be a UPTS context. If the judgement $\Gamma \Vdash M : A$ ($\Gamma \Vdash_1 M : A$) is derivable in \mathbf{OUPTS}_S , then $\Gamma \vdash M : A$ ($\Gamma \vdash_1 M : A$) is derivable in \mathbf{UPTS}_S and vice versa.

Proof Sketch. It is easy to adopt the proof of the similar lemma 23 in [vBJMP93] to our setting. The main change is that we are using UPTSs instead of PTSs here. A minor point is that we are using an explicit *lookup* function. \square

6.3 The Meta-Operational View of PTSs

PTSs can not only be equipped with a logical semantics, e.g. via the proposition-as-types interpretation, but, more fundamentally, PTSs are usually equipped with an operational semantics, defined by an internal notion of functional computation, such as β -reduction. The operational view of PTSs is concerned with their internal notion of computation, but here we are interested in the *meta-operational view*, which deals with the question of how to embed PTSs in a formal system with an operational semantics, so that typical computational tasks like type checking and type inference become possible by exploiting the operational semantics of the metalanguage. In the following we employ for this purpose the efficiently executable sublanguage of rewriting logic that is supported by the Maude engine.

First, we introduce several well-known classes of PTS signatures, giving rise to corresponding PTSs that are practically interesting and enjoy particularly good properties.

Definition 6.3.1 A PTS signature S is *decidable* iff: (1) \mathcal{S} is denumerable, (2) \mathcal{A} and \mathcal{R} are decidable, and (3) for all $s_1, s_2 \in \mathcal{S}$ the predicates $\exists s'_2 : (s_1, s'_2) \in \mathcal{A}$ and $\exists s'_3 : (s_1, s_2, s'_3) \in \mathcal{R}$ are decidable.

Decidability is a reasonable requirement to ensure that type inference and type checking do not become undecidable because of a too complex PTS signature.

Definition 6.3.2 A PTS signature S is *functional* iff (1) $(s_1, s_2) \in \mathcal{A}$ and $(s_1, s'_2) \in \mathcal{A}$ implies $s_2 = s'_2$, and (2) $(s_1, s_2, s_3) \in \mathcal{R}$ and $(s_1, s_2, s'_3) \in \mathcal{R}$ implies $s_3 = s'_3$.

In functional PTS signatures, the relations \mathcal{A} and \mathcal{R} can be viewed as partial functions $\mathcal{A} : \mathcal{S} \rightsquigarrow \mathcal{S}$ and $\mathcal{R} : \mathcal{S} \times \mathcal{S} \rightsquigarrow \mathcal{S}$. Functionality ensures that every term has a unique type modulo \equiv_β [GN91]. The class of functional PTSs³ includes, for example, all systems of the λ -cube.

Definition 6.3.3 A PTS signature S is *full* iff for all $s_1, s_2 \in \mathcal{S}$ there is an s_3 such that $(s_1, s_2, s_3) \in \mathcal{R}$. A PTS signature S is *semi-full* iff $(s_1, s_2, s_3) \in \mathcal{R}$ implies that for each s'_2 there is an s'_3 such that $(s_1, s'_2, s'_3) \in \mathcal{R}$.

Full PTSs allow us to form dependent types $\{X : A\}B$ very liberally, by avoiding those restrictions on the sorts of A and B that are imposed by the side condition $(s_1, s_2, s_3) \in \mathcal{R}$ of the Pi rule. As an example, CC is a full PTS.

Definition 6.3.4 Given a PTS signature S , a *top sort* is a sort s such that there is no sort s' with $(s, s') \in \mathcal{A}$. The set of top sorts is denoted by \mathcal{S}^t .

To avoid inessential technicalities in our presentation, we will later focus on PTS signatures without top sorts, which introduce some kind of nonuniformity in the set of sorts. Just as \mathcal{R} can be seen as a function $\mathcal{R} : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ in full PTS signatures, \mathcal{A} can be viewed as a function $\mathcal{A} : \mathcal{S} \rightarrow \mathcal{S}$ in functional PTS signatures without top sorts.

Semi-full PTSs have the nice property that we can get rid of the third premise in **Lda** by replacing it with the following rule:

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash M : B}{\Gamma \vdash [X : A]M : \{X : A\}B} \quad (s_1, s_2, s_3) \in \mathcal{R} \text{ and } B \notin \mathcal{S}^t \quad (\text{Lda}')$$

The premises together with the side conditions in **Lda'** imply that $\{X : A\}B$ is a type (cf. rule Pi). Indeed, as explained in [vBJMP93] in the context of PTSs, replacing **Lda** by **Lda'** does not change the set of derivable judgements in semi-full UPTSs.

For full UPTSs without top sorts we can completely eliminate the side conditions in the rule **Lda'**, and we obtain **Lda''** without changing the set of derivable judgements:

$$\frac{\Gamma \vdash A : s \quad \Gamma, X : A \vdash M : B}{\Gamma \vdash [X : A]M : \{X : A\}B} \quad (\text{Lda}'')$$

³The attributes for PTS signatures are naturally lifted to the corresponding PTSs.

Our example PTS signature of CC at the beginning of Section 6.2 has `Type` as a top sort. However, it is straightforward to extend CC by an infinite universe hierarchy yielding a PTS without top sorts.

Together with the introduction of UPTSs in the previous section, we have now (following the corresponding arguments for PTSs in [vBJMP93]) three families of inference systems which only differ in the choice of the rule `Lda`. For a full PTS signature S without top sorts all of them define the same unary entailment system, which is denoted by `UPTSS`.

In the remainder of this chapter we will present a standard type checking algorithm for a class of UPTSs using rewriting logic as a formal specification language. In spite of some unsolved theoretical questions such as the expansion postponement problem, efficient algorithms for the important classes of functional PTSs and semi-full PTSs (satisfying appropriate decidability and normalization properties) have been presented in [vBJMP93]. In order to avoid excessive technical details and to make clear the general way we use rewriting logic to represent type checking algorithms, we restrict ourselves in the following to UPTSs that are decidable, normalizing (w.r.t. β -reduction), functional, full, and without top sorts. The class of UPTSs that are decidable, normalizing, functional and semi-full can be treated along the same lines (using the rule `Lda'` instead of `Lda`”).

The use of UPTSs instead of PTSs is motivated by our desire to obtain a *formal executable representation* that takes names seriously and makes type checking simpler and more uniform. The approach is different from the constructive engine [Hue89] and its presentation in [Pol93] that employs named global variables and a de Bruijn representation for local variables. It is also different from [Coq91], [vBJMP93], and the formalization [MP93], that distinguish between two unrelated sets of global names and local names.

6.3.1 Uniform PTSs in Membership Equational Logic

The standard way to implement type checking, which goes back to [ML72] and [Hue89], is to cast the inference rules into an equivalent syntax-directed form [vBJMP93], and to define a type-inference function on the basis of this new system. Formally and technically this could be done in the executable sublanguage of MEL or in any other functional programming language, but the use of MEL is attractive, since it allows us to formulate the logical and operational versions of PTSs in a single uniform language with a simple semantics, which in particular does not presuppose higher-order constructs, but is used to explain them in more elementary terms. Also, data structures and functions of the specification can be directly used in the implementation.

In our setting there is another reason why MEL is more natural than the use of a (higher-order) functional programming language: the equational specification of

the calculus of substitutions presented above is naturally equipped with an operational semantics just by viewing the equations as rewrite rules. By contrast, in a functional programming language that is not based on equational rewriting, the substitution calculus has to be *encoded*, which essentially means that a (specialized) rewrite engine for this calculus has to be implemented in the functional language itself and, what is even more cumbersome, this engine has to be explicitly invoked when needed. In this sense, a specification/programming style based on rewriting is more abstract and closer to mathematical practice for applications of this kind than a (higher-order) functional programming approach.

Using the specification of the above substitution calculus, a purely equational executable specification of a type checker for UPTSs with decidable type checking can be written in MEL using standard equational/functional programming techniques. The core of this specification consists of a type-inference function

```
op type : Context Trm -> Trm? .
```

that computes a type for each term which is well-typed in the given context. The function can be defined in a way similar to the one given in [Sev96], but using CINNI, instead of abstracting from the treatment of names.

Thanks to CINNI, freshness conditions are avoided. Therefore, an implementation based on this specification appears to be more elegant than that of the constructive engine with its hybrid treatment of names. As an additional advantage, multiple declarations of the same name are naturally admitted in contexts if we use judgements of the form $\Gamma \Vdash M : A$. However, it is also easy to disallow these more general contexts if desired by implementing simple judgements $\Gamma \Vdash_1 M : A$.

Instead of discussing this purely equational approach in more detail, we present an alternative approach in the following section that exploits features of rewriting logic that are beyond equational and functional languages. Our experience shows that this alternative approach scales up well to more complex type theories, e.g. extensions of UPTSs such as OCC (see Chapter 8), in a more satisfactory way than the purely functional and equational approaches to type checking.

6.3.2 Uniform PTSs in Rewriting Logic

As shown by an extensive collection of examples in [MOM94, MOM96], rewriting logic can be used as a logical framework that can naturally represent inference systems of different kinds in a logically and operationally satisfying way. In the present section we view a type checker as a particular inference system. In contrast to a (higher-order) functional programming approach that would require us to *encode* the inference system in terms of a type checking function, the rewriting

logic approach offers the advantage that inference rules can be expressed directly, namely, as rewrite rules. We will in fact make use of a type inference system expressed as a collection of rewrite rules that transform a conjunction of judgements into a simplified form, in the style of *constraint solving systems*. This yields a rewrite system that is efficiently executable, while still maintaining a close correspondence to the logical specification of UPTSs.

The rewriting logic specification represents *rewriting-based OUPTSs (ROUPTSs)* and is able to perform type checking, i.e. to decide derivability of judgements of the form $\Gamma \Vdash, \Gamma \vdash M : A$, and $\Gamma \Vdash M : A$, for the class of decidable, normalizing, functional, full and PTS signatures without top sorts discussed before. As in PTSs, type checking is reduced to type inference, that is, to solving incomplete queries of the form $\Gamma \vdash M \rightarrow : ?$.

Instead of giving an informal account we directly discuss the formal specification in rewriting logic.

First, we exploit our assumption that the PTS signature is decidable, functional, full and without top sorts, which means that the relations \mathcal{A} and \mathcal{R} can be specified by equationally-defined functions **Axioms** and **Rules**:

```
fth FPTs-SIG is
  sort Sorts .
  op Axioms : Sorts -> Sorts .
  op Rules : Sorts Sorts -> Sorts .
endfth
```

As usual for syntax-directed approaches following the ideas of [ML72] and [Hue89] we “invert” the inference rules in order to obtain a goal-directed algorithm from the inductive definition. In contrast to a purely equational and functional approach, the rewriting logic specification we aim at has a rewrite transition system as a model, and can therefore be seen as an operational generalization of the equational and functional paradigms. In contrast to [vBJMP93] and [Pol93], the type-checking algorithm itself receives a direct formal status as a transition system, which is a good basis for reasoning formally about operational properties and especially about its correctness.

The inductive definition of UPTSs can be seen as a static description of a set of judgements that we would like to equip with a dynamic structure. More precisely, a *(static) logical implication*

$$A_1 \wedge \dots \wedge A_n \Rightarrow B$$

can be seen as an *inference rule* or *(dynamic) state transition* refining a goal B into subgoals A_1, \dots, A_n , and can be directly represented as a rewrite rule

$$B \Rightarrow A_1 \wedge \dots \wedge A_n$$

in rewriting logic. Each state consists of a finite set of subgoals that remain to be solved.

The static description can be seen as inducing the following invariant that our dynamic system should always satisfy: for each state, the empty set of goals is reachable iff the logical interpretation of the state is true.

Although the inference rules of a formal system typically take the form of Horn clauses that can be operationally refined to rewrite rules, there may be functional and equational parts (e.g. auxiliary functions or substitution calculi) that are more naturally expressed in the MEL fragment. It is this mix of different paradigms that allows us to express the type-checking algorithm in a way that is very close to the logical specification.

We discuss below the rewriting logic specification of the UPTS type checker in some detail. Instead of an equational theory introduced by the `fmod` keyword, the specification takes the form of a rewrite theory, introduced by the `mod` keyword, that has a transition system as its initial semantics:

```
mod PTS[S :: FPTSSIG] is
```

We reuse most components of the functional module defined before, but in addition to the *typing judgement*

```
op _|-_:_ : Context Trm Trm -> Judgement .
```

we add the following *auxiliary judgements*:

```
op _Sort : Trm -> Judgement .
op (_,_,_)Rule : Trm Trm Trm -> Judgement .
op _=_ : Trm Trm -> Judgement .
op _<->_ : Trm Trm -> Judgement .
op _|-_->:_ : Context Trm Trm -> Judgement .
op _|-(->:_)(->:_)>:_ : Context Trm Trm Trm Trm ->
                          Judgement .
```

Recall that, in our setting of PTS signatures without top sorts, T is a type in Γ iff $\Gamma \vdash T : s$. Presupposing that Γ is a well-typed context and A, B are types in Γ , the meaning of the auxiliary judgements is the following: The judgement A `Sort` means that there is an $s \in \mathcal{S}$ such that $A \equiv_\beta s$. The judgement (A, B, s) `Rule` means that there are $s_1, s_2 \in \mathcal{S}$ such that $A \equiv_\beta s_1$, $B \equiv_\beta s_2$ and $(s_1, s_2, s) \in \mathcal{R}$. The judgement $A <-> B$ just means that $A \equiv_\beta B$. The judgement $\Gamma \vdash M -> : T$ means that M has an *inferred type* T in Γ . Regarding this refinement of typing judgements we only assume that $\Gamma \vdash M -> : T$ implies $\Gamma \vdash M : T$, and conversely that $\Gamma \vdash M : T$ implies that $T \equiv_\beta T'$ and $\Gamma \vdash M -> : T'$ for some T' .

Furthermore, the judgement $\Gamma \vdash ((M \rightarrow: S)(N \rightarrow: T)) \rightarrow: U$ abbreviates $\Gamma \vdash M \rightarrow: S$, $\Gamma \vdash N \rightarrow: T$, and $\Gamma \vdash (M N) \rightarrow: U$. Finally, the judgement $M = N$ just means that M and N are equal terms.

In order to express intermediate goals or queries, like $\Gamma \vdash M \rightarrow: ?$, that are present in the operational refinement but not in the abstract presentation, we extend terms by explicit metavariables:

```
sort MetaVar .
subsort MetaVar < Trm .
op ? : Qid -> MetaVar .
var MV : MetaVar .
```

In ROUPTSs we use the weak head normal form, calculated by the following function `whnf`, to check if two terms are convertible, and in particular if a term is convertible to the form s or $\{X : A\}M$. We also use sorts `WhNf` and `WhReducible` containing terms in weak head normal form and weak head reducible terms, respectively.

```
sort WhNf WhReducible .
subsort WhNf < Trm .

subsort Sorts < WhNf .
subsort Var < WhNf .
mb ([X : A] M) : WhNf .
mb ({X : A} B) : WhNf .
mb (s N) : WhNf .
mb (X{m} N) : WhNf .
cmb ((P Q) N) : WhNf if (P Q) : WhNf .
mb (({X : A} M) N) : WhNf .

subsort WhReducible < Trm .

mb (([X : A] M) N) : WhReducible .
cmb (M N) : WhReducible if M : WhReducible .

op whnf : Trm -> Trm? .

ceq whnf(M) = M if M : WhNf .
eq whnf([X : A] M) N = whnf([X := N] M) .
ceq whnf(M N) = whnf(whnf(M) N) if M : WhReducible .
```

A configuration is a conjunctive set of judgements that have to be solved or verified by the type checker. We represent a set of judgements as a list. This

allows us to solve goals in a well-defined order, a fact that we exploit later in this section.

```

sort JudgementList .

op emptyJudgementList : -> JudgementList .
subsort Judgement < JudgementList .
op __ : JudgementList JudgementList -> JudgementList
      [assoc id: emptyJudgementList] .

var JS : JudgementList .

sort Configuration .

op {{_}} : JudgementList -> Configuration .

```

Replacement of metavariables by terms (that is, textual replacement) has the obvious definition, not spelled out here, except for its syntax:

```

op <_:=_>_ : MetaVar Trm Trm -> Trm .
op <_:=_>_ : MetaVar Trm Subst -> Subst .
op <_:=_>_ : MetaVar Trm Context -> Context .
op <_:=_>_ : MetaVar Trm Judgement -> Judgement .
op <_:=_>_ : MetaVar Trm JudgementList -> JudgementList .

```

It is used only in the following *equality elimination rule*, that instantiates a metavariable throughout the entire configuration if it is uniquely determined by an equality:

```

r1 [Subst] : {{ (MV = A) JS }} => {{ < MV := A > JS }} .

```

A rule like this is typical of a constraint-based programming approach, and indeed the configuration can be seen as a set of constraints that should be simplified using the subsequent rules [MOM96].

In addition to simplification of constraints by rewrite rules, simplification by equational rewriting also plays a major role in our approach. As an example, the judgement of convertibility between normalizing terms can be checked using *whnf* as follows. In order to avoid redundant reductions we reduce the general problem to a check of convertibility between weak head normal forms (which are treated by the first five rules below). In the case of binders we perform renaming to equalize names.

```

rl [Conv1] : {{ (s <-> s) JS }} => {{ JS }} .

rl [Conv2] : {{ (X{m} <-> X{m}) JS }} => {{ JS }} .

crl [Conv3] : {{ ((M N) <-> (M' N')) JS }} =>
  {{ (M <-> M') (N <-> N') JS }}
  if (M N) : WhNf /\ (M' N') : WhNf .

rl [Conv4] : {{ ({X : A} T <-> {Y : A'} T') JS }} =>
  {{ (A <-> A')
    ([X := Y{0}] [shift Y] T <-> T') JS }} .

rl [Conv5] : {{ ([X : A] M <-> [Y : A'] M') JS }} =>
  {{ (A <-> A')
    ([X := Y{0}] [shift Y] M <-> M') JS }} .

crl [Conv6] : {{ (M <-> N) JS }} =>
  {{ (whnf(M) <-> N) JS }}
  if M : WhReducible .

crl [Conv7] : {{ (M <-> N) JS }} =>
  {{ (M <-> whnf(N)) JS }}
  if N : WhReducible .

```

We use two auxiliary judgements to formalize side conditions:

```

rl [Sort] : {{ (s Sort) JS }} => {{ JS }} .

rl [Rule] : {{ ((s1,s2,MV) Rule) JS }} =>
  {{ (MV = Rules(s1,s2)) JS }} .

```

Each inference rule of OUPSTs gives rise to a rewrite rule obtained by reversing the direction of inference:

```

rl [Ax] : {{ (G |- s ->: MV) JS }} =>
  {{ (MV = Axioms(s)) JS }} .

crl [Lookup] : {{ (G |- X{m} ->: MV) JS }} =>
  {{ (MV = lookup(G,X{m})) JS }}

```

```

if lookup(G,X{m}) .

r1 [Pi] :    {{ (G |- {X : A} B ->: MV) JS }} =>
             {{ (G |- A ->: ?(NEW1)) (? (NEW1) Sort)
                (G,(X : A) |- B ->: ?(NEW2))
                ((?(NEW1), ?(NEW2), MV) Rule) JS }} .

r1 [Lda] :   {{ (G |- [X : A] M ->: MV) JS }} =>
             {{ (G |- A ->: ?(NEW1)) (? (NEW1) Sort)
                (G,(X : A) |- M ->: ?(NEW2))
                (MV = {X : A} ?(NEW2)) JS }} .

r1 [App1] :  {{ (G |- (M N) ->: MV) JS }} =>
             {{ (G |- M ->: ?(NEW1)) (G |- N ->: ?(NEW2))
                (G |- (M ->: ?(NEW1))(N ->: ?(NEW2)) ->: MV)
                JS }} .

r1 [App2] :  {{ (G |- (M ->: {X : A} B)(N ->: A') ->: MV)
                JS }} =>
             {{ (A <-> A') (MV = [X := N] B) JS }} .

```

The terms $?(NEW1)$ and $?(NEW2)$ above denote fresh metavariables. Hence rewriting has to be controlled by a simple *strategy*, that *constraints* the possible rewrites by instantiating the variables $NEW1$ and $NEW2$ only with fresh names each time a rule is applied. Notice that, in contrast to ordinary variables, where names are taken seriously, we abstract from (i.e. we do not care about) metavariable names, since they do not have a formal status inside UPTSs, but belong instead to the metalevel which is partially made explicit in the operational refinement.⁴

According to the explanations given before, the new judgements have certain closure properties w.r.t. \equiv_β . The following simplification rules allow us to work with (partially) normalized judgements in the inference rules:

```

cr1 [Norm1] : {{ (T Sort) JS }} =>
              {{ (whnf(T) Sort) JS }}
              if T : WhReducible .

cr1 [Norm2] : {{ ((A,B,T) Rule) JS }} =>
              {{ ((whnf(A),B,T) Rule) JS }}
              if A : WhReducible .

```

⁴By a further refinement of the present specification we can obtain a system which takes even metavariables seriously, but this is not necessary for the purposes of the present chapter.

```

crl [Norm3] :  {{ ((A,B,T) Rule) JS }} =>
               {{ ((A,whnf(B),T) Rule) JS }}
               if B : WhReducible .

crl [Norm4] :  {{ (G |- (M ->: A)(N ->: B) ->: T) JS }} =>
               {{ (G |- (M ->: whnf(A))(N ->: B) ->: T) JS }}
               if A : WhReducible .

```

This completes the definition of the *type-inference* system for judgements of the form $\Gamma \vdash M \rightarrow : A$. *Type checking* is reduced to type inference in the standard way, that is, $\Gamma \vdash M : A$ is verified by first checking if A is a type in Γ , and if this is the case we then check if A and the inferred type of M are convertible. Exploiting the fact that in PTSs without top sorts each type is contained in some sort, this can be specified by the rule

```

rl  [Aux] :    {{ (G |- M : A) JS }} =>
               {{ (G |- A ->: ?(NEW1)) (? (NEW1) Sort)
                  (G |- M ->: ?(NEW2)) (? (NEW2) <-> A) JS }} .

```

This rule can be slightly optimized by using an adaption of Lemma 3 from [Pol95], which allows us to omit the goal $(? (NEW1) \text{Sort})$ on the right hand side, since it is implied by the remaining goals.

Finally, we add rules to check the context typing judgement and the main typing judgement:

```

rl  [Ctxt1] :  {{ (emptyContext ||-) JS }} => {{ JS }} .

rl  [Ctxt2] :  {{ (G,(X : A) ||-) JS }} =>
               {{ (G ||-) (G |- A ->: ?(NEW))
                  (? (NEW) Sort) JS }} .

rl  [Main] :   {{ (G ||- M : A) JS }} =>
               {{ (G ||-) (G |- M : A) JS }} .

endm

```

Again we have omitted the straightforward rule corresponding to **Simple**, which allows us to check derivability of typing judgements $\Gamma \Vdash_1 M : A$ that disallow multiple occurrences of the same variable in Γ .

To verify a judgement J we start with an initial configuration $\{\{J\}\}$. Either this configuration can be reduced to $\{\{\text{emptyJudgementList}\}\}$, meaning that the judgement has been proved, or the final configuration contains unsolved goals giving an informative indication of an error.

Notice that we have not only used inductive definitions to specify PTSs and UPTSs logically, but that, in addition, the operational version of UPTSs given by the rewrite rules above is an inductive definition of a labeled transition system which gives us a more refined view of the type-checking process.

The most important property of a type checker is soundness, i.e., each judgement that has been verified should be derivable in the type system. In fact the formal system has been defined in such a way that the soundness of each of the rewrite rules above relative to OUPPTSs can be verified by straightforward inspection of the rules using the meaning of all auxiliary judgements given earlier.

More precisely, let S range over decidable, normalizing, functional, full PTS signatures without top sorts. We denote by \mathbf{ROUPTS}_S the entailment system in which sentences are rewrites of the form $\{\{JS\}\} \rightarrow \{\{JS'\}\}$ and such a rewrite is derivable iff it is derivable in the rewrite theory that has been presented above. Then the next proposition gives a sound and complete total correspondence $\mathbf{OUPTS}_S \curvearrowright \mathbf{ROUPTS}_S$.

Proposition 6.3.5 (Soundness and Completeness of ROUPTSs)

Let M, A be UPTS terms, let Γ be a UPTS context, and let J be one of the judgements $\Gamma \Vdash$, $\Gamma \Vdash M : A$, or $\Gamma \Vdash_1 M : A$. If the rewrite $\{\{J\}\} \rightarrow \{\{\text{emptyJudgementList}\}\}$ is derivable in \mathbf{ROUPTS}_S , then J is derivable in \mathbf{OUPTS}_S and vice versa.

Proof Sketch. The soundness part follows from the simple observation that for each ROUPTS rewrite rule the right hand side together with its possible condition implies the left hand side under the intended logical interpretation given earlier. The completeness part can be obtained by adapting the inductive proof of Lemma 29 in [vBJMP93]: Instead of the conventional notion of terms and substitution we have to use CINNI syntax with explicit substitutions, and instead of of PTSs we have to use OUPPTSs. \square

As explained in Chapter 2, executability in the following proposition means that the structural equations are implementable and the remaining equations and membership axioms satisfy the standard variable restriction. Since we are interested in completeness of the operational semantics of rewriting logic for the goals relevant in our application, we also verify a number of sufficient conditions that we defined in Chapter 2.

Proposition 6.3.6 (Executability of ROUPTSs)

The ROUPTS specification is executable, sort-preserving, equationally confluent, and coherent. Furthermore, the underlying MES is partially terminating in the sense that all membership, equational, and reduction goals, satisfying the condition that `whnf` is applied only to representations of weak head normalizing UPTS terms, are terminating.

Proof Sketch. Sort-preservation can be easily checked by inspection of each equation. To verify confluence observe that the entire equational specification is orthogonal and has three subspecifications: (1) the specification of explicit substitutions $[_:=_]$, $[\mathbf{shift}_]$, $[\mathbf{lift}_{_}]$, and their application $_{_}$, (2) the specification of metavariable substitution $\langle _ := _ \rangle$, (3) the specification of \mathbf{whnf} , and (4) the specification of \mathbf{lookup} . Orthogonality of (1) has already been verified for the proof of confluence in Theorem 5.3.2. Orthogonality of (2) and (4) is obvious, because there are no critical pairs, and orthogonality of (3) follows from the fact that critical pairs can be eliminated by a simple transformation, because their conditions are unsatisfiable. Furthermore, there are no critical pairs between (1), (2), (3), and (4), so that we can conclude that the MES is orthogonal and hence confluent. Similarly, coherence of the entire RWS follows from the absence of critical pairs between equations and rules.

Finally, we show partial termination of the MES, that is termination of all membership and reduction goals under the condition of the proposition. Termination of membership goals $M : \mathbf{WhNf}$ and $M : \mathbf{WhReducible}$ follows by structural induction over the terms M . For the remaining termination proof we again exploit orthogonality of our specification, which implies that it is sufficient to prove termination under an innermost reduction strategy [O'D77] (cf. proof of Theorem 5.3.3). We use the following strategy: Given a reduction goal M or G , we repeat the following two steps as long as applicable: (a) We reduce it to normal form w.r.t. (1) if this form has not been reached yet, and then (b) we select an arbitrary innermost occurrence of \mathbf{whnf} or \mathbf{lookup} and apply one of the equations from (3) or (4), respectively. Termination of this strategy follows from termination of Step (a), which holds according to Theorem 5.3.3, and from the fact that \mathbf{whnf} and \mathbf{lookup} are either eliminated in Step (b) or replaced by corresponding occurrences with smaller measures. For $\mathbf{whnf}(M)$ the measure is the minimal number of β -reduction steps necessary to reach the weak head normal form from M , and for $\mathbf{lookup}(G, X\{m\})$ the measure is the length of the context G . \square

A remarkable property of our specification is that it can be executed efficiently in the sense that we do not need an exhaustive search to verify whether $\{\{J\}\} \rightarrow \{\{\mathbf{emptyJudgementList}\}\}$ is derivable in \mathbf{ROUPTS}_S . Instead, we can use a simple execution strategy, i.e. a strategy without backtracking, and there is no additional restriction on the strategy beyond the freshness requirement for metavariables mentioned before. In fact, this is a consequence of confluence and partial termination of the rewrite part of our specification, which is stated in the following proposition.⁵

⁵Confluence modulo renaming of metavariables would be sufficient in practice, but it happens that, due to the deterministic nature of our specification, we have confluence here in the strongest sense.

Proposition 6.3.7 (Confluence and Termination of ROUPTSs)

The ROUPTS specification is rewrite-confluent and partially terminating in the sense that all rewrite goals $\{\{J\}\} \rightarrow ?$, where J is one of the judgements $\Gamma \Vdash$, $\Gamma \Vdash M : A$, or $\Gamma \Vdash_1 M : A$ with UPTS terms M, A and a UPTS context Γ , are terminating.

Proof Sketch. Confluence of rewrite rules follows from an analysis of (conditional) critical pairs. In fact, there is only a single nontrivial critical pair generated by the overlapping rules `Conv6` and `Conv7`. Termination follows from structural induction over terms using the fact that `whnf` is only applied to terms M for which the goals

$$(G \Vdash -) \quad (G \Vdash M : ?(\text{NEW})) \quad (?(\text{NEW}) \text{Sort})$$

have been already verified for some context G . As a consequence, M is well-typed in \mathbf{ROUPTS}_S , and by the chain of soundness results given in Propositions 6.3.5, 6.2.5, and 6.2.4, we conclude that M is α -equivalent to a well-typed term in \mathbf{PTS}_S , and hence strongly normalizing. \square

6.4 Final Remarks

In this chapter we have given presentations of PTSs at different levels of abstraction. Moreover, we have discussed very natural representations of these systems in MEL or RWL. Both, abstractions and representations are uniformly captured by the general notion of correspondence between entailment systems. Our treatment is guided by the general logics methodology, which explores the space of formal systems by using a particular formal system, namely rewriting logic, as a logical framework. Our representations of PTSs range from an abstract textbook representation to a more refined operational representation which exploits the executable sublanguage of rewriting logic. Apart from its methodological aspect, this chapter contains a more specific contribution, namely uniform pure type systems, a new variant of pure type systems that provides a solution to the known problem with closure under α -conversion in systems with dependent types. Our solution is inspired by earlier work of Pollack, who first pointed out the difficulty to obtain closure under α -conversion if names are taken seriously.

By instantiating our operational representation of PTSs to the calculus of constructions with a predicative universe hierarchy, we furthermore show in Appendix A how our approach directly leads to an executable prototype of the type theory in Maude. We furthermore think that the potential of our approach is by no means confined to formal representations and prototyping, but we think that it provides an interesting alternative to the implementation of type theories and typed higher-order logics, which are traditionally conducted using functional programming languages such as ML.

Finally, we would like to point out that the techniques presented in this chapter have been applied in the design and implementation of a proof assistant for OCC, the *open calculus of constructions*, an extension of the calculus of constructions that incorporates rewriting logic with its membership equational sublogic as a computational sublanguage and is presented in Chapter 8. Similar to MEL and RWL, OCC supports conditional equations, conditional assertions, and conditional rewrite rules together with an operational semantics based on a combination of conditional rewriting modulo structural equations and exhaustive goal-oriented proof search. Using the Maude rewriting engine and its reflective capabilities, we have developed an experimental version of a proof assistant for OCC that is based on the ideas on CINNI and UPTSs presented here and in Chapter 5.

Unlike the approach of Chapter 4, where we used a type theory as a *metalogic* to reason about a UNITY-style temporal logic, type theories appear in the present chapter as *object logics*. Another difference is that in contrast to Chapter 4 we have emphasized the representational aspects, since the choice of a good formal representation is important in its own right, and a major issue in the application of a framework logic like MEL and RWL. Apart from the benefit of executability that our last specification of UPTSs enjoys, a formal specification provides the basis for *formal* metatheoretic proofs. Indeed, MEL and RWL together with their initial model semantics provide very general notions of *equational inductive definitions*, a fact that has been exploited for representing several formal systems in this chapter.

The general problem of carrying out metatheoretic proofs about such closed formal systems, soundness and completeness proofs being typical examples, requires the development of useful induction principles on the basis of possibly different but related presentations of the formal system. Such induction principles can be formulated either using an *internal approach* (as in Chapter 4), e.g. by using a formal system such as OCC, which contains the framework logic as a sublogic in a suitable sense, or using an *external approach*, such as the one adopted in Twelf [SP98], where an external first-order logic is added on top of a higher-order logical framework for inductive reasoning about the representations. In a certain sense similar to the latter, but avoiding its hybrid character, one can instead use a *reflective approach* (cf. the approach to reflective metalogical frameworks presented in [BCM99, BCM00]), which introduces induction principles at the metalevel of the representation in a reflective framework such as rewriting logic.

6.5 Acknowledgements

Support for this work by DARPA and NASA (Contract NAS2-98073), by Office of Naval Research (Contract N00014-96-C-0114), by National Science Foundation

Grant (CCR-9633363), and by a DAAD grant in the scope of HSP-III is gratefully acknowledged. I am indebted to José Meseguer for his suggestion to use the general logics methodology as a framework for this presentation and for his advice and collaboration in the context of our joint paper [SM99], which has finally evolved into this chapter. I also appreciate the help of Steven Eker, who gave valuable hints concerning the C++ implementation of Maude, which enabled me to extend the engine by a built-in equational module for metavariable substitution (cf. Appendix A). Furthermore, I would like to thank Manuel Clavel, Narciso Martí-Oliet and the anonymous referees of our paper [SM99] for their useful comments, and, last but not least, Cesar Muñoz for many discussions on calculi of explicit substitutions and on the difficulties caused by α -conversion in type theories with explicit names.

Chapter 7

Classical Logic via Type Theory:

A Proof-Theoretic Approach
to the HOL/Nuprl Connection

In type theories exploiting the propositions-as-types interpretation such as Martin-Löf’s type theory [PSN90], the type theory of Nuprl [CAB⁺86], and the calculus of constructions [CH88] the intuitionistic (higher-order) logic resides in one or more propositional universes of types, which are *intensional* in the sense that logically equivalent propositions are not necessarily equal. The propositional universe is clearly distinguished from the boolean data type which contains exactly two elements and hence is *extensional* in the sense that logical equivalence of booleans implies equality. In this chapter we investigate Doug Howe’s connection [How96a, How98a] between the classical logic of the HOL system [GM93b] and a classical variant of the Nuprl type theory [CAB⁺86] from the viewpoint of general logics [Mes89a]. The main idea behind this connection is to rediscover the classical extensional logic present in the boolean data type of an expressive type theory like Nuprl using a hybrid computational/set-theoretic semantics [How96b, How97]. This semantics justifies a *classical variant of Nuprl* [How97] which is equipped with the *axiom of the excluded middle*. In the remainder of this chapter we will simply refer to this variant as Nuprl. The benefit of this approach is that an extensional and an intensional logic can coexist in a single framework and that we can take advantage of both. The former is well suited to deal directly with external classical knowledge, whereas for the latter theorem-proving is well supported in the Nuprl system and, moreover, it allows extraction of programs from constructive proofs.

In this chapter, which is a revised version of [SNM00, SNM01], the connection between the HOL and Nuprl proof assistants earlier introduced by Howe is reexamined from a proof-theoretic point of view as a translation between the sentences of HOL and Nuprl followed by a theory interpretation. Using standard terminology from the theory of general logics [Mes89a], we establish a proof-theoretic correctness result of the translation which nicely complements Howe’s semantics-based argument. We mainly focus on the theoretical aspects of the HOL/Nuprl connection in this chapter, but we also briefly discuss two lines of practical work, namely, an executable formal specification of the HOL/Nuprl theory translator in Maude and a proof translator, which emerged as an application of our proof-theoretic approach and has been implemented by Pavel Naumov on top of Nuprl [NSM01].

We found that the HOL/Nuprl connection as it was originally implemented in [How99] can be understood in the framework of general logics as a composition of two stages:

1. The first stage is a *translation* of an *axiomatic HOL theory* into an *axiomatic Nuprl theory*. The use of the term “axiomatic” emphasizes the fact that the theories are not necessarily only definitional extensions of the basic logic. The translation is, by its very nature, metalogical, in the sense that, by relating two different logics, it is beyond the scope of each of them. It is

therefore a critical stage whose correctness cannot be reduced to that of the two theorem provers involved and requires careful analysis. Furthermore, defining the translation as a mathematical notion in its own right has the important advantage of being independent of the theorem provers, so that it is not affected by internal changes to those systems provided the logics remain the same.

2. The second stage is the *interpretation* of an axiomatic Nuprl theory inside Nuprl. In this way we can often obtain a *computationally meaningful* theory, which is closer to the spirit of Nuprl, that favors definitional extensions. As in Howe's extension of Nuprl, this interpretation stage can take place inside the Nuprl system in a formally rigorous way. However, since there are many possible choices for the interpretation, and certain proofs to be carried out, user interaction is required.

The present chapter clarifies the translation in terms of maps between entailment systems, which provide an abstract metalogical setting for formal systems within the theory of general logics [Mes89a]. The questions we address are: What are the entailment systems involved? How can the translation be explained at an abstract level that makes its correctness easy to verify? The interpretation stage on the other hand does not take place between different logics but within a single logic, so that an actual interpretation can be formally verified in a rigorous way. The critical interlogic translation stage should be kept as simple as possible, so that we can achieve a high confidence that it is indeed correct. Furthermore, the map of entailment systems defines a theory translator, that we have formally specified in Maude in an executable way, and we have used to translate large libraries of HOL theories into Nuprl, as explained in the last section. As another benefit of the proof-theoretic approach, the correctness result in terms of entailment systems immediately suggests extending the HOL/Nuprl connection to a translator that does not only translate theorems but also their proofs. In the last section we also report on some practical experience with an implementation of such a proof translator on top of the Nuprl system.

Ideally, there is a close connection between the soundness of the mapping between the two formal systems and proof translation: If \mathcal{L} and \mathcal{L}' are the source and target logics, respectively, and α is the mapping of \mathcal{L} into \mathcal{L}' then *soundness*¹ is the property that $\Gamma \vdash_{\mathcal{L}} P$ implies $\alpha(\Gamma) \vdash_{\mathcal{L}'} \alpha(P)$ for any set Γ of axioms and any formula P . Hence, a proof of soundness would demonstrate that for each proof of P from Γ in \mathcal{L} there is a corresponding proof of $\alpha(P)$ from $\alpha(\Gamma)$ in \mathcal{L}' . Furthermore, if the soundness proof is conducted in a constructive way as in this chapter, it implicitly contains an algorithm for proof translation. In fact, in

¹Soundness is the central property of a map of entailment systems, which represent the notion of derivability in general logics [Mes89a]. For the purpose of the introduction we adopt a simplified viewpoint disregarding the issue of signature translation.

our case the proof translator from HOL into NuPRL is an algorithm, which is essentially extracted from a mathematical proof of soundness.

Obviously, soundness of the underlying mapping is the main theoretical requirement for the correctness of the translator. Notice, however, that soundness can always be achieved by extending the target system by additional axioms and inference rules, as it is often necessary in practice. Of course, such an extension could make the target system inconsistent, which is why the soundness proof is meaningful only in the presence of a consistency proof for the target system extension. Typically, such a consistency proof is done using an abstraction of the target system, e.g. by construction of a semantic model. In a more general setting, where we work relative to arbitrary theories, a suitably general *model preservation* property² property is that each model M of \mathcal{L} can be obtained from a model M' of \mathcal{L}' by a mapping β such that $\beta(M') \models_{\mathcal{L}} P$ iff $M' \models_{\mathcal{L}'} \alpha(P)$. Although the model preservation property, which in the case of HOL and Nuprl follows from Howe's hybrid computational/set-theoretic semantics for Nuprl [How96b, How97], is generally important for the translation mapping to be semantically meaningful, it is of little use for the implementation of a proof translator which typically relies on proof-theoretic constructions.

Another noteworthy point is that the translation does not rely on the more advanced features of Nuprl that go beyond Martin-Löf's extensional polymorphic type theory as presented in [PSN90]. Therefore, the translation can also be regarded as a translation between HOL and a classical variant of Martin-Löf's type theory itself, and actually this is the subset of Nuprl that we will be concerned with.

7.1 Preliminaries

In the sequel we often make use of finite lists of objects. We use $[]$ for the empty list and a comma to denote list concatenation. Sometimes we employ a suggestive notation, like σ^m which makes explicit the length m of the list. If σ is a metavariable ranging over certain objects, σ^m is a metavariable ranging over lists of length m of such objects. The elements of the list can be accessed by σ_i^m for $i \in \{1 \dots m\}$. If the objects have a certain structure, say if they are of the form $x : \sigma$, we also write $x^n : \sigma^n$ for a list of length n of such objects. So we assume notational conventions such as $x^n : \sigma^n = x_1^n : \sigma_1^n, \dots, x_n^n : \sigma_n^n$.

²This is usually expressed in the context of a map of institutions [GB92], which constitute the model-theoretic component of general logics [Mes89a].

7.2 The Logic of HOL

HOL [GM93b] is a proof development system based on higher-order logic. It uses a Hindley-Milner-style polymorphic lambda-calculus together with an axiomatization of the logic using polymorphic equality, implication and Hilbert's choice operators as basic ingredients. As most higher-order logics it is a logic of total functions. The HOL system favors conservative theory extension (to introduce new constants and/or new data types) but axiomatic extensions are also supported.

The following presentation of HOL is close to [GM93b], but we have cast it into a categorical setting, which is suggested by the framework of general logics [Mes89a].

7.2.1 Syntactic Categories

We assume countably infinite disjoint sets of *constants*, *variables*, *type constants*, and *type variables*. Metavariables for type constants and constants are ν and c , respectively. We use the metavariables α , β , and γ for arbitrary but distinct type variables, and metavariables x , y , and z for arbitrary but distinct variables.

A *type constant declaration* takes the form $\nu : \mathbf{Type}^n \rightarrow \mathbf{Type}$, where ν is the *type constant* and n is its *arity*.³ It is called an *atomic type declaration* if $n = 0$ (in this case we write $\nu : \mathbf{Type}$) and a *type operator declaration* otherwise. A *type signature* Ω is a finite set of type constant declarations with distinct type constants. We furthermore assume that a type signature Ω contains a *standard type constant declaration* $\circ : \mathbf{Type}$ and refer to \circ as the *type of propositions*. The category of type signatures **HolTypeSign** has type signatures as objects, and its morphisms are functions mapping type constants in one signature to type constants in the other preserving the standard type constants and the arity of each type constant.

A *type context* takes the form $\alpha^n : \mathbf{Type}^n$ with all the type variables α_i^n distinct. The category of type contexts is denoted by **HolTypeCtxt**. Its morphisms, called *type context inclusions*, are inclusions between the sets of declared type context variables. A *full type context* has the form $\Omega \vdash \alpha^n : \mathbf{Type}^n$ and the category of full type contexts is **HolFullTypeCtxt** = **HolTypeSign** \times **HolTypeCtxt**.

The set $HolType(\Omega)$ of *types* over a type signature Ω is inductively defined as follows:

1. $\alpha^n : \mathbf{Type}^n \vdash \alpha_i^n$ is in $HolType(\Omega)$ for each $i \in \{1 \dots n\}$.
2. If $\alpha^n : \mathbf{Type}^n \vdash \sigma_i^m$ is in $HolType(\Omega)$ for each $i \in \{1 \dots m\}$ and

³ \mathbf{Type} is part of our suggestive notation, but it does not have a formal status in HOL.

$\nu : \mathbf{Type}^m \rightarrow \mathbf{Type}$ is a type constant declaration in Ω , then

$\alpha^n : \mathbf{Type}^n \vdash \nu(\sigma^m)$ is in $\mathit{HolType}(\Omega)$.

We write ν instead of $\nu(\sigma^m)$ when $m = 0$.

3. If $\alpha^n : \mathbf{Type}^n \vdash \sigma$ and $\alpha^n : \mathbf{Type}^n \vdash \tau$ are in $\mathit{HolType}(\Omega)$, then $\alpha^n : \mathbf{Type}^n \vdash \sigma \rightarrow \tau$ is in $\mathit{HolType}(\Omega)$.

An expression σ is said to be a *type* over a full type context $\Omega \vdash \alpha^n : \mathbf{Type}^n$ if $\alpha^n : \mathbf{Type}^n \vdash \sigma$ is a type over Ω . The set of types over $\Omega \vdash \alpha^n : \mathbf{Type}^n$ is denoted by $\mathit{HolType}(\Omega \vdash \alpha^n : \mathbf{Type}^n)$. Each full type context morphism is lifted to a function between sets of types in the natural way. This defines a functor $\mathit{HolType} : \mathbf{HolFullTypeCtxt} \rightarrow \mathbf{Set}$. A *full type* has the form $\Omega \vdash \alpha^n : \mathbf{Type}^n \vdash \sigma$, where σ is a type over $\Omega \vdash \alpha^n : \mathbf{Type}^n$.

Let Ω be a type signature. A *constant declaration* over Ω takes the form $\alpha^n : \mathbf{Type}^n \vdash c : \sigma$, where σ is a type over $\Omega \vdash \alpha^n : \mathbf{Type}^n$ and where each type variable α_i^n occurs in σ . Here c is a *constant* and σ is called the *type* of c . A *signature* Δ over Ω is a set of constant declarations over Ω with distinct constants. We assume that Δ contains *standard constant declarations* $\vdash \Rightarrow : \circ \rightarrow \circ \rightarrow \circ$ and $\alpha : \mathbf{Type} \vdash = : \alpha \rightarrow \alpha \rightarrow \circ$ for *logical implication* and *polymorphic equality*, respectively. A *signature morphism* between signatures Δ and Δ' over Ω is a function from Δ to Δ' which preserves standard constants and the type of each constant. The category of signatures over Ω is denoted by $\mathit{HolSign}(\Omega)$. Actually, we have a functor $\mathit{HolSign} : \mathbf{HolTypeSign} \rightarrow \mathbf{Cat}$ which lifts each type signature morphism $f : \Omega \rightarrow \Omega'$ to a functor $\mathit{HolSign}(f) : \mathit{HolSign}(\Omega) \rightarrow \mathit{HolSign}(\Omega')$. A *full signature* $\Omega \vdash \Delta$ consists of a type signature Ω and a signature Δ over Ω . The category of full signatures is $\sum \Omega : \mathbf{HolTypeSign} . \mathit{HolSign}(\Omega)$ and is denoted by $\mathbf{HolFullSign}$.⁴

Given a full type context $\Omega \vdash \alpha^n : \mathbf{Type}^n$, a *context* over $\Omega \vdash \alpha^n : \mathbf{Type}^n$ has the form $x^m : \sigma^m$, with distinct variables x_i^m and types σ_i^m over $\Omega \vdash \alpha^n : \mathbf{Type}^n$. The category of contexts over $\Omega \vdash \alpha^n : \mathbf{Type}^n$ is denoted by $\mathit{HolCtxt}(\Omega \vdash \alpha^n : \mathbf{Type}^n)$ and its morphisms, called *context inclusions*, are inclusions between the sets of declared context variables such that the type of each variable is preserved.⁵ Actually, we have a functor $\mathit{HolCtxt} : \mathbf{HolFullTypeCtxt} \rightarrow \mathbf{Cat}$ lifting full type context morphisms to functors. A *full context* takes the form $\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m$, where Δ is a signature over Ω and where $x^m : \sigma^m$ is a context over $\Omega \vdash \alpha^n : \mathbf{Type}^n$. The category of *full contexts* is $\mathbf{HolFullCtxt} = \sum \Omega : \mathbf{HolTypeSign} . \mathit{HolSign}(\Omega) \times \sum \omega : \mathbf{HolTypeCtxt} . \mathit{HolCtxt}(\Omega \vdash \omega)$.

⁴For a functor $F : \mathbf{I} \rightarrow \mathbf{Cat}$ the category ΣF , also written $\sum x : \mathbf{I} . F(x)$, is defined by the general Grothendieck construction [BW90] which generalizes the set-theoretic sum.

⁵More powerful morphisms that allow renaming of context variables are possible if the notion of term is generalized so that *accidental hiding* of context variables by λ -abstractions is avoided. A possible representation to formalize this is provided by the CINNI calculus (see Chapter 5).

Let $\Omega \vdash \Delta$ be a full signature. The set $HolTrm(\Omega \vdash \Delta)$ of typed terms over $\Omega \vdash \Delta$ is a subset of expressions of the form $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash M : \tau$ such that $\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m$ is a full context. It is inductively defined as follows:⁶

1. $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash x_i^m : \sigma_i^m$ is in $HolTrm(\Omega \vdash \Delta)$ for each $i \in \{1 \dots m\}$.
2. If $\beta^k : \mathbf{Type}^k \vdash c : \tau$ is a constant declaration in Δ and $\alpha^n : \mathbf{Type}^n \vdash \tau_i^k$ is a type over Ω for each $i \in \{1 \dots k\}$, then $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash c_{\tau[\tau^k/\beta^k]} : \tau[\tau^k/\beta^k]$ is in $HolTrm(\Omega \vdash \Delta)$.
3. If $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m, y : \rho \vdash M : \tau$ is in $HolTrm(\Omega \vdash \Delta)$, then $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash (\lambda y : \rho. M) : \rho \rightarrow \tau$ is in $HolTrm(\Omega \vdash \Delta)$.
4. If $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash M : \rho \rightarrow \tau$ and $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash N : \rho$ are in $HolTrm(\Omega \vdash \Delta)$, then $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash (M N) : \tau$ is in $HolTrm(\Omega \vdash \Delta)$.

If $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash M : \tau$ is a typed term over $\Omega \vdash \Delta$, we say that M is a *term* over $\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m$ and τ is its *type*. We will make use of the property of the HOL type system that M has a unique type. The set of *terms* over a full context $\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m$ is denoted by $HolTrm(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)$. Each full context morphism is lifted to a function between sets of terms. This defines a functor $HolTrm : \mathbf{HolFullCtxt} \rightarrow \mathbf{Set}$. On terms we assume the usual notion of α -conversion \equiv_α , but we do *not* identify terms that are α -convertible.

A *formula* over a full context $\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m$ is a term over $\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m$ of type \mathbf{o} . A *sentence* over $\Omega \vdash \Delta$ has the form $\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash A^k \vdash A$ with formulae A_i^k and A over $\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m$. The set of sentences over $\Omega \vdash \Delta$ is denoted by $HolSen(\Omega \vdash \Delta)$. Each signature morphism can be lifted to a function between sets of sentences. So we have a functor $HolSen : \mathbf{HolFullSign} \rightarrow \mathbf{Set}$.

We conclude the presentation of the syntax of HOL with an overview of the syntactic entities introduced so far:

⁶To avoid any technicalities connected with the treatment of bound variables we specialize the notion of terms in HOL by enforcing the convention that all bound and free variables are distinct.

Objects	Category/Functor
$\Omega \vdash \alpha^n : \text{Type}^n \vdash \sigma$	HolFullType
$\Omega \vdash \Delta \vdash \alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash M$	HolFullTerm
$\Omega \vdash \Delta \vdash \alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H \vdash A$	HolFullSen
σ	$\text{HolType} : \text{HolFullTypeCtxt} \rightarrow \text{Set}$
M	$\text{HolTrm} : \text{HolFullCtxt} \rightarrow \text{Set}$
$\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H \vdash A$	$\text{HolSen} : \text{HolFullSign} \rightarrow \text{Cat}$
Ω	HolTypeSign
$\Omega \vdash \Delta$	HolFullSign
$\Omega \vdash \alpha^n : \text{Type}^n$	HolFullTypeCtxt
$\Omega \vdash \Delta \vdash \alpha^n : \text{Type}^n \vdash x^m : \sigma^m$	HolFullCtxt
Ω	HolTypeSign
Δ	$\text{HolSign} : \text{HolTypeSign} \rightarrow \text{Cat}$
$\alpha^n : \text{Type}^n$	$\text{HolTypeCtxt} : \text{HolTypeSign} \rightarrow \text{Cat}$
$x^m : \sigma^m$	$\text{HolCtxt} : \text{HolFullTypeCtxt} \rightarrow \text{Cat}$

7.2.2 Deduction and Entailment

Given a full signature $\Sigma = (\Omega \vdash \Delta)$, the *deductive system of HOL* over Σ is the binary relation $(\vdash_1^{\text{Hol}}) \subseteq \mathcal{P}_{\text{fin}}(\text{HolSen}(\Sigma)) \times \text{HolSen}(\Sigma)$ specified by the *inference rules of HOL*, which are the following:

$\frac{}{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash A \vdash A}$	(ASSUME)
$\frac{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash G \vdash A \quad \alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H \vdash A \Rightarrow B}{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H, G \vdash B}$	(MP)
$\frac{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H \vdash B}{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash G \vdash A \Rightarrow B}$	(DISCH)

where G is obtained from H by removing all formulae α -equivalent to A .

$\frac{}{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash \vdash M =^\rho M}$	(REFL)
$\frac{}{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash \vdash ((\lambda y : \rho . M) N) =^\tau M[N/y]}$	(BETA-CONV)
$\frac{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H_i^k \vdash M_i^k =^{\rho_i^k} N_i^k \text{ for all } i \in \{1 \dots k\} \quad \alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash G \vdash A[M^k/z^k]}{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H^k, G \vdash A[N^k/z^k]}$	(SUBST)
$\frac{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m, y : \rho \vdash H \vdash M =^\tau N}{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H \vdash (\lambda y : \rho . M) =^{\rho \rightarrow \tau} (\lambda y : \rho . N)}$	(ABS)

where y is not free in H .

$\frac{\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash H \vdash A}{\beta^k : \text{Type}^k \vdash x^m : \sigma^m[\tau^n/\alpha^n] \vdash H \vdash A[\tau^n/\alpha^n]}$	(INST-TYPE)
--	-------------

where $\beta^k : \text{Type}^k \vdash \tau_i^n$ are types and α_i^n does not occur in H for $i \in \{1 \dots n\}$.

For better readability we have written $M =^\rho N$ instead of $((=_{\rho \rightarrow \rho \rightarrow \circ}) M) N$.

As in the HOL system, the assumptions on the lefthand side of \vdash are lists instead of sets, which is justified by the fact that HOL inference rules do not depend on their order. Another noteworthy point is that we have not identified α -equivalent terms. The α -conversion is, however, used implicitly in the HOL implementation of the rules **MP** and **SUBST**, which makes their implementation slightly more liberal than the rules given above. To compensate for this, we assume in addition to the rules above a rule for renaming of bound variables, thereby avoiding a commitment to the technicalities of the HOL implementation.

Derivability of a sentence ϕ from a set of sentences Γ in the deductive system of HOL is defined in the conventional way employing sequential proofs: A sentence ϕ can be *derived* from a set of sentences Γ iff there exists a sequence ϕ_1, \dots, ϕ_n such that **(1)** $\phi = \phi_n$ and **(2)** for all i either $\phi_i \in \Gamma$ or $\Gamma' \vdash_1^{Hol} \phi_i$ for some Γ' such that $\Gamma' \subseteq \Gamma \cup \{\phi_1, \dots, \phi_{i-1}\}$. The sequence ϕ_1, \dots, ϕ_n is called a *sequential proof* of ϕ from Γ .

Given a full signature Σ , the deductive system of HOL defines the *entailment relation* $(\vdash_\Sigma^{Hol}) \subseteq \mathcal{P}_{\text{fin}}(\text{HolSen}(\Sigma)) \times \text{HolSen}(\Sigma)$ of HOL as follows: $\Gamma \vdash_\Sigma^{Hol} \phi$ holds iff ϕ can be derived from Γ in the deductive system of HOL. According to the terminology of [Mes89a] **(HolFullSign, HolSen, \vdash^{Hol})** constitutes an entailment system. We call it the *entailment system of HOL*.

7.2.3 Theories

An (*axiomatic*) *HOL theory* (Σ, Γ) consists of a full signature Σ together with a set Γ of sentences over Σ called *axioms*. A full signature morphism $H : \Sigma \rightarrow \Sigma'$ is said to be a *theory morphism* $H : (\Sigma, \Gamma) \rightarrow (\Sigma', \Gamma')$ iff $\Gamma' \vdash_{\Sigma'}^{Hol} \text{HolSen}(H)(\phi)$ for all $\phi \in \Gamma$. This gives a category of theories that will be denoted by **HolTh**.

In contrast to the original presentation of HOL, we have removed from the minimal/logical theory given below all constants presupposed by the inference rules, namely $\circ, =, \Rightarrow$. The constants are now standard constants and therefore a fixed part of every signature.

All mathematical developments in HOL take place in *standard theories* extending the *logical theory bool*. Therefore, for the remainder of this chapter we define **bool** as \circ , and we use **bool** to emphasize that we are working with classical extensional logic. Indeed, it is the logical theory **bool** which makes HOL a logic of this kind. The logical theory **bool** = (Σ, Γ) has a signature Σ which contains the standard type constant **bool** (i.e. \circ), and the standard constants $=$ and \Rightarrow . The remaining constants and axioms of **bool** are the following.

Σ contains the constants $\mathbf{T} : \text{bool}$, $\mathbf{F} : \text{bool}$, $\neg : \text{bool} \rightarrow \text{bool}$, $\wedge : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$, $\vee : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$, $\forall : (\alpha \rightarrow \text{bool}) \rightarrow \text{bool}$, $\exists : (\alpha \rightarrow$

$\mathbf{bool}) \rightarrow \mathbf{bool}$, and Hilbert's choice operator $\epsilon : (\alpha \rightarrow \mathbf{bool}) \rightarrow \alpha$.

Γ contains suitable definitional axioms to equip these constants with their standard (classical) meaning. Indeed all the constants above are defined in terms of (higher-order) equality $=$, implication \Rightarrow , and Hilbert's choice operator ϵ . Moreover, there are some nondefinitional axioms in Γ , namely

$$\begin{array}{ll} \vdash \forall x : \mathbf{bool} . (x =^{\mathbf{bool}} \mathbf{T}) \vee (x =^{\mathbf{bool}} \mathbf{F}) & \text{(BOOL-CASES-AX)} \\ \vdash \forall x : \mathbf{bool} . \forall y : \mathbf{bool} . (x \Rightarrow y) \Rightarrow (y \Rightarrow x) \Rightarrow (x =^{\mathbf{bool}} y) & \text{(IMP-ANTISYM-AX)} \\ f : \alpha \rightarrow \beta \vdash (\lambda x : \alpha . (f x)) =^{\beta} f & \text{(ETA-AX)} \\ \vdash \forall P : \alpha \rightarrow \mathbf{bool} . \forall x : \alpha . (P x) \Rightarrow (P (\epsilon P)) & \text{(SELECT-AX)} \end{array}$$

with the usual abbreviation $\forall x : \sigma . M$ for $\forall (\lambda x : \sigma . M)$.

In addition, Σ contains an atomic type constant \mathbf{ind} , and there is an axiom **INFINITY-AX** in Γ stating that \mathbf{ind} has an infinite number of elements.

The theory \mathbf{bool} has further constants with definitional axioms, namely $\mathbf{OneOne} : (\alpha \rightarrow \beta) \rightarrow \mathbf{bool}$, $\mathbf{Onto} : (\alpha \rightarrow \beta) \rightarrow \mathbf{bool}$, $\mathbf{TypeDefinition} : (\alpha \rightarrow \mathbf{bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow \mathbf{bool}$, which are used to introduce new data types in a conservative way, by declaring them to be isomorphic to subsets of existing data types.

Usually we will only be interested in so-called *standard theories* extending the logical theory \mathbf{bool} . However, with the modifications mentioned above HOL without additional theories is also a reasonable system to study.

7.3 The Type Theory of Nuprl

Nuprl's type theory [CAB⁺86] is a variant of Martin-Löf's 1982 polymorphic, extensional type theory (the version contained in [PSN90] with extensional equality). Although Nuprl has very advanced features (e.g. subset types, subtyping, quotient types, recursive types, intersection types, partial functions, and direct computation, which make these type theories rather different), semantically Nuprl can be viewed as an extension of Martin-Löf's type theory, in the sense that it has a richer variety of types and more flexible rules which give rise to a richer collection of well-typed terms.

Apart from the advanced features mentioned above, there is a major difference in the meaning of sequents between these two type theories. In Martin-Löf's type theory well-formedness of the lefthand side of a sequent is presupposed, i.e., it is implied by a derivable sequent. This is not the case in Nuprl, where well-formedness is part of the assumption expressed by the lefthand side of the sequent. Details about this difference can be found in [All87]. For our purposes here this difference is not important, since we consider only rules where all well-formedness conditions are made explicit. In this way we obtain a set of rules which are valid and derivable in Martin-Löf's type theory as well as in Nuprl.

In contrast to HOL, terms in Nuprl are neither explicitly nor implicitly equipped with types. Instead types are ordinary terms, and the judgement that a type can be *assigned* to a term is a sentence in the logical language which is not decidable in general. Indeed, since Nuprl is polymorphic, a term may be associated with different types.

Even though the advanced features of Nuprl provide an important motivation for the HOL/Nuprl connection, the connection itself does not rely on features that go beyond Martin-Löf's type theory as presented in [PSN90]. Indeed, in the following we present the part of classical Nuprl that is sufficient to establish the HOL/Nuprl connection. We give a simplified presentation based on [CAB⁺86], but, as for HOL, we have used a categorical formulation in order to achieve a uniform and systematic description of the translation.

7.3.1 Syntactic Categories

We assume countably infinite disjoint sets of *constants* and *variables* with the same metavariable conventions as for HOL.

A *Nuprl signature* Σ is a set of constants. We assume that Σ contains a number of *standard constants* including \mathbb{U}_i for $i \geq 1$, **app**, λ , **pi**, and further constants that we do not explicitly mention here. We denote the category of signatures and bijective functions preserving standard constants by **NuprlSign**.

Given a signature Σ , we inductively define the set $NuprlTrm(\Sigma)$ of *terms* over Σ as follows:

1. $x^m \vdash x_i^m$ is in $NuprlTrm(\Sigma)$ for each $i \in \{1 \dots m\}$.
2. If c is a constant in Σ and $x^m \vdash M_i^k$ is in $NuprlTrm(\Sigma)$ for each $i \in \{1 \dots k\}$, then $x^m \vdash c(M^k)$ is in $NuprlTrm(\Sigma)$.
We write c instead of $c(M^k)$ when $k = 0$.
3. If $x^m, y \vdash M$ is in $NuprlTrm(\Sigma)$, then $x^m \vdash (y.M)$ is in $NuprlTrm(\Sigma)$.

If $x^m \vdash M$ is a term over Σ , we say that M is a *term* over $\Sigma \vdash x^m$. Instead of making constants c explicit by writing $c(M^k)$, we often use a more convenient notation⁷ such as $(M N)$ for function application **app**(M, N), $\lambda x . M$ for λ -abstraction $\lambda(x.M)$, and $(x : S \rightarrow T)$ for dependent types **pi**($S, (x.T)$).

Given a signature Σ , a *context* over Σ takes the form $x^m : S^m$, where S_{i+1}^m is a term over $\Sigma \vdash x_1^m, \dots, x_i^m$ and where all the variables x_i^m are distinct. A *context*

⁷Nuprl employs user-definable display forms to bridge the gap between the formal notion of term and the way terms are presented to a user of the system.

inclusion is an inclusion between the sets of declared context variables such that the type of each variable is preserved. A *full Nuprl context* is of the form $\Sigma \vdash x^m : S^m$, where $x^m : S^m$ is a context over Σ . The category of contexts over Σ with context inclusions as morphisms is denoted by $NuprlCtxt(\Sigma)$. Furthermore, each signature morphism can be lifted to a morphism between contexts, so that we obtain a functor $NuprlCtxt : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$. The category of full contexts is $\mathbf{NuprlFullCtxt} = \sum \Sigma : \mathbf{NuprlSign} . NuprlCtxt(\Sigma)$.⁸

A *term* over a full context $\Sigma \vdash x^m : S^m$ is precisely a term over $\Sigma \vdash x^m$. The set of terms over $\Sigma \vdash x^m : S^m$ is denoted by $NuprlTrm(\Sigma \vdash x^m : S^m)$. Each full context morphism is lifted to a function between sets of terms. This defines a functor $NuprlTrm : \mathbf{NuprlFullCtxt} \rightarrow \mathbf{Set}$.

Given a signature Σ , we also define *sentences* over Σ , which have the form $x^m : S^m \vdash T$, with a context $x^m : S^m$ over Σ and a term T over $\Sigma \vdash x^m : S^m$. The set of sentences over Σ is denoted by $NuprlSen(\Sigma)$. For each fixed signature Σ this is a category if we take context inclusions lifted to sentences as morphisms. Moreover, signature morphisms can be lifted to functors between such categories, so that we have a functor $NuprlSen : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$.

It is worthwhile mentioning that this notion of sentence is a proper specialization of the judgements admitted in [CAB⁺86], which take the form $x^m : S^m \vdash T[\text{ext } P]$, the pragmatic intention being that the extraction term P is usually hidden from the user, but it can be extracted from a completed proof. Assuming that T is intended as a proposition, we are usually not interested in the extraction term P . Therefore we will only use *abstract judgements* of the form $x^m : S^m \vdash T$. We define such an abstract judgement to be derivable iff $x^m : S^m \vdash T[\text{ext } P]$ is derivable for some P . To make explicit an element P that inhabits a type T , we use equality types of the form $M = N \in T$, so that we can derive $x^m : S^m \vdash T[\text{ext } P]$ iff $x^m : S^m \vdash P \in T$ is derivable, where we assume that $P \in T$ is identified with $P = P \in T$.

Again we conclude the presentation of the syntax with an overview:

Objects	Category/Functor
$\Sigma \vdash x^m : S^m \vdash T$	NuprlFullTerm
$\Sigma \vdash x^m : S^m \vdash M = N \in T$	NuprlFullSen
M	$NuprlTrm : \mathbf{NuprlFullCtxt} \rightarrow \mathbf{Set}$
$x^m : S^m \vdash M = N \in T$	$NuprlSen : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$
Σ	NuprlSign
$\Sigma \vdash x^m : S^m$	NuprlFullCtxt
Σ	NuprlSign
$x^m : S^m$	$NuprlCtxt : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$

⁸Recall that for a functor $F : \mathbf{I} \rightarrow \mathbf{Cat}$ we use the notation $\sum x : \mathbf{I} . F(x)$ to denote the general Grothendieck construction [BW90].

7.3.2 Derivability and Entailment

We define an *entailment relation* $(\vdash_{\Sigma}^{Nuprl}) \subseteq \mathcal{P}_{\text{fin}}(NuprlSen(\Sigma)) \times NuprlSen(\Sigma)$ where $\Gamma \vdash_{\Sigma}^{Nuprl} \phi$ holds iff the sentence ϕ can be derived using the sentences in Γ as assumptions. Here, we consider derivability in the fragment of classical Nuprl given by the inference rules below, which are either basic inference rules or trivially derivable in Nuprl. $(\mathbf{NuprlSign}, NuprlSen, \vdash^{Nuprl})$ constitutes an entailment system. We call it the *entailment system of Nuprl*.

$$\frac{H, G \vdash T : \mathbb{U}_i}{H, x : T, G \vdash x \in T} \quad (\text{HYP})$$

$$\frac{H \vdash S : \mathbb{U}_i \quad H, G \vdash T}{H, x : S, G \vdash T} \quad \text{if } x \text{ is not declared in } H \text{ or } G \quad (\text{THIN})$$

$$\frac{H, G \vdash N \in S \quad H, x : S, G \vdash T}{H, G \vdash T[N/x]} \quad (\text{CUT})$$

$$\frac{H, x : S \vdash T : \mathbb{U}_i \quad H \vdash T[M/x] \quad H \vdash M = N \in S}{H \vdash T[N/x]} \quad (\text{SUBST})$$

$$\frac{}{H \vdash \mathbb{U}_i \in \mathbb{U}_j} \quad \text{if } i < j \quad (\text{UNIV-INTRO})$$

$$\frac{H \vdash M \in \mathbb{U}_i}{H \vdash M \in \mathbb{U}_j} \quad \text{if } i < j \quad (\text{CUMULATIVITY})$$

$$\frac{H \vdash T \in \mathbb{U}_i \quad H \vdash M \in T \quad H \vdash N \in T}{H \vdash (M = N \in T) \in \mathbb{U}_i} \quad (\text{EQUALITY-FORM})$$

$$\frac{H \vdash M = N \in T}{H \vdash N = M \in T} \quad (\text{EQUALITY-SYM})$$

$$\frac{H \vdash K = L \in T \quad H \vdash L = M \in T}{H \vdash K = M \in T} \quad (\text{EQUALITY-TRANS})$$

$$\frac{H \vdash A \in \mathbb{U}_i \quad H, x : A \vdash B \in \mathbb{U}_i}{H \vdash (x : A \rightarrow B) \in \mathbb{U}_i} \quad (\text{FUN-FORM})$$

$$\frac{H \vdash S \in \mathbb{U}_i \quad H, x : S \vdash M \in T}{H \vdash (\lambda x . M) \in (x : S \rightarrow T)} \quad (\text{FUN-INTRO})$$

$$\frac{H \vdash N \in S \quad H \vdash M \in (x : S \rightarrow T)}{H \vdash (M N) \in T[N/x]} \quad (\text{FUN-ELIM})$$

$$\frac{H \vdash S \in \mathbb{U}_i \quad H, x : S \vdash M = N \in T}{H \vdash (\lambda x . M) = (\lambda x . N) \in (x : S \rightarrow T)} \quad (\text{FUN-XI})$$

$$\frac{H \vdash N \in S \quad H, x : S \vdash M \in T}{H \vdash ((\lambda x . M) N) = M[N/x] \in T[N/x]} \quad (\text{FUN-BETA})$$

$$\frac{H \vdash f \in (x : A \rightarrow B) \quad H \vdash g \in (x : A \rightarrow B) \quad H, x : A \vdash (f x) = (g x) \in B}{H \vdash f = g \in (x : A \rightarrow B)} \quad (\text{FUN-EXT})$$

In addition to these rules we assume a rule for renaming of bound variables, and we make use of standard rules for

1. the *empty type* `Void`,
2. the *singleton type* `Unit` with an element `•`,
3. the *boolean type* `ℬ` with elements `tt` and `ff`
and elimination operator `if M then N else N'`,
4. the *type of natural numbers* `ℕ` with standard operations,
5. the *dependent product types* $x : S \times T$ with a pair constructor $\langle M, N \rangle$
and elimination operators `1of(M)`, `2of(M)`,
6. the *disjoint union types* $S+T$ with left and right injections `inl(M)`, `inr(M)`
and elimination operator `decide(M, (x.N), (x.N'))`,
7. and *subset types* $\{x : T \mid A\}$.

Finally, we assume a standard constant `inhabited` with the classical axiom scheme given later.

The propositions-as-types interpretation is made explicit using the following logical abbreviations: $\mathbb{P}_i = \mathbb{U}_i$, `False` = `Void`, `True` = `Unit`, $A \wedge B = A \times B$, $A \vee B = A + B$, $\forall x : A. B = x : A \rightarrow B$, and $\exists x : A. B = x : A \times B$. If x is not free in B we write $A \Rightarrow B$ and $A \rightarrow B$ instead of $\forall x : A. B$ and $x : A \rightarrow B$, respectively. Furthermore, $\neg A$ abbreviates $A \Rightarrow \text{False}$.

The following rules are logical versions of the rules FUN-FORM, FUN-INTRO, and FUN-ELIM. They can be derived using the equivalence of judgements $x^m : S^m \vdash T$ and $x^m : S^m \vdash P \in T$ discussed before.

$$\frac{H \vdash A \in \mathbb{U}_i \quad H, x : A \vdash B \in \mathbb{U}_i}{H \vdash (\forall x : A. B) \in \mathbb{U}_i} \quad (\text{ALL-FORM})$$

$$\frac{H \vdash S \in \mathbb{U}_i \quad H, x : S \vdash T}{H \vdash \forall x : S. T} \quad (\text{ALL-INTRO})$$

$$\frac{H \vdash N \in S \quad H \vdash \forall x : S. T}{H \vdash T[N/x]} \quad (\text{ALL-ELIM})$$

7.3.3 Classical Extension

The translation described in the next section makes use of Nuprl's operator

$$\uparrow b = \text{if } b \text{ then True else False}$$

which converts an element of \mathbb{B} into a (propositional) type.

The following properties can be formally verified inside Nuprl:

1. $\vdash \forall b \in \mathbf{bool}. \uparrow b \in \mathbb{P}_1$
2. $\vdash \forall b \in \mathbf{bool}. \uparrow b \Rightarrow (b = \mathbf{tt} \in \mathbb{B})$
3. $\vdash \forall b \in \mathbf{bool}. (b = \mathbf{tt} \in \mathbb{B}) \Rightarrow \uparrow b$

Hence, in pure Nuprl (with only definitional extensions) we can cast a boolean value into a proposition, but not vice versa. However, we need conversions in both directions to make use of the boolean logic in propositional proofs. Indeed, what is still missing in the boolean logic is a boolean equality.

For the translation of HOL's equality we wish to define a boolean polymorphic equality using Nuprl's propositional equality, but so far we do not have any means for converting a proposition into a boolean, which amounts to deciding whether a propositional type is inhabited. So we add a standard constant `inhabited` and we assume the following family of axioms stating that `inhabited(T)` decides if its argument, a type T in \mathbb{U}_i , is inhabited, and that it returns an element of T if this is the case

$$\vdash \mathbf{inhabited} \in x : \mathbb{U}_i \rightarrow x + (x \rightarrow \mathbf{Void})$$

Using the new axioms we can easily define an operator $\downarrow_b(-)$ casting a propositional type into a boolean value deciding the proposition:

$$\downarrow_b P = \mathbf{decide}(\mathbf{inhabited}(P), (x.\mathbf{tt}), (x.\mathbf{ff}))$$

where `decide` performs case analysis for elements of a disjoint union type.

The following *casting theorems* have been verified *inside* Nuprl:

1. $\vdash \forall P \in \mathbb{P}_i. \downarrow_b P \in \mathbb{B}$
2. $\vdash \forall P \in \mathbb{P}_i. (\uparrow \downarrow_b P) \Rightarrow P$
3. $\vdash \forall P \in \mathbb{P}_i. P \Rightarrow (\uparrow \downarrow_b P)$

7.3.4 Theories

An (*axiomatic*) Nuprl theory (Σ, Γ) consists of a signature Σ together with a set Γ of sentences over Σ called *axioms*. The set of all such Γ over a signature Σ is denoted by $\mathit{NuprlAxSet}(\Sigma)$. A signature morphism $H : \Sigma \rightarrow \Sigma'$ is said to be a *theory morphism* $H : (\Sigma, \Gamma) \rightarrow (\Sigma', \Gamma')$ iff $\Gamma' \vdash_{\Sigma'}^{\mathit{Nuprl}} \mathit{NuprlSen}(H)(\phi)$ for all $\phi \in \Gamma$. This gives a category of theories that will be denoted by **NuprlTh**.

A theory morphism $H : (\Sigma, \Gamma) \rightarrow (\Sigma', \Gamma')$ is called *axiom-preserving* iff $NuprlSen(H)(\Gamma) \subseteq \Gamma'$. The subcategory of **NuprlTh** with the objects of **NuprlTh** but morphisms restricted to axiom-preserving morphisms is denoted by **NuprlTh₀**. This will be the target category of the translation between HOL and Nuprl theories. **NuprlTh**, on the other hand, is used for the second stage of the HOL/Nuprl connection, namely, the interpretation of Nuprl theories.

7.4 Theory Translation

In accordance with the general logics terminology introduced in [Mes89a], which is used as a guideline for the following definitions, the translation from HOL theories to Nuprl theories will be specified by a functor

$$\Phi : \mathbf{HolFullSign} \rightarrow \mathbf{NuprlTh}_0$$

which translates HOL signatures into Nuprl theories and a natural transformation

$$\alpha : HolSen \rightarrow NuprlSen \circ \Phi$$

where

$$HolSen : \mathbf{HolFullSign} \rightarrow \mathbf{Cat}$$

$$NuprlSen : \mathbf{NuprlTh}_0 \rightarrow \mathbf{Cat}$$

which translates HOL sentences into Nuprl sentences.

In order to focus on the essence of the translation we will not deal with the technicalities of translating names and avoiding name clashes. Instead, we assume that all HOL (type) constants and (type) variables are nonstandard constants and variables, respectively, in Nuprl. We furthermore assume that there is a countably infinite set of reserved variables in Nuprl that are not used to represent HOL (type) variables.

First we define the functor

$$\Phi : \mathbf{HolFullSign} \rightarrow \mathbf{NuprlTh}_0$$

$$\Phi(\Sigma) = (\Phi(\Sigma), \Psi(\Sigma))$$

with a functor Φ and a (dependent) function Ψ (to be defined)

$$\Phi : \mathbf{HolFullSign} \rightarrow \mathbf{NuprlSign}$$

$$\Psi : \prod \Sigma : \mathbf{HolFullSign} . NuprlAxSet(\Phi(\Sigma))$$

where

$$NuprlAxSet : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$$

Applying Φ to a full HOL signature $\Sigma = (\Omega \vdash \Delta)$ yields a Nuprl theory (Σ', Γ') consisting of the signature part Σ' and the axiom part Γ' .

Here, and in the following, we do not define the behavior of functors on morphisms explicitly. Actually, there is only one natural and straightforward way to extend our definitions to morphisms.

We deal with the two levels Ω and Δ separately. First, we deal with the type signature Ω , which contains type constants to be translated into a Nuprl signature part $\Phi(\Omega)$ and an axiom part $\Psi(\Omega)$, where Φ is a functor and Ψ is a (dependent) function:

$$\begin{aligned} \Phi &: \mathbf{HolTypeSign} \rightarrow \mathbf{NuprlSign} \\ \Psi &: \prod \Omega : \mathbf{HolTypeSign} . \mathbf{NuprlAxSet}(\Phi(\Omega)) \end{aligned}$$

where

$$\mathbf{NuprlAxSet} : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$$

1. $\Phi(\{\nu : \mathbf{Type}^n \rightarrow \mathbf{Type}\}) = \{\nu\}$,
2. $\Phi(\emptyset) = \emptyset$,
3. $\Phi(\Omega_1 \cup \Omega_2) = \Phi(\Omega_1) \cup \Phi(\Omega_2)$,
4. $\Psi(\{\nu : \mathbf{Type}^n \rightarrow \mathbf{Type}\}) = \{\alpha^n : \mathbf{S} \vdash \nu(\alpha^n) \in \mathbf{S}\}$,
5. $\Psi(\emptyset) = \emptyset$,
6. $\Psi(\Omega_1 \cup \Omega_2) = \Psi(\Omega_1) \cup \Psi(\Omega_2)$.

Since semantically all HOL types are assumed to be nonempty, we have employed $x : \mathbf{S}$ as a suggestive notation for $x : \mathbb{U}_1$, $\mathit{inh}_x : \mathbf{Inh}(x)$ where inh_x is a reserved variable that is associated to x only and $\mathbf{Inh}(x)$ is defined as $\exists z : x \mathbf{True}$. Likewise, $x \in \mathbf{S}$ is a suggestive notation for $x \in \mathbb{U}_1 \wedge \mathbf{Inh}(x)$.

We also need to translate HOL type contexts $\alpha^n : \mathbf{Type}^n$ into Nuprl contexts giving a functor:

$$\begin{aligned} \Phi &: \mathbf{HolTypeCtxt} \rightarrow \mathbf{NuprlCtxt} \\ \Phi(\alpha^n : \mathbf{Type}^n) &= \alpha^n : \mathbf{S}^n \end{aligned}$$

This functor can be lifted to full type contexts:

$$\begin{aligned} \Phi &: \mathbf{HolFullTypeCtxt} \rightarrow \mathbf{NuprlFullCtxt} \\ \Phi(\Omega \vdash \alpha^n : \mathbf{Type}^n) &= \Phi(\Omega) \vdash \Phi(\alpha^n : \mathbf{Type}^n) \end{aligned}$$

To specify the translation Ψ of the signature Δ over Ω which contains constant declarations over Ω we need the following natural transformation

$$\alpha : \mathbf{HolType} \rightarrow \mathbf{NuprlTrm} \circ \Phi,$$

where:

$$\begin{aligned} \mathit{HolType} &: \mathbf{HolFullTypeCtxt} \rightarrow \mathbf{Set} \\ \Phi &: \mathbf{HolFullTypeCtxt} \rightarrow \mathbf{NuprlFullCtxt} \\ \mathit{NuprlTrm} &: \mathbf{NuprlFullCtxt} \rightarrow \mathbf{Set} \end{aligned}$$

translating HOL types into Nuprl types, which are terms in Nuprl:

1. $\alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\alpha_i^n) = \alpha_i^n$,
2. $\alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\mathbf{o}) = \mathbb{B}$,
3. $\alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\sigma \rightarrow \tau) = \alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\sigma) \rightarrow \alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\tau)$,
4. $\alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\nu(\sigma_1, \dots, \sigma_n)) = \nu(\alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\sigma_1), \dots, \alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\sigma_n))$.

Notice that the above HOL type \mathbf{o} of propositions is translated classically as the Nuprl data type \mathbb{B} . Formally, the use of \mathbb{B} , which enjoys the law of excluded middle, is unnecessarily strong for the interpretation of \mathbf{o} . However, since all HOL mathematics is developed in an extension of the classical logical theory \mathbf{bool} , this choice is appropriate.

Next, we give the translation of constants Δ over a type signature Ω . Again, we have a signature part $\alpha(\Omega)(\Delta)$ and an axiom part $\beta(\Omega)(\Delta)$, where α is a natural transformation and β is a type-signature-indexed family of (dependent) functions:

$$\begin{aligned} \alpha &: \mathit{HolSign} \rightarrow \mathbf{NuprlSign} \\ \beta(\Omega) &: \prod \Delta : \mathit{HolSign}(\Omega) . \mathit{NuprlAxSet}(\alpha(\Omega)(\Delta)) \end{aligned}$$

where

$$\begin{aligned} \mathit{HolSign} &: \mathbf{HolTypeSign} \rightarrow \mathbf{Cat} \\ \mathbf{NuprlSign} &: \mathbf{HolTypeSign} \rightarrow \mathbf{Cat} \\ \mathit{NuprlAxSet} &: \mathbf{NuprlSign} \rightarrow \mathbf{Cat} \end{aligned}$$

1. $\alpha(\Omega)(\{\alpha^n : \mathbf{Type}^n \vdash c : \sigma\}) = \{c\}$,
2. $\alpha(\Omega)(\emptyset) = \emptyset$,
3. $\alpha(\Omega)(\Delta_1 \cup \Delta_2) = \alpha(\Omega)(\Delta_1) \cup \alpha(\Omega)(\Delta_2)$,
4. $\beta(\Omega)(\{\alpha^n : \mathbf{Type}^n \vdash c : \sigma\}) = \{\alpha^n : \mathbf{S}^n \vdash c(\alpha^n) \in \alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\sigma)\}$,
5. $\beta(\Omega)(\emptyset) = \emptyset$,

$$6. \beta(\Omega)(\Delta_1 \cup \Delta_2) = \beta(\Omega)(\Delta_1) \cup \beta(\Omega)(\Delta_2).$$

Notice that we translate a polymorphic HOL constant into a Nuprl operator c representing a family of constants indexed by types.

The previous definitions are lifted to full signatures as follows:

$$\Phi : \mathbf{HolFullSign} \rightarrow \mathbf{NuprlSign}$$

$$\Psi : \prod \Sigma : \mathbf{HolFullSign} . \mathit{NuprlAxSet}(\Phi(\Sigma))$$

1. $\Phi(\Omega \vdash \Delta) = \Phi(\Omega) \cup \alpha(\Omega)(\Delta)$
2. $\Psi(\Omega \vdash \Delta) = \Psi(\Omega) \cup \beta(\Omega)(\Delta)$

Next, we define the translation of contexts $x^m : \sigma^m$, that depends on type contexts and gives therefore rise to a natural transformation:

$$\alpha : \mathit{HolCtxt} \rightarrow \mathit{NuprlCtxt} \circ \Phi$$

$$\mathit{HolCtxt} : \mathbf{HolFullTypeCtxt} \rightarrow \mathbf{Cat}$$

$$\Phi : \mathbf{HolFullTypeCtxt} \rightarrow \mathbf{NuprlFullCtxt}$$

$$\mathit{NuprlCtxt} : \mathbf{NuprlFullCtxt} \rightarrow \mathbf{Cat}$$

$$\alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(x^m : \sigma^m) = x^m : \alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\sigma^m)$$

We now lift the natural transformation α to a functor Φ between full contexts:

$$\Phi : \mathbf{HolFullCtxt} \rightarrow \mathbf{NuprlFullCtxt}$$

$$\Phi(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m) =$$

$$\Phi(\Omega \vdash \Delta) \vdash \Phi(\alpha^n : \mathbf{Type}^n), \alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(x^m : \sigma^m)$$

Our goal is the translation of HOL sentences into Nuprl sentences. Since HOL sentences are constructed from terms we define their translation first:

$$\alpha : \mathit{HolTrm} \rightarrow \mathit{NuprlTrm} \circ \Phi$$

where

$$\mathit{HolTrm} : \mathbf{HolFullCtxt} \rightarrow \mathbf{Set}$$

$$\Phi : \mathbf{HolFullCtxt} \rightarrow \mathbf{NuprlFullCtxt}$$

$$\mathit{NuprlTrm} : \mathbf{NuprlFullCtxt} \rightarrow \mathbf{Set}$$

1. $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(x_i^m) = x_i^m,$

2. $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(=_{\tau \rightarrow \tau \rightarrow \circ}) = (=_{\mathbf{b}}^{\alpha(\Omega)(\tau)})$,
3. $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(\Rightarrow_{\circ \rightarrow \circ \rightarrow \circ}) = (\Rightarrow_{\mathbf{b}})$,
4. $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(c_{\tau[\tau^k/\beta^k]}) =$
 $c(\alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\tau_1^k), \dots, \alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(\tau_k^k))$,
 where $(\beta^k : \mathbf{Type}^k \vdash c : \tau) \in \Delta$ and $\alpha^n : \mathbf{Type}^n \vdash \tau_i^k$ are types over Ω ,
5. $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(M N) =$
 $(\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(M)$
 $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(N))$,
6. $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(\lambda x : \sigma . M) =$
 $(\lambda x . \alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m, x : \sigma)(M))$

where we have employed the abbreviations

1. $\Rightarrow_{\mathbf{b}}$ is the boolean implication defined as
 $\Rightarrow_{\mathbf{b}} = \lambda x, y . \text{if } x \text{ then } y \text{ else } \mathbf{tt}$,
2. $=_{\mathbf{b}}^T$ is the polymorphic equality defined as
 $=_{\mathbf{b}}^T = \lambda x, y . \downarrow_b(x = y \in T)$.

In summary, there are three things happening in α above:

1. The basic HOL logical operators $=$ and \Rightarrow are translated to the corresponding Nuprl operators of the boolean data type \mathbb{B} .
2. We forget the HOL type annotations, since Nuprl is an untyped language, even though in practice it might be desirable to preserve this information (e.g. using the typed lambda operator of Nuprl).
3. A polymorphic HOL constant which has been translated into a type indexed family of Nuprl constants has to be instantiated appropriately each time it is used.

Now the natural transformation of HOL sentences

$$\alpha : \mathit{HolSen} \rightarrow \mathit{NuprlSen} \circ \Phi$$

with

$$\mathit{HolSen} : \mathbf{HolFullSign} \rightarrow \mathbf{Cat}$$

$$\mathit{NuprlSen} : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$$

$$\Phi : \mathbf{HolFullSign} \rightarrow \mathbf{NuprlSign}$$

is given by

$$\begin{aligned} \alpha(\Omega \vdash \Delta)(\alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m \vdash H \vdash A) = \\ \Phi(\alpha^n : \mathbf{Type}^n), \alpha(\Omega \vdash \alpha^n : \mathbf{Type}^n)(x^m : \sigma^m) \\ \vdash \alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(H) \\ \vdash \uparrow\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(A) \end{aligned}$$

with

1. $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)([]) = []$
2. $\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)([A_1, \dots, A_n]) =$
 $[h_1 : \uparrow\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(A_1), \dots,$
 $h_n : \uparrow\alpha(\Omega \vdash \Delta \vdash \alpha^n : \mathbf{Type}^n \vdash x^m : \sigma^m)(A_n)]$

Above h_1, \dots, h_n are reserved variables of Nuprl, that are distinct from HOL variables and are not used for any other purpose.

7.5 Correctness of the Translation

In the following we adapt some terminology from the theory of general logics [Mes89a]. To justify the translation we introduce an appropriate notion of map between entailment systems. To this end, we first instantiate the general notions from [Mes89a] that we recalled in Section 2.3 to the specific categories and functors we are interested in.

First we observe that the functor $\Phi : \mathbf{HolFullSign} \rightarrow \mathbf{NuprlTh}_0$ can be easily extended to a functor $\Phi : \mathbf{HolTh}_0 \rightarrow \mathbf{NuprlTh}_0$, called the α -extension to theories, by mapping an HOL-theory (Σ, Γ) to the Nuprl theory (Σ', Γ') with $\Sigma' = \Phi(\Sigma)$ and $\Gamma' = \Psi(\Gamma) \cup \alpha_\Sigma(\Gamma)$.

We also extend the functors $HolSen : \mathbf{HolFullSign} \rightarrow \mathbf{Cat}$ and $NuprlSen : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$ to functors $HolSen : \mathbf{HolTh}_0 \rightarrow \mathbf{Cat}$ and $NuprlSen : \mathbf{NuprlTh}_0 \rightarrow \mathbf{Cat}$ by just ignoring the axioms in the theories.

Correspondingly, we extend the natural transformation $\alpha : HolSen \rightarrow NuprlSen \circ \Phi$ with $HolSen : \mathbf{HolFullSign} \rightarrow \mathbf{Cat}$ and $NuprlSen : \mathbf{NuprlSign} \rightarrow \mathbf{Cat}$ to a natural transformation between $HolSen : \mathbf{HolTh}_0 \rightarrow \mathbf{Cat}$ and $NuprlSen \circ \Phi : \mathbf{HolTh}_0 \rightarrow \mathbf{Cat}$ by setting $\alpha_{(\Sigma, \Gamma)} = \alpha_\Sigma$.

Now $\Phi : \mathbf{HolTh}_0 \rightarrow \mathbf{NuprlTh}_0$ is an α -simple functor [Mes89a], i.e., it is the α -extension to theories of a functor from signatures to theories. However, since the axiomatic part in the target of the translation is essential, Φ is *not* α -plain.

Correctness Theorem

$(\Phi, \alpha) : (\mathbf{HolFullSign}, HolSen, \vdash^{Hol}) \rightarrow (\mathbf{NuprlSign}, NuprlSen, \vdash^{Nuprl})$ is a map between entailment systems, i.e., the following properties are satisfied:

1. $\alpha : \mathit{HolSen} \rightarrow \mathit{NuprlSen} \circ \Phi$ is a natural transformation, where

$$\begin{aligned} \mathit{HolSen} &: \mathbf{HolTh}_0 \rightarrow \mathbf{Cat}, \\ \Phi &: \mathbf{HolTh}_0 \rightarrow \mathbf{NuprlTh}_0, \\ \mathit{NuprlSen} &: \mathbf{NuprlTh}_0 \rightarrow \mathbf{Cat}, \end{aligned}$$
2. $\Phi : \mathbf{HolTh}_0 \rightarrow \mathbf{NuprlTh}_0$ is α -sensible (see below),
3. $\Gamma \vdash_{\Sigma}^{\mathit{Hol}} \phi \Rightarrow \alpha_{\Sigma}(\Gamma) \vdash_{\Phi(\Sigma)}^{\mathit{Nuprl}} \alpha_{\Sigma}(\phi)$ for each full HOL signature Σ .

Proof Sketch

Concerning (1), we have not made explicit the translation of morphisms. It should, however, be clear that there is only one natural choice and α is a natural transformation, since it is compatible with all renaming morphisms.

In condition (2) the functor $\Phi : \mathbf{HolTh}_0 \rightarrow \mathbf{NuprlTh}_0$ is α -sensible iff

1. There is a functor $\Phi^{\diamond} : \mathbf{HolFullSign} \rightarrow \mathbf{NuprlSign}$ such that $\mathit{NuprlSign} \circ \Phi = \Phi^{\diamond} \circ \mathit{HolFullSign}$, where

$$\begin{aligned} \mathit{HolFullSign} &: \mathbf{HolTh}_0 \rightarrow \mathbf{HolFullSign} \text{ and} \\ \mathit{NuprlSign} &: \mathbf{NuprlTh}_0 \rightarrow \mathbf{HolFullSign} \end{aligned}$$
 are the obvious forgetful functors.
2. $\Gamma^{\bullet} = (\emptyset' \cup \alpha_{\Sigma}(\Gamma))^{\bullet}$ whenever $\Phi(\Sigma, \emptyset) = (\Sigma', \emptyset')$ and $\Phi(\Sigma, \Gamma) = (\Sigma', \Gamma')$. Here we use the deductive closure w.r.t. \vdash^{Nuprl} , i.e., $\Gamma^{\bullet} = \{\phi : \Gamma \vdash^{\mathit{Nuprl}} \phi\}$.

This is clearly satisfied in our case (choosing $\Phi : \mathbf{HolFullSign} \rightarrow \mathbf{NuprlSign}$ for Φ^{\diamond}). Actually, the fact that Φ is simple is a stronger property (we have $\Gamma' = \emptyset' \cup \alpha_{\Sigma}(\Gamma)$) which shows that we do not need the full generality of maps between entailment systems here.

Condition (3) of the correctness theorem states that the entailment relation is preserved by the translation. Given an arbitrary full HOL signature Σ , the implication can be verified by showing that the translation of each HOL inference rule can be derived in Nuprl using the axioms $\Psi(\Sigma)$. Throughout the following we assume $\Psi(\Sigma)$ without explicitly mentioning it.

To minimize metalogical reasoning we can make use of a number of simple theorems that have been verified inside Nuprl:

```

*T assert_beq_1  ∀T:ℒ.  ∀x,y:T.  x = y ⇒ ↑(x =b y)
*T assert_beq_2  ∀T:ℒ.  ∀x,y:T.  ↑(x =b y) ⇒ x = y
*T REFL_lemma   ∀T:ℒ.  ∀t:T.  ↑(t =b t)
*T SUBST_lemma  ∀T:ℒ.  ∀a,b:T.  ∀p:T → ℒ.  ↑(a =b b) ⇒ (p a) ⇒ (p b)
*T DISCH_lemma  ∀p,q:ℒ.  (↑p ⇒ ↑q) ⇒ ↑(p ⇒b q)
*T MP_lemma     ∀t1,t2:ℒ.  ↑(t1 ⇒b t2) ⇒ ↑t1 ⇒ ↑t2

```

Most of the translated inference rules have surprisingly short proofs in Nuprl if we use the theorems above and the following two well-formedness lemmas. For better readability we write $\llbracket x \rrbracket$ for $\alpha(u)(x)$ where u is clear from the context.

Notice also that under the propositions-as-types interpretation the Nuprl rules FUN-FORM, FUN-INTRO, FUN-ELIM correspond to logical rules for universal quantification including implication as a special case. Below, we denote the logical versions of these rules without explicit extraction terms by ALL-FORM, ALL-INTRO and ALL-ELIM, respectively.

Lemma 7.5.1 (Well-formedness lemma for translated HOL-types)

For each HOL type $\alpha^n : \text{Type}^n \vdash \sigma$ we can derive
 $\vdash \forall \alpha^n : \mathbf{S}^n . \llbracket \sigma \rrbracket \in \mathbf{S}$ in Nuprl.

Lemma 7.5.2 (Well-formedness lemma for translated HOL-terms)

For each HOL term $\alpha^n : \text{Type}^n \vdash x^m : \sigma^m \vdash M$ of type τ we can derive
 $\vdash \forall \alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket . \llbracket M \rrbracket \in \llbracket \tau \rrbracket$ in Nuprl.

Both lemmas can be proved by induction over HOL types and HOL terms, respectively.

In the following we sketch the proof for each translated HOL rule separately. To avoid technicalities we omit routine steps and well-formedness conditions.

Lemma 7.5.3

$$\frac{}{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, h_1 : \uparrow \llbracket A \rrbracket \vdash \uparrow \llbracket A \rrbracket} \quad (\text{ASSUME}')$$

Proof Use HYP. □

Lemma 7.5.4

$$\frac{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket G \rrbracket \vdash \uparrow \llbracket A \rrbracket \quad \alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket H \rrbracket \vdash \uparrow (\llbracket A \rrbracket \Rightarrow_b \llbracket B \rrbracket)}{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket H, G \rrbracket \vdash \uparrow \llbracket B \rrbracket} \quad (\text{MP}')$$

Proof Backchaining via MP_lemma. □

Lemma 7.5.5

$$\frac{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket H, A, G \rrbracket \vdash \uparrow \llbracket B \rrbracket}{\alpha^n : \mathbf{S}^n, x^m : \sigma^m, \llbracket H, G \rrbracket \vdash \uparrow (\llbracket A \rrbracket \Rightarrow_b \llbracket B \rrbracket)} \quad (\text{DISCH}')$$

Proof Backchaining via DISCH_lemma. □

Lemma 7.5.6

$$\frac{}{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket \vdash \uparrow(\llbracket M \rrbracket =_{\mathbf{b}}^{\llbracket \rho \rrbracket} \llbracket M \rrbracket)} \quad (\text{REFL}')$$

Proof Backchaining via `REFL_lemma`. \square

Lemma 7.5.7

$$\frac{}{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket \vdash \uparrow(\llbracket ((\lambda y : \rho . M) N) \rrbracket =_{\mathbf{b}}^{\llbracket \tau \rrbracket} \llbracket M[N/y] \rrbracket)} \quad (\text{BETA-CONV}')$$

Proof

Clearly, N has type ρ , and M has type τ assuming $y : \rho$.

By the definition of $=_{\mathbf{b}}$ we have to prove

$$\frac{}{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket \vdash \llbracket ((\lambda y : \rho . M) N) \rrbracket = \llbracket M[N/y] \rrbracket \in \llbracket \tau \rrbracket}$$

Assume $\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket$. By the well-formedness lemmas we have $\llbracket N \rrbracket \in \llbracket \sigma \rrbracket$ and $y : \llbracket \rho \rrbracket \vdash \llbracket M \rrbracket \in \llbracket \tau \rrbracket$. Now we can apply `FUN-BETA` to obtain $((\lambda y . \llbracket M \rrbracket) \llbracket N \rrbracket) = \llbracket M \rrbracket \llbracket \llbracket N \rrbracket / y \rrbracket \in \llbracket \tau \rrbracket$. \square

Lemma 7.5.8

$$\frac{\begin{array}{l} \alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket H_i^k \rrbracket \vdash \uparrow(\llbracket M_i^k \rrbracket =_{\mathbf{b}}^{\llbracket \rho_i^k \rrbracket} \llbracket N_i^k \rrbracket) \text{ for all } i \\ \alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket G \rrbracket \vdash \uparrow \llbracket A[M^k/z^k] \rrbracket \end{array}}{\alpha^n : \mathbf{S}^n, x^m : \sigma^m, \llbracket H^k, G \rrbracket \vdash \uparrow \llbracket A[N^k/z^k] \rrbracket} \quad (\text{SUBST}')$$

Proof

By the definition of $=_{\mathbf{b}}$ we have to prove

$$\frac{\begin{array}{l} \alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket H_i^k \rrbracket \vdash \llbracket M_i^k \rrbracket = \llbracket N_i^k \rrbracket \in \llbracket \rho_i^k \rrbracket \text{ for all } i \\ \alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket G \rrbracket \vdash \uparrow \llbracket A[M^k/z^k] \rrbracket \end{array}}{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket H^k, G \rrbracket \vdash \uparrow \llbracket A[N^k/z^k] \rrbracket}$$

Assume $\alpha^n : \mathbf{S}^n, x^m : \sigma^m, \llbracket H^k, G \rrbracket$. Our goal is $\uparrow \llbracket A[N^k/z^k] \rrbracket$. Define $B_i = A[N_1^k, \dots, N_{i-1}^k, x, M_{i+1}^k, \dots, M_k^k/z_1^k, \dots, z_{i-1}^k, z_i^k, z_{i+1}^k, \dots, z_k^k]$ for $i \in \{1 \dots k\}$ with a fresh variable x . Notice that $A[M^k/z^k] = B_1[M_1^k/x]$, $A[N^k/z^k] = B_k[N_k^k/x]$, and $B_i[N_i^k/x] = B_{i+1}[M_{i+1}^k/x]$ for $i \in \{1 \dots k-1\}$.

The first premise gives $\llbracket M_i^k \rrbracket = \llbracket N_i^k \rrbracket \in \llbracket \rho_i^k \rrbracket$ for each i . Using `SUBST` for each $i \in \{1 \dots k\}$, from $\uparrow \llbracket B_i[M_i^k/x] \rrbracket$ we derive $\uparrow \llbracket B_i[N_i^k/x] \rrbracket$, which is equal to $\uparrow \llbracket B_{i+1}[M_{i+1}^k/x] \rrbracket$ if $i < k$.

Hence, combining these steps using `EQUALITY-TRANS`, from our second premise $\uparrow \llbracket A[M^k/z^k] \rrbracket$ which is equal to $\uparrow \llbracket B_1[M_1^k/x] \rrbracket$ we can derive $\uparrow \llbracket B_k[N_k^k/x] \rrbracket$, which is equal to $\uparrow \llbracket A[N^k/z^k] \rrbracket$. \square

Lemma 7.5.9

$$\frac{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, y : \llbracket \rho \rrbracket \vdash \uparrow(\llbracket M \rrbracket =_{\mathbf{b}}^{\llbracket \tau \rrbracket} \llbracket N \rrbracket)}{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket \vdash \uparrow(\llbracket \lambda y : \rho . M \rrbracket =_{\mathbf{b}}^{\llbracket \rho \rightarrow \tau \rrbracket} \llbracket \lambda y : \rho . N \rrbracket)} \quad (\text{ABS}')$$

Proof

By the definition of $=_{\mathbf{b}}$ we have to prove

$$\frac{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, y : \llbracket \rho \rrbracket \vdash \llbracket M \rrbracket = \llbracket N \rrbracket \in \llbracket \tau \rrbracket}{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket \vdash \llbracket \lambda y : \rho . M \rrbracket = \llbracket \lambda y : \rho . N \rrbracket \in \llbracket \rho \rightarrow \tau \rrbracket}$$

which follows from FUN-XI. \square

Lemma 7.5.10

$$\frac{\alpha^n : \mathbf{S}^n, x^m : \llbracket \sigma^m \rrbracket, \llbracket H \rrbracket \vdash \uparrow \llbracket A \rrbracket}{\beta^k : \mathbf{S}^k, x^m : \llbracket \sigma^m[\tau^n/\alpha^n] \rrbracket, \llbracket H \rrbracket \vdash \uparrow \llbracket A[\tau^n/\alpha^n] \rrbracket} \quad (\text{INST-TYPE}')$$

provided that we have types $\beta^k : \mathbf{Type}^k \vdash \tau_i^n$ and the α_i^n do not occur in H .

Proof

Using the fact that the types on the lefthand side of these sequents are well-formed, we can use ALL-INTRO and ALL-ELIM to reduce our goal to

$$\frac{\vdash \forall \alpha^n : \mathbf{S}^n . \forall x^m : \llbracket \sigma^m \rrbracket . \llbracket B_1 \rrbracket, \wedge \cdots \wedge, \llbracket B_l \rrbracket \Rightarrow \uparrow \llbracket A \rrbracket}{\vdash \forall \beta^k : \mathbf{S}^k . \forall x^m : \llbracket \sigma^m[\tau^n/\alpha^n] \rrbracket . \llbracket B_1 \rrbracket, \wedge \cdots \wedge, \llbracket B_l \rrbracket \Rightarrow \uparrow \llbracket A[\tau^n/\alpha^n] \rrbracket}$$

if H takes the form $\llbracket B_1, \dots, B_l \rrbracket$.

By the well-formedness lemma we have $\forall \beta^k : \mathbf{S}^k \llbracket \tau_i^n \rrbracket \in \mathbf{S}$. So we can finish the proof by using ALL-ELIM. \square

Even though metalogical reasoning can be minimized using the Nuprl theorems given before, the metalogical nature of these proofs is unavoidable. The proof translator to be discussed in Section 7.7.2 can be seen as a formalization of their computational content of these metalogical proofs using the metalanguage of the Nuprl system, which is ML.

7.6 Nuprl Theory Interpretations

Originally the Nuprl system was intended to deal with *definitional theories*, which are theories where all the axioms in Γ are definitions, i.e. equations defining the construct on the lefthand side. Since HOL theories are not required to be definitional, we need this more general notion of *axiomatic Nuprl theory* which has been introduced before.

Typically, we are not interested in the full generality of using an axiomatic Nuprl theory which has been obtained by translation from an axiomatic HOL theory.

Instead, we may prefer to *instantiate constants* with fixed meanings. One reason may be that we want them to be *computationally meaningful*.⁹ This (partial) instantiation of constants leads from one axiomatic Nuprl theory to another one, typically with a more restricted class of models. Usually, some or all of the original axioms become theorems in the new theory. Ideally, we obtain a definitional theory, which means that we have constructed an *internal model* of an axiomatic Nuprl theory inside Nuprl itself.

Given Nuprl theories (Σ, Γ) and (Σ', Γ') , we say that (Σ, Γ) is *interpreted* in (Σ', Γ') by I iff $I : (\Sigma, \Gamma) \rightarrow (\Sigma', \Gamma')$ is a theory morphism in the category **NuprlTh**. Notice that these morphisms are not necessarily axiom-preserving, since it is typically the point of such an interpretation to get rid of axioms.

Since we have to verify $\Gamma' \vdash_{\Sigma'}^{Nuprl} NuprlSen(I)(\phi)$ for all $\phi \in \Gamma$, the sentences $NuprlSen(I)(\phi)$ are called *proof obligations*. As explained in the introduction of this chapter, the activity of setting up a theory morphism and verifying the proof obligations characterizes the second stage of the HOL/Nuprl connection which requires user interaction in general. Subsequently, we interpret the translation of the logical theory of HOL. In fact, we will see that this can be done, thanks to the classical axiom scheme for **inhabited**, in a purely definitional way.

7.6.1 Interpreting the Logical Theory

The logical HOL theory **bool** is a theory like every other theory, and does not need any particular consideration. It is however important to notice that **bool** is not a pure definitional theory in HOL. Still, it does *not* require any external metalogical treatment, since all its axioms can be verified in a rigorous formal way *inside* the classical Nuprl system. We follow [How99] where the proof obligations have been verified using the interpretation given next.

Each HOL constant c will be interpreted by a Nuprl constant $\llbracket c \rrbracket$. We assume the following definitional axioms:

1. $\llbracket \neg \rrbracket = \lambda x . \text{if } x \text{ then ff else tt}$
2. $\llbracket \wedge \rrbracket = \lambda x, y . \text{if } x \text{ then } y \text{ else ff}$
3. $\llbracket \vee \rrbracket = \lambda x, y . \text{if } x \text{ then tt else } y$
4. $\llbracket \forall \rrbracket = \lambda T . \lambda P . \downarrow_b(\forall x : T . \uparrow P)$
5. $\llbracket \exists \rrbracket = \lambda T . \lambda P . \downarrow_b(\exists x : T . \uparrow P)$

⁹For instance, the fact that a new data type is introduced in HOL by a conservative theory extensions that characterizes the data type only up to isomorphism gives us some freedom for its interpretation in Nuprl.

As an abbreviation we use $\text{arb}(T)$ which picks an arbitrary element inhabiting a nonempty type T :

$$\text{arb}(T) = \text{decide}(\text{inhabited}(T), x.x, x.\bullet)$$

Hilbert's choice operator ($\llbracket \epsilon \rrbracket T P$), where P is a boolean predicate on T , picks an element of the subset of T specified by P if this subset is nonempty, or yields an arbitrary element of T otherwise.

$$\llbracket \epsilon \rrbracket = \lambda T . \lambda P . \text{decide}(\text{inhabited}(\{x : T \mid \uparrow(P x)\}), x.x, x.\text{arb}(T))$$

Finally, the HOL data type `ind` is interpreted by $\llbracket \text{ind} \rrbracket = \mathbb{N}$, where \mathbb{N} is the Nuprl data type of natural numbers. Since we only need to prove the translation of the axiom `INFINITY-AX`, any other infinite type would do.

Using the interpretation $\llbracket _ \rrbracket$, all the proof obligations, i.e. the translated axioms of the HOL theory `bool`, can be discharged and only the definitional Nuprl axioms given above will remain. The fact that HOL types are translated to nonempty Nuprl types is essential to verify the proof obligation corresponding to the declaration of ϵ .

7.7 Practical Work

As applications of our results we briefly discuss two lines of practical work that has been conducted in the context of the research presented in this chapter: First, we discuss an executable formal specification developed by the author of this thesis which explores the practical capabilities of Maude to build logic translators on the basis of their formal specification [CDE⁺99b, MS99], and second we report on some experience with the development of a proof translator, which has been implemented by Pavel Naumov on top of the Nuprl proof development system and is described in more detail in the joint paper [NSM01].

7.7.1 An Executable Formal Specification

The theory of general logics together with an executable logical framework such as rewriting logic and its implementation in Maude provides as a general methodology [MOM94] to specify logics in a formally rigorous way. An application of the general methodology in the context of type theory was studied in Chapter 6, where rewriting logic was employed as an executable logical framework to represent deductive systems of type theories. Since rewriting logic cannot only be used as an executable logical framework, but also as an executable metalogic, we can go one step further and formally specify not only the logics involved, but also the maps between them.

To obtain a formal executable specification of the translation between theories described in Section 7.4 we employed the specification language Maude [CDE⁺99a, CDE⁺00b]. For practical reasons, we have defined a more refined version of this translation, which has a similar functionality as Howe's original translator [How99], in *membership equational logic* [BJM97], the equational sublogic of *rewriting logic* that is implemented in Maude. In contrast to [How99] the translator is specified in a purely equational way as a single function that maps axiomatic HOL theories into axiomatic Nuprl theories, which we think is a more suitable basis for formal metalogical analysis. The specification is more refined than the abstract presentation in this chapter, since it allows renaming, it can cope with hierarchically structured HOL theories, and it has to deal with several technicalities concerning the interfaces to HOL and Nuprl. Last but not least, the specification can be executed using the Maude rewriting engine with reasonable efficiency, and has indeed been tested by translating the core library of HOL into Nuprl. In summary, the formal specification of the HOL/Nuprl connection nicely demonstrates this use of Maude as an executable metalogic (cf. [CDE⁺99b]), which also supports practical necessities like parsing external languages (HOL theories in this case) in a very uniform environment.

In the specification of the translator we have employed membership equational logic as a metalogic to specify the translation between theories of these logics. A consequence of the fact that the metalogic is executable is that the specification can be used as an actual translator. Another important application of a metalogic is formal theorem proving. Ideally, reasoning about logics and their relationships and efficient execution should be unified in a single framework so that an interactive theorem prover can exploit the executability to achieve a larger degree of automation. As already explained in Chapter 6, this could be done using rewriting logic itself as a reflective metalogical framework [BCM99], or alternatively in a more expressive logic which contains rewriting logic as an executable sublogic such as the *open calculus of constructions* (OCC) presented in Chapter 8. An additional advantage that would be gained by the use of OCC is the possibility to formally specify the translation in a way which is not only executable but also typed in a very strong sense. Thanks to the dependent types available in OCC, this could be done by following exactly the strong typing discipline based on syntactic categories and multilevel natural transformations that we used in our mathematical treatment of the HOL/Nuprl connection as suggested by the theory of general logics. Although we have not yet explored this particular application of OCC in the context of general logics, we will come back in Chapter 8 to the more general issue of formalizing categorical concepts in a computationally useful way.

7.7.2 Towards a Proof Translator

The translation described in this chapter is a translation between theories, i.e., we translate a HOL sentence A over Σ into a Nuprl sentence $\alpha(A)$ over $\Phi(\Sigma)$. The proof-theoretic correctness result ensures that if A is provable from Γ in HOL then $\alpha(A)$ is provable from $\Psi(\Sigma) \cup \alpha_\Sigma(\Gamma)$ in Nuprl. The proof of the correctness theorem is carried out by constructing a Nuprl proof of $\alpha(A)$ for each HOL proof of A . This suggests extending the translation between theories to a translation between proof calculi [Mes89a], so that this proof translation is a computable function between the data types of proofs. Indeed, this idea turned out to be feasible in practice as demonstrated by the implementation of an HOL/Nuprl proof translator [NSM01] by Pavel Naumov that is briefly described in this section.

The proof translator works together with HOL90.10 and has been implemented in ML on top of the Nuprl system. This facilitates the translation of HOL proofs into Nuprl proofs considerably, since the powerful collection of Nuprl tactics can be employed. Most of the inference rules of the HOL logic can be handled by a simple combination of Nuprl tactics and the theorems given earlier in a very direct way. Routine proof obligations such as well-typedness conditions are discharged automatically by Nuprl.

In spite of the clear theoretical picture some difficulties have been encountered in the implementation that are worth mentioning:

1. The first difficulty is of a technical nature, namely, that the HOL system is designed in such a way that proof objects are not explicitly present. Consequently, it was necessary to equip the system with an explicit notion of proof object that can be extracted from each theorem.¹⁰
2. Another problem is caused by the somewhat unsatisfactory situation that the HOL system does not only use the proof rules of the HOL logic as presented in the HOL documentation, but it uses in addition a considerable number of derived rules that are not reduced to basic proof rules. It seems that efficiency considerations are the reason for this “optimization,” but for the proof translator this means that either:
 - (a) the derived rules have to be eliminated *inside* HOL by reducing them to basic inference rules, or
 - (b) that the derived rules must be given the same status as basic inference rules, which increases the number of rules that have to be treated *outside* HOL by the proof translator.

¹⁰Technically, this has been done on top of a proof recording mechanism developed in [Won99] for the purpose of proof validation.

We decided that in the long term the proof translator should follow alternative (b), since it turned out that, as in the case of the basic inference rules, most of the derived rules mentioned above can be implemented in Nuprl with little effort. At present, the translator supports nearly all basic proof rules and a number of derived rules.

3. Finally, the proof translator can also deal with another particularity: The HOL implementation of the MP rule also applies to a premise $\neg M$ presupposing that $\neg M$ is equal to $M \Rightarrow \text{F}$. In other words, the definitional axiom of the logical theory is built into the inference rules.

So far the proof translator has been applied to a number of simple HOL theorems of logical nature, namely, to all theorems contained in the theorem library `taut_thms.sml` of the HOL distribution. Although these theorems are quite simple, this is definitely not true for the extracted HOL proofs. The translation of the proof of `DE_MORGAN_THM`, which was automatically generated by HOL, revealed that the proof objects generated by HOL can be surprisingly large and confusing (in this case the HOL proof contained 4733 HOL inference rules). We conjecture that this problem occurs only if proofs are generated by certain automatic brute-force tactics, and that it can be circumvented by optimizing these tactics in HOL or by proving such theorems using more efficient tactics in Nuprl. Nevertheless these proofs were good test cases for the capabilities of the proof translator and also for the capabilities of Nuprl to handle large proof objects. For a detailed description of the proof translator and further experiments we refer to [NSM01]. This reference also compares our approach with the implementation of an HOL/Coq proof translator presented in [Den00].

7.8 Final Remarks

We have complemented Howe's semantics-based justification of the HOL/Nuprl connection with a proof-theoretic counterpart. Our result showing that the translation can be seen as a map between the entailment systems of HOL and Nuprl does not only provide a simple proof-theoretic justification for *translating theories*, but it does in addition suggest *proof translation* as an interesting application that has not been considered in the context of the HOL/Nuprl connection so far. The feasibility of proof translation has been demonstrated by the proof translator developed by Pavel Naumov. Experiments with a number of simple theorems indicate that proof translation is practical even though the proof objects can be rather complex. As a consequence we think that scaling up the approach to complex libraries of formal mathematics is a realistic goal, provided that proof objects are not flat proofs but *structured proofs* that reflect the modularity of the development in terms of intermediate lemmas.

From a more general point of view, the HOL/Nuprl connection shows how *classical extensional reasoning* is possible in a logical type theory such as Nuprl *without giving up the intensionality of the propositions-as-types logic*. In this way hybrid developments using both intensional and extensional logic become possible as demonstrated by [FH97]. It is especially noteworthy that the HOL/Nuprl connection does not depend on features of Nuprl that go beyond Martin-Löf's type theory (in particular subset types are not really necessary) and furthermore does not rely on the fact that these type theories are polymorphic. As a consequence, a similar connection can be established for type theories such as the extended calculus of constructions (cf. Section 2.5) or the open calculus of constructions (cf. Chapter 8). On the other hand, this approach is incompatible with the inductive calculus of constructions if we use the boolean data type in the impredicative `Set` universe. This follows from the result that the assumption of a casting function between `Prop` and a boolean type `bool` in `Set` leads to an inconsistency [Geu01]. The approach to classical logic that we employ in Chapter 4 essentially axiomatizes `Prop` as a boolean data type, giving up its intensional character.

7.9 Acknowledgements

This chapter emerged from a fruitful collaboration between Pavel Naumov and myself under the guidance of José Meseguer. I gratefully acknowledge support for the work conducted at SRI by DARPA and NASA (Contract NAS2-98073), by Office of Naval Research (Contract N00014-96-C-0114), by NSF Grant (CCR-9633363), and by a DAAD grant in the scope of HSP-III. I would like to thank both, José Meseguer for his proposal to investigate this subject, his advice, and his detailed suggestions, and Pavel Naumov, who not only implemented our ideas in the Nuprl system but also helped on the theoretical side to close the gap between theory and practice. I furthermore enjoyed several discussions with Robert Constable and Stuart Allen on the particularities of Nuprl and the issue of logic translation and I am very grateful for their time during my visits at Cornell. I would also like to thank Doug Howe, since this project draws on his earlier work and would not have been possible without his support. He made not only the source code of the HOL/Nuprl connection available to us, but he also invited me to Lucent/Bell Labs, giving me the opportunity to discuss with him our initial ideas on a proof-theoretic correctness result. Last but not least, I appreciate all the useful suggestions by Narciso Martí-Oliet regarding this chapter, and I have to apologize that they are reflected only partially due to lack of time.

Chapter 8

Towards a Unified Language:

The Open Calculus of Constructions

In this chapter we develop and study a language based on ideas from rewriting logic, its membership equational sublogic, and type theory, more precisely an extension of the calculus of constructions with Martin-Löf-style universes. The result is an equational and rewriting-based programming and specification language with dependent types, a language that we call the *open calculus of constructions* (OCC). After giving some important motivation for the interest in this language, which quite strongly deviates from the prevailing lines of research in type theory, we present the formal system which we justify via a very intuitive set-theoretic semantics. Furthermore, we use a prototype of the type theory to illustrate the pragmatics and the practical significance of our approach by means of a considerable number of examples. To this end we frequently refer to previous chapters of this thesis, since applications of rewriting logic, membership equational logic, the calculus of inductive constructions (CIC), and other techniques, such as classical reasoning using type theory, give also rise to interesting and important applications of OCC.

To motivate our work we begin with a brief review of the two languages involved: rewriting logic and the extended calculus of constructions. Each of these languages is a typical representative of an important paradigm, which is the first-order paradigm of equational/rewriting specification in the former case and the higher-order paradigm of type theory in the later.

Rewriting logic (RWL) and its underlying membership equational logic (MEL) favors the use of abstract executable specifications and has a flexible notion of computation based on conditional rewriting modulo equations. Membership equational logic suggests a very liberal notion of inductive definitions, given by the initial or, in the case of parameterization, by the more general free semantics of equational theories, which makes rewriting logic powerful enough to serve as a semantic and logical framework [Mes96, MOM94, MOM96, MMO95, Mes98, Mes92] to represent a wide variety of formalisms in a very direct way.

The calculus of constructions (CC) [CH85, CH88, Coq85], on the other hand, provides higher-order functions and dependent types, but it is based on a fixed notion of computation, namely β -reduction. Already the subsystem F of CC is powerful enough for a computational encoding of free algebraic datatypes [BB85], but even for this restricted class the encoding is inconvenient, potentially inefficient, and does not provide induction principles [CPM90]. In essence, this unsatisfactory situation has led to an integration with a monomorphic Martin-Löf-style type theory, which has supported a rich collection of different datatypes from the very beginning. This development has involved some intermediate systems that are important to mention, because they greatly inspired the design of OCC. After the extension of CC by a predicative, cumulative universe hierarchy in the generalized calculus of constructions (GCC) [Coq86], an important improvement was the introduction of the extended calculus of constructions (ECC) [Luo91, Luo94], which thanks to a fully cumulative universe hierarchy has a very

simple and intuitive type checking algorithm based on a notion of principal types. Using ECC as a core formalism,¹ two different approaches to support free inductive data type definitions have been developed: the uniform theory of dependent types (UTT) [Luo92, Luo94] and the calculus of inductive constructions (CIC) [CPM90]. In both approaches the inductive definition of a type extends the underlying computation system by rules for higher-order primitive recursion over the newly introduced type. A combination of inductive types with pattern matching and restricted fixpoint operators, as implemented in the most recent version of COQ [BBC⁺99], is a considerable improvement, but the main problem remains, namely that the user is forced to define functions in terms of syntactically restricted schemes. These and other disadvantages are taken in [BJO99] to motivate the addition of algebraic inductive definitions to CC, allowing the definition of functions using (unconditional) rules following a general schema in the spirit of abstract data type systems [JO97] while preserving operational properties such as confluence and strong normalization. In addition to the restrictions on function definitions over inductive types imposed by all these approaches, another drawback is that they all confine the notion of inductive types to free inductive types, which are freely generated by their constructors, as opposed to the notion of equational inductive definitions provided by equational specification languages such as membership equational logic (MEL). The idea of addressing some of these limitations using some combination of MEL with CC has already been suggested as a long-term research objective in [Jou98].

Apart from these difficulties in present extensions of CC we see another apparently more serious problem caused by a too intensional notion of computation, that to our knowledge has not been widely discussed in the literature so far. In standard approaches to logical type theories we have the situation that type checking and in particular proof checking take place in a *closed computational world*, which makes it not only potentially inefficient [BJO99] but more importantly too restrictive to deal satisfactorily with *abstract logical specifications* leading to problems with *modularity* and *formal interoperability*. Logical type theories such as UTT and CIC require a module to be implemented, i.e. realized conservatively, in the type theory in order to be operationally useful, which in many nontrivial cases is needed to typecheck client modules. In practice such an implementation may not be available for various reasons, for instance because it is not accessible for the client (as in the case of a modular development where private parts of modules are hidden), or because it does not exist, since it is realized/justified externally (as in the case of imported theorems from other theorem provers).

In the following we elaborate on our view that only an approach which establishes a link between the deductive system of the logic and the computational system

¹We disregard another feature of ECC at this point, namely strong Σ -types, which are essentially subsumed by inductive types.

of the type theory can solve this problem. Indeed, all extensions of CC we are aware of maintain a strong barrier between the logical and the computational system. Although computational equality implies logical equality in CC [Luo94] (e.g. if we use Leibnitz equality) the converse is not true in general. Although in the empty context we have the result that computational equality and logical equality coincide, in practice we are usually working with nonempty contexts for various reasons and logical equality (which is undecidable) is typically coarser than computational equality (which is decidable). Other type theories such as Martin-Löf's polymorphic, extensional type theory and the type theory of Nuprl use a judgemental equality which coincides with logical equality, but cannot be explained in computational terms, which is why typechecking becomes as hard as theorem proving in these type theories.

More generally, we adopt the view that abstract logical specifications should be well supported even if they are not implementable by means of the calculus, i.e. as conservative extensions. In other words, logical and computational conservativity should not be enforced, since there are many important ways to justify theorems, axioms and computation rules beyond using the limited notion of proof provided by the calculus itself. Relative conservativity between abstract specifications (open world view) is practically more important than absolute conservativity inside the calculus (closed world view).

To realize this idea we reconsider the underlying philosophy of executable specification languages like equational logics. Logical axioms and theorems should be equipped with a clear operational meaning that has a direct impact on the capabilities of computation in the type theory. Even if we maintain the core feature of CC and Martin-Löf's early monomorphic type theory in [ML74], namely that type checking is based on a notion of computation, this modification can have a tremendous impact on the capabilities of the type system that ultimately restricts what we can express in the language. More concretely, we do not want to fix a priori in an absolute way what computation rules (e.g. β -reduction or certain forms of algebraic rewrite rules) are available, but we would like to make the notion of available computation rules dependent on the logical facts in the present context. By interpreting logical facts (which can be axioms or theorems) as computation rules in an appropriate way we ensure that the operational semantics is correct w.r.t. the model-theoretic semantics by construction. Completeness of the operational semantics may be desirable for certain purposes (e.g. to decide certain equalities or other predicates), but of course it generally cannot be achieved in the expressive logics that we consider here. We have to accept the fact that the notion of computation gives us an incomplete picture of the logical world, although computational completeness for certain fragments, such as MEL or RWL is clearly possible under suitable restrictions (cf. Chapter 2).

Our approach considerably generalizes more conservative approaches that impose an a priori restriction on the computational system, e.g. to achieve (strong)

normalization by syntactic means. Although maintaining good computational properties, like confluence and (strong) normalization, is encouraged for the relevant application scenarios, we do not enforce these properties and, more importantly, we do not rely on them for the model-theoretic semantics. The semantics we give here is classical set-theoretic one, which has the advantage of intuitive simplicity. By no means do we intend to exclude other model-theoretic semantics, e.g. of constructive nature, which also exist. In fact, the choice of the right semantics strongly depends on the application context. We have chosen a classical set-theoretic semantics here, not only because it fits into the setting of this thesis, but also because it is the semantics which is prevalent in the practice of mathematics and computer science. The semantics that we give in this chapter generalizes the model-theoretic semantics of membership equational logic and rewriting logic which have also been given in the framework of classical set theory (cf. Chapter 2). In a similar way, the operational semantics given in this chapter generalizes the operational semantics of both of these logics.

We refer to the type theory we study in this chapter as the *open calculus of constructions* (OCC), since it is an equational extension of CC with an open computational system and a flexible universe hierarchy. Similar to membership equational logic, OCC supports conditional equations and conditional assertions together with an operational semantics based on conditional rewriting modulo equations. More precisely, there are three types of equalities: structural equality, computational equality and assertional equality. The operational semantics is based on reduction, in the case of computational equality, and on a simple form of goal-oriented deduction, in the case of assertional predicates, of which assertional equality is a special case. In addition to this operational notion of equality, which is also the basis for the type inference and type checking algorithm, we will see that it is straightforward to extend the operational semantics by a notion of nonequational computation in the spirit of rewriting logic. As a consequence we can specify a wide class of labeled transition systems and execute them by rewriting inside the type theory. In this way we are able to close the gap between the paradigms of equational logic, rewriting logic, and type theory, and we also address the problems mentioned above, i.e., we arrive at a type theory that has a more powerful notion of computation and that supports abstract equational specifications to enhance modularity and formal interoperability and furthermore supports symbolic execution of system models beyond pure equational computation.

Based on ideas developed in Chapters 5 and 6, mainly the *explicit substitution calculus* (CINNI) and the notion of *uniform pure type systems* (UPTS), we will introduce the syntax, semantics, and the formal system of OCC in the following sections. Using Maude we have developed an *experimental prototype* of OCC which has several additional features, like definitions, implicit arguments and metavariables, which naturally support a style of goal-oriented programming and

proof construction. The prototype makes use of Maude’s reflective capabilities which allow us to reuse nearly all features of Maude and provides a computational system of acceptable performance. In the present chapter we focus on theoretical aspects concerning the model-theoretic and operational semantics, and the formal system of OCC (Sections 8.1.2 and 8.1.3), and also on a variety applications (Sections 8.2 and 8.3), which have been developed using the OCC prototype. The main purpose of these last two sections is to serve as a proof-of-concept for the language that we are going to introduce. For us the examples were helpful in the design phase of OCC, and we hope that for the reader they will be helpful to understand the pragmatics of OCC, which cannot be conveyed by just presenting a formal system and its semantics. Regarding the OCC prototype, we should add that the current version is experimental and has a number of limitations, but it has very well met its purpose, namely to study the practical consequences of the different design decisions that we contemplated. A discussion of the reflective architecture of the OCC prototype would be worthwhile, since it could be the basis for an actual implementation, but it is beyond the scope of this thesis, which focuses on the theory and pragmatics of OCC rather than on possible implementations. We would furthermore like to point out that in its current version the OCC prototype supports only very specific OCC signatures, implements only a sublanguage of OCC, and requires OCC specifications to satisfy a number of ad hoc restrictions, which are not relevant for the abstract theoretical treatment in the following. Many of these restrictions are currently imposed to make use of the Maude rewriting engine in a very direct way, based on a internal translation of higher-order OCC specifications into first-order rewriting logic specifications, and would most likely disappear in an actual implementation.

8.1 Presentation of the Calculus

8.1.1 Syntax

OCC is a type theory that is concerned with three classes of terms: *elements*, *types* and *universes*. Types serve as an abstraction for collections of elements, and universes as an abstraction for collections of types. However, these three classes are by no means disjoint. In fact, each universe is also a type, namely the type of the types which belong to this universe. Similarly, each type is also an element, namely an element of the universe it belongs to.

In contrast to existing extensions of CC, such as GCC [Coq86], ECC [Luo94], UTT [Luo94], and CIC [CPM90], which all have a fixed universe hierarchy, we follow the idea of pure type systems [Ber88, Ter89] (see also Chapter 6) by introducing OCC as a type theory that is parameterized by signatures defining the universe structure. Unlike pure type systems we do not admit an entirely arbi-

trary structure, but we restrict ourselves to certain well-founded hierarchies of universes,² where impredicative universes can only appear at the bottom (if they are used at all). As in CC we assume the existence of a distinguished universe that we write as **Prop**. Our intention is that **Prop** serves as the *main propositional universe*, i.e. as the first universe of logical types. Indeed, the formal system of OCC is designed to make sense under the *propositions-as-types interpretation*, where propositions are interpreted as types and proofs are interpreted as elements of these types. It is important to emphasize that the designation of **Prop** as a propositional universe does not prevent us from speaking of types in other universes as propositions, because the propositions-as-types interpretation applies to other universes as well. We can see in the following definition of an OCC signature that **Prop** can be either predicative or impredicative. In the first case, **Prop** could become the bottom universe of a predicative hierarchy of propositional universes, whereas in the second case there is basically no need for further propositional universes thanks to the strong closure properties of **Prop**.

An *OCC signature* $\Sigma = (\mathcal{S}, \mathcal{S}^p, \mathcal{S}^i, \preceq, \mathbf{Prop}, \mathcal{A}, \mathcal{R}, \leq)$ consists of a countable set of *OCC universes* \mathcal{S} with disjoint subsets \mathcal{S}^p and \mathcal{S}^i of *predicative* and *impredicative universes*, respectively, a well-founded partial order \preceq on \mathcal{S} , called the *universe hierarchy*, a distinguished universe $\mathbf{Prop} \in \mathcal{S}$, called the *main propositional universe*, an acyclic binary relation $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$, called the *universe containment relation*, a set $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$, called the *set of rules*, and a partial order \leq on \mathcal{S} , called the *subuniverse relation*, such that the following conditions are satisfied.

1. $(s_1, s_2) \in \mathcal{A}$ or $s_1 < s_2$ imply $s_1 \prec s_2$;
2. each universe $s \notin \mathcal{S}^p$ is minimal w.r.t \preceq ;
3. $(s_1, s_2) \in \mathcal{A}$ and $(s_1, s'_2) \in \mathcal{A}$ imply $s_2 = s'_2$;
4. $(s_1, s_2, s_3) \in \mathcal{R}$ and $(s_1, s_2, s'_3) \in \mathcal{R}$ imply $s_3 = s'_3$;
5. $(s_1, s_2) \in \mathcal{A}$ and $s'_1 \leq s_1$ imply that $(s'_1, s'_2) \in \mathcal{A}$ for some $s'_2 \leq s_2$;
6. $(s_1, s_2, s_3) \in \mathcal{R}$ and $s'_1 \leq s_1, s'_2 \leq s_2$ imply that $(s'_1, s'_2, s'_3) \in \mathcal{R}$ for some $s'_3 \leq s_3$;
7. $(s_1, s_2, s_3) \in \mathcal{R}$ and $s_2 \in \mathcal{S}^i$ imply $s_2 = s_3$;

²A similar idea, namely to consider a family of calculi parameterized by well-founded partial orders of universes, has already been used in [Hue87] to define extensions of CC. Furthermore, a very general definition of cumulative PTSs is introduced in [Pol94], but, apart from the fact that OCC and PTSs are very different formal systems, the universe hierarchy is too unconstrained for our purposes and it uses contravariant universe subtyping instead of covariant subtyping like in ECC that we need for our set-theoretic semantics.

8. $(s_1, s_2, s_3) \in \mathcal{R}$ and $s_2 \notin \mathcal{S}^i$ imply $s_1, s_2 \preceq s_3 \in \mathcal{S}^p$;
9. $s \equiv s'$ implies that the least upper bound $s \sqcup s'$ w.r.t. \leq is defined, denoting by \equiv the smallest equivalence relation containing \leq ;
10. the relation \leq on \mathcal{S} is decidable, and the partial functions $\mathcal{A} : \mathcal{S} \rightsquigarrow \mathcal{S}$, $\mathcal{R} : \mathcal{S} \times \mathcal{S} \rightsquigarrow \mathcal{S}$, and $\sqcup : \mathcal{S} \times \mathcal{S} \rightsquigarrow \mathcal{S}$ are computable and their definedness is decidable.

For a motivation of this definition we should first point out that the meaning of \mathcal{A} and \mathcal{R} is the same as in PTS signatures. The well-foundedness of the universe hierarchy, which covers the universe containment relation \mathcal{A} and the subuniverse relation $<$ according to (1), and the minimality of nonpredicative universes (2) are essential for our set-theoretic model construction.³ Conditions (3) and (4) express functionality of \mathcal{A} and \mathcal{R} an obvious requirement for a monomorphic type theory (see Chapter 6 for the corresponding property of PTSs). Conditions (5) and (6) require compatibility of \mathcal{A} and \mathcal{R} with universe specialization. Condition (7) expresses that if a dependent function type can be formed over an impredicative universe s_2 then it must live in s_2 as well, and condition (8) states that if a dependent function type can be formed over any other universe s_2 and its domain is in s_1 then it must live in a predicative universe at least as high as s_1 and s_2 . Condition (7) is equivalent to the requirement that $(s_1, s_2, s_3) \in \mathcal{R}$ and $s_2 \in \mathcal{S}^i$ imply $s_2 \preceq s_3 \in \mathcal{S}^i$ in the presence of (2), which better exhibits the similarity to (8). It is noteworthy that we do not enforce any closure properties of universes and that we have admitted universes that are neither predicative nor impredicative leaving some freedom for possible models (see next section). Condition (9) ensures that types in two semantically related universes s and s' can always be viewed from the perspective of a single well-defined universe $s \sqcup s'$ so that a standard typed equality can be used for types. Condition (10) is finally used to give an effective formal system. In summary, we should say that the conditions on OCC signatures are rather weak and should be strengthened depending on the intended application of OCC. For instance, one may consider only OCC instances with a trivial subuniverse relation if universe subtyping is not desired, or instances of OCC without impredicative universes. Among many other weaker and stronger possibilities, the notion of OCC signatures admits standard universe hierarchies such as those used in ECC or CIC.

Since in OCC there is no a priori distinction between *terms* and *types*, and furthermore between *types* and *propositions*, we use all these notions synonymously

³It is possible to relax these conditions and admit (not necessarily minimal) strongly impredicative universes s satisfying $(s, s) \in \mathcal{A}$, if we give up the possibility of having at least a trivial set-theoretic interpretation for *each* universe. In view of Girard's paradox [Gir72, Coq86] such a universe could not be used for logical purposes, but it could still be computationally meaningful as suggested by [Car86].

from a mathematical point of view. However, we sometimes prefer to use one name in favor of the other to convey our intended interpretation. Given an OCC signature Σ we define *OCC terms* by the following CINNI syntax.

$$s \mid X_m \mid M \ N \mid [X : S]M \mid \{X : S\}T \mid M : T \mid \epsilon A \mid M = N \mid \parallel A \mid !! A \mid ?? A$$

We have to add that in $[X : S]M$ and $\{X : S\}T$ the name X is bound in M and T . Here and in the following, M, N, P, A, B, S, T range over OCC terms, s ranges over OCC universes, and X, Y range over names. For better readability, we use the standard conventions to omit parentheses.

The syntax OCC has the standard constructs known from (uniform) pure type systems (cf. Chapter 6), namely *universes* s , *variables* X_m , typed λ -*abstractions* $[X : S]M$, and *dependent function types* $\{X : S\}T$, but additionally we have *type assertions/casting* $M : T$, the ϵ -*construct* ϵA to denote an irrelevant proof of a proposition A , a built-in *propositional equality* $M = N$, and finally we have three kinds of *operational propositions*, written as $\parallel A$, $!! A$, or $?? A$. Operational propositions can either be *structural propositions* designated by \parallel , *computational propositions* designated by $!!$, or an *assertional propositions* designated by $??$. Subsequently, we use τ to range over $\{\parallel, !!, ??\}$. For convenience, we sometimes write $[X^n : S^n] M$ and $\{X^n : S^n\} T$ to abbreviate $[X_1^n : S_1^n] \dots [X_n^n : S_n^n] M$ and $\{X_1^n : S_1^n\} \dots \{X_n^n : S_n^n\} T$, respectively. Furthermore, we write $S \rightarrow T$ instead of $\{X : S\}T$ if X is not referenced in T , and this notation is extended to lists so that $S^n \rightarrow T$ abbreviates $S_1^n \rightarrow S_2^n \rightarrow \dots \rightarrow S_n^n \rightarrow T$. For better readability, we use the standard convention that \rightarrow is right associative and has a stronger binding precedence than $\{X : S\}T$, so that $S \rightarrow S' \rightarrow S''$ means $S \rightarrow (S' \rightarrow S'')$ and $\{X : S\}T \rightarrow T'$ means $\{X : S\}(T \rightarrow T')$.

OCC contexts are lists of *declarations* of the form $X : S$. The empty context is written as \square . Using Γ to range over OCC contexts, we have to add that in contexts of the form $X : S$, Γ the name X is bound in Γ (but not in S). Finally, we write $Z^n : S^n$ to abbreviate the context $Z_1^n : S_1^n, \dots, Z_n^n : S_n^n$.

An *OCC specification* (Σ, Γ) consists of an OCC signature Σ and an OCC context Γ . An OCC specification (Σ, Γ) can introduce *rewrite predicates* by a declaration in Γ of the form $R : \{Y, Y' : S\} T \rightarrow \mathbf{Prop}$. The idea is that each rewrite predicate can be regarded as a labeled transition system, which can be executed in a very similar way as rewriting logic specifications. Note that $R : \{Y, Y' : S\} T \rightarrow \mathbf{Prop}$ is the declaration of a ternary predicate R where S is the type of states and T is the type of actions, which could range from atomic labels to rewrite proofs, depending on the requirements of the application. In the case where the type T does not depend on Y and Y' , this declaration takes the form $R : S \rightarrow S \rightarrow T \rightarrow \mathbf{Prop}$, but we will see in Section 8.2.8 that dependent types are essential to obtain a strongly typed representation of RWL specifications in the general case.

OCC judgements are either *typing judgements* $\Gamma \vdash M : S$, *type inference judgements* $\Gamma \vdash M \rightarrow : S$, *assertional judgements* $\Gamma \vdash ?? A$, *assertional equality judgements*

ments $\Gamma \vdash ?? (M = N)$, *assertional subtyping judgements* $\Gamma \vdash ?? (S \leq T)$, *structural equality judgements* $\Gamma \vdash || (M = N)$, *computational equality judgements* $\Gamma \vdash !! (M = N)$, *computational rewrite judgements* $\Gamma \vdash !! (R_k M N P)$, and *assertional rewrite judgements* $\Gamma \vdash ?? (R_k M N P)$. Obviously, assertional equality judgements, assertional subtype judgements, and assertional rewrite judgements are special cases of assertional judgements. For a better understanding of the set-theoretic semantics, we first give an intuitive idea of the meaning of these judgements, which are all *relative*, in the sense that they are explained under the assumption that the context Γ is meaningful. The meaning of a *typing judgement* $\Gamma \vdash M : S$ is that M is an element of type S in the context Γ . The stronger *type inference judgement* $\Gamma \vdash M \rightarrow : S$ implies, additionally, that S is an inferred type of M . The *structural equality judgement* $\Gamma \vdash || (M = N)$ means that M and N are structurally equal, i.e., that they cannot be computationally distinguished, in the context Γ . The *computational equality judgement* $\Gamma \vdash !! (M = N)$ means that M reduces to N in the context Γ by means of any number of computation steps. The *assertional judgement* $\Gamma \vdash ?? A$ means that the fact that A is inhabited can be proved in Γ by purely computational means, namely by a combination of exhaustive goal-oriented search and simplification by reduction. Finally, the *computational rewrite judgement* $\Gamma \vdash !! (R_k M N P)$ means that by virtue of the computational rewrite rules specified for the rewrite predicate R_k in Γ , the element M can be rewritten to the element M , and this rewrite is labeled by the element P .

In addition to OCC terms and OCC contexts we use their $\text{CINNI}_{\text{OCC}}$ counterparts, where $\text{CINNI}_{\text{OCC}}$ denotes the instantiation of CINNI to the syntax of OCC, which is completely analogous to the instantiation of CINNI to the syntax of UPTS in Chapter 6. The set of $\text{CINNI}_{\text{OCC}}$ terms/contexts subsumes the set of OCC terms/context, but $\text{CINNI}_{\text{OCC}}$ terms can additionally contain substitutions. On the other hand, substitutions can always be eliminated according to the results of Chapter 5, so that each $\text{CINNI}_{\text{OCC}}$ term/context is equivalent to a unique OCC term/context.⁴ We use this as a justification for introducing the *general convention* to identify $\text{CINNI}_{\text{OCC}}$ terms/contexts that are equivalent by virtue of the equational theory of $\text{CINNI}_{\text{OCC}}$ throughout the present chapter. Recall, however, that this convention does *not* imply that α -equivalent terms are identified. In fact, a key point of our entire treatment is that there is no need for the assumption of α -equality to define a notion of reduction. In analogy with the abbreviations for terms and contexts we write $[X^n := N^n]$, \uparrow_{X^n} and $\uparrow_{X^n} S$ to abbreviate the CINNI substitution $[X_1^n := N_1^n] \dots [X_n^n := N_n^n]$, $\uparrow_{X_1^n} \dots \uparrow_{X_n^n}$ and $\uparrow_{X_n^n} \dots \uparrow_{X_1^n}(S)$. Using this notation, we furthermore write $[X^n : S] M$ and $\{X^n : S\} T$ to abbreviate $[X^{n-1} : S][X^n : \uparrow_{X^{n-1}} S] M$ and $\{X^{n-1} : S\}\{X_n^n :$

⁴Apart from the advantage of a first-order description, the use of CINNI was especially convenient in the OCC prototype, where the addition of metavariables did not require a change in the underlying formalism, because they have to be instantiated eventually.

$\uparrow_{X^{n-1}} S\} T$, respectively, where X^{n-1} abbreviates X_1^n, \dots, X_{n-1}^n .

As we saw above, OCC terms, contexts, and judgements can contain some information, namely the operational kinds $\tau \in \{\|\, \!, \?\?\}$ of propositions, which is essential for the operational semantics given in Section 8.1.3. The model-theoretic semantics, however, which is presented in the following section, completely abstracts from operational aspects and hence ignores this information.

To give a first flavor of OCC we now give a concrete example of an OCC specification (Σ, Γ) . We define Σ as the smallest OCC signature with an ECC-style universe hierarchy that has predicative universes $\text{Type}_i \in \mathcal{S}^p$ for $i \in \mathbb{N}$ and a single impredicative universe $\text{Prop} \in \mathcal{S}^i$ such that $(s \in s'$ abbreviates $(s, s') \in \mathcal{A}$)

$$\text{Prop} \in \text{Type}_0 \in \text{Type}_1 \in \text{Type}_2 \dots,$$

it is cumulative, i.e.

$$\text{Prop} \leq \text{Type}_0 \leq \text{Type}_1 \leq \text{Type}_2 \dots,$$

and has rules

$$(s, \text{Prop}, \text{Prop}) \in \mathcal{R} \text{ and } (s, s', s \sqcup s') \in \mathcal{R}$$

for all $s \in \mathcal{S}$ and $s' \in \mathcal{S}^p$.

The context Γ over this signature is given below. It first introduces a type `nat` with constructors `0` and `suc` and an induction principle `ind`. It then specifies an operation `plus` with structural axioms of commutativity and associativity and two computational axioms. It furthermore introduces a binary predicate `le` that should satisfy two assertional axioms, the second one being a conditional one. Regarding variables in CINNI syntax, we use the convention of Chapter 5 that the index 0 is omitted. Similarly, we simply write `Type` instead of `Type0`.

[`nat : Type`]

[`0 : nat`]

[`suc : nat → nat`]

[`ind : {P : nat → Prop}`

(`P 0`) → (`{i : nat}(P i) → (P (suc i))`) →

`{n : nat}(P n)`]

[`plus : nat → nat → nat`]

[`comm : || {i, j : nat}(plus i j) = (plus j i)`]

[`assoc : || {i, j, k : nat}(plus i (plus j k)) = (plus (plus i j) k)`]

[`plus_eq_1 : !! {i : nat}(plus i 0) = i`]

[`plus_eq_2 : !! {i, j : nat}(plus i (suc j)) = (suc (plus i j))`]

```
[ le : nat → nat → Prop ]
[ le_1 : ?? {j : nat}(le 0 j) ]
[ le_2 : ?? {i, j : nat}(le i j) → (le (suc i)(suc j)) ]
```

Valid and derivable judgements are defined in the next two sections. Here we only give a few examples of such judgements in the context Γ :

```
 $\Gamma \vdash || ((\text{plus } (\text{suc } (\text{suc } 0))(\text{suc } 0)) = (\text{plus } (\text{suc } 0)(\text{suc } (\text{suc } 0))))$ 
 $\Gamma \vdash !! ((\text{plus } (\text{suc } (\text{suc } 0))(\text{suc } 0)) = (\text{suc } (\text{plus } (\text{suc } 0)(\text{suc } 0))))$ 
 $\Gamma \vdash ?? (\text{le } (\text{suc } 0) (\text{suc } (\text{suc } 0)))$ 
 $\Gamma \vdash M : \{n : \text{nat}\}(\text{le } n (\text{suc } n))$ 
```

with $M = (\text{ind } ([n : \text{nat}] (\text{le } n (\text{suc } n)))$
 $(\epsilon (\text{le } 0 (\text{suc } 0)))$
 $([i : \text{nat}][H : ?? (\text{le } i (\text{suc } i))](\epsilon (\text{le } (\text{suc } i)(\text{suc } (\text{suc } i))))))$.

8.1.2 Semantics

The classical set-theoretic semantics that we give in the following has the remarkable feature that all terms and types are interpreted in a uniform way, regardless of their association with a particular universe. Although our semantics should also be classified as a proof-irrelevance semantics (w.r.t. impredicative universes), it is different from the standard approach (see e.g. [Coq90a, Luo94, Wer97]), which requires a special treatment of types in the impredicative universe and hence relies on metatheoretic results such as the subject reduction property. It is well known that the use of a proof-irrelevance approach is unavoidable in a classical set-theoretic semantics, because nontrivial models do not exist [Rey84].⁵ By presenting the semantics before introducing the formal system we do not want to imply that the classical set-theoretic semantics is more fundamental or even a standard semantics for OCC. However, we think that it is useful to make the formal system accessible to the reader and as a tool to prove minimal soundness and (relative) consistency properties. The semantics furthermore explains the connection between OCC and classical mathematics in general, and between OCC and membership equational logic and rewriting logic in particular, which also have classical set-theoretic models as we presented them in Section 2.4. Furthermore, the semantics does not only provide a justification for the formal system of OCC, but also for the use of several classical axioms encountered elsewhere in this thesis (cf. Chapters 4 and 7), such as the law of the excluded middle, logi-

⁵Impredicative polymorphism, however, can be interpreted constructively as suggested by many authors including [Pit87, Mes89b, Gir72], and indeed this fact is exploited in the constructive set-theoretic semantics given for ECC in [Luo94].

cal extensionality, proof irrelevance, and extensional equality for functions. Our semantics is also applicable to several more restrictive extensions of CC, such as GCC and ECC. In principle, it is also applicable to CIC, but due to its classical nature the semantics does not justify the use of an impredicative universe for data types, which we think is not a problem, since in this thesis we adopt the view advocated in [Luo94] that data types should reside in predicative universes in the style of Martin-Löf's type theory.

It is very clear from these explanations that in contrast to many type-theoretic formalisms, starting with Whitehead and Russell's theory of types [WR13], including Martin-Löf's type theories [PSN90], Nuprl [CAB⁺86], and the developments in the line of CC [CH88], we aim by no means at an alternative (possibly more satisfactory) foundation for mathematics. Instead, our more pragmatic motivation is to add a computational layer on top of an existing foundation (here we have chosen Zermelo-Fraenkel set theory), which can be abstracted away, but imposes a strong discipline of use and provides computational means that are lacking in set-theoretic approaches⁶ One may argue that regarding the support of a strong mathematical discipline and by its emphasis on typed morphisms rather than on sets, category theory has a similar objective, except that the notion of computation does not play a central role. Not surprisingly, we will encounter connections between OCC and category theory at various places in this chapter, and we will argue in favor of a computational approach to category theory, for which OCC could provide a possible setting.

In spite of some similarities, the OCC approach is quite different from that of Martin-Löf's polymorphic, intuitionistic type theory [ML74, PSN90] and Nuprl [CAB⁺86, All87], which are both open-ended [Tsu01] in the sense that the semantics is given for the entire underlying untyped language and the introduction of new types and corresponding rules can be justified on this semantic basis. In contrast, OCC is a monomorphic type theory, and our semantics is a classical set-theoretic one, which only gives a meaning to a proper subset of all terms, from which the formal system of OCC again defines the subset of terms, which are well-typed (in an operational sense) and can actually be used in formal developments. Instead of being open-ended from an external point of view, OCC is intended as a framework (in this sense it is similar to Martin-Löf's logical framework [PSN90], which is also a monomorphic type theory) that can be used to specify new types internally without being limited to conservative extensions. Especially important is the capability of OCC to specify the computational system with a high degree of flexibility within the bounds provided by its logic.

Unlike the proof-irrelevance semantics given in [Coq90a, Wer97], our semantics interprets the subuniverse relation directly as inclusion between set-theoretic uni-

⁶There are some notable exceptions such as [Fef75, Bee88] and the approach [Gru92], where sets are represented as maps.

verses rather than just as an injection, which would require us either to make such injections explicit in OCC terms or to ensure that such injections could always be uniquely added, an argument which would have to be based on metatheoretical properties and hence on the formal system of the type theory. Furthermore, a distinguishing feature of our semantics is that everything is interpreted in the expected way with the slight deviation that functions are always assumed to be in a certain “set-theoretic normal form”, a simple and intuitive notion which will be made precise below.

In the following, let $\Sigma = (\mathcal{S}, \mathcal{S}^p, \mathcal{S}^i, \preceq, \text{Prop}, \mathcal{A}, \mathcal{R}, \leq)$ be an arbitrary OCC signature.

A *set-theoretic universe*⁷ is a set that is closed under the standard constructions of Zermelo-Fraenkel set theory with the axiom of choice (ZFC) [Mos94, Sho77].

An *OCC frame* for Σ consists of a family of sets $(\mathcal{U}_s)_{s \in \mathcal{S}}$ and satisfies the following conditions: For each OCC universe $s \in \mathcal{S}$ we impose the minimal condition $\emptyset \in \mathcal{U}_s$. Additionally, for each impredicative OCC universe $s \in \mathcal{S}^i$ we require that \mathcal{U}_s is the two-element boolean set $\mathbb{B} = \{\mathbb{T}, \mathbb{F}\}$ with $\mathbb{T} = \{\emptyset\}$ and $\mathbb{F} = \emptyset$, and for each predicative OCC universe $s \in \mathcal{S}^p$ we require a classical set-theoretic universe \mathcal{U}_s , such that $s \prec s'$ implies $\mathcal{U}_s \in \mathcal{U}_{s'}$ for all OCC universes $s, s' \in \mathcal{S}$.

We immediately observe that in each OCC frame $(s, s') \text{In} \mathcal{A}$ implies $\mathcal{U}_s \in \mathcal{U}_{s'}$, and that $s \leq s'$ implies $\mathcal{U}_s \subseteq \mathcal{U}_{s'}$ for all $s, s' \in \mathcal{S}$, which means that the cumulative hierarchy of OCC universes is strictly reflected in the set-theoretic semantics, i.e. by set-theoretic membership and inclusion. We furthermore define the set $\mathcal{U} = \bigcup \{\mathcal{U}_s \mid s \in \mathcal{S}\}$. In the semantics of OCC we will use the sets in \mathcal{U} to interpret types, and the elements in $\bigcup \mathcal{U}$ to interpret arbitrary terms.

To construct an OCC frame for a fixed OCC signature we use well-known axiomatic extensions of ZFC which are sufficiently strong to interpret all OCC universes. As the starting point for the model construction we use the countable, well-founded partial order \preceq that contains \in and \leq by the definition of an OCC signature. We first give an interpretation for the predicative universes \mathcal{S}^p . To this end, let \preceq' be the restriction of \preceq to \mathcal{S}^p , which is again a countable, well-founded partial order, and consider an extension \preceq'' of \preceq' to a total well-founded order on \mathcal{S}^p . For the standard construction of a cumulative hierarchy of set-theoretic universes we presuppose an axiomatic extension of ZFC by a hierarchy $(\kappa_s)_{s \in \mathcal{S}}$ of strongly inaccessible cardinals [Sho77] so that $s \prec'' s'$ implies $\kappa_s < \kappa_{s'}$ for all $s, s' \in \mathcal{S}^p$. Each strongly inaccessible cardinal κ_s provides us with a set-theoretic universe \mathcal{V}_{κ_s} according to von Neumann’s cumulative hierarchy [Mos94], so that we can simply define \mathcal{U}_s as \mathcal{V}_{κ_s} for each $s \in \mathcal{S}^p$. To complete the construction of an OCC frame it remains to interpret all nonpredicative universes in $\mathcal{S} - \mathcal{S}^p$. The interpretation of impredicative universes $s \in \mathcal{S}^i$ is already fixed by the OCC frame, namely by $\mathcal{U}_s = \mathbb{B} = \{\emptyset, \{\emptyset\}\}$. The interpretation of all remaining

⁷called *Z-F universe* in [Mos94] and just *universe* in [Mac71]

universes $s \in \mathcal{S} - \mathcal{S}^p - \mathcal{S}^i$ can be chosen arbitrarily as long as $\emptyset \in \mathcal{U}_s \in \mathcal{U}_{s_0}$ is satisfied,⁸ where s_0 is the minimal element of \preceq'' .

Corresponding constructions of universes for the semantics of type theories are given in [Luo94] for ECC, in [Wer97] for CIC, and in [How97] for Nuprl.⁹ Alternatively, in order to construct models for arbitrary OCC signatures, we can work with a fixed, but sufficiently strong, extension of ZFC by the Tarski-axiom [Tar38, Tar39], which implies the existence of arbitrary large, strongly inaccessible cardinals. The resulting system is also known as Grothendieck-Tarski set theory and has the property that *each* set is an element of some universe set. This assumption has also been used by Grothendieck to overcome difficulties with the set-theoretic explanation of large categories [Mac71]. Since OCC aims at applications involving formalizations of concepts from category theory, it appears to us that Grothendieck-Tarski set theory is a quite natural choice. It is worth mentioning that this is also the set theory used as the basis for formalized mathematics in the Mizar system [RT99].

To give a natural formulation of the OCC semantics we slightly extend the set-theoretic notion of *function application* $\beta(\alpha)$, which is conventionally only used for $\alpha \in \text{Dom}(\beta)$, by defining $\beta(\alpha) = \emptyset$ for all $\alpha \notin \text{Dom}(\beta)$, provided that β is an *arbitrary* set-theoretic function.¹⁰ In particular, we regard the empty function \emptyset more generally as a constant function which always yields \emptyset . This can be justified intuitively by the view that the empty function is a constant function in disguise (because it has the empty domain). We say that two functions β and β' are *weakly extensionally equivalent* iff $\beta(\alpha) = \beta'(\alpha)$ for all α . Weak extensional equivalence is slightly weaker than the set-theoretic extensional equivalence, i.e. equality of functions viewed as sets. Observe, however, that for the important case of functions β and β' on a common domain, i.e., $\text{Dom}(\beta) = \text{Dom}(\beta')$, both notions coincide. In a context where only the applicative behavior of a function β matters, i.e., the only way to use β is to apply it to an element for which it is *known* to be defined,¹¹ we can replace β by its weakly extensionally equivalent counterpart $\Downarrow\beta$, which is defined by $\Downarrow\beta = \{(\alpha, \alpha') \in \beta \mid \alpha' \neq \emptyset\}$. Observe that for the empty function \emptyset we have $\Downarrow\emptyset = \emptyset$. More generally, for functions β with $\beta(\alpha) \neq \emptyset$ for all $\alpha \in \text{Dom}(\beta)$ we have $\Downarrow\beta = \beta$, i.e., most functions are invariant under \Downarrow . We finally extend \Downarrow to sets by defining $\Downarrow\gamma = \{\Downarrow\beta \mid \beta \in \gamma\}$. In our context, the operators \Downarrow and $\Downarrow\Downarrow$ produce functions and sets of functions in a certain “normal form”, which is used for the interpretation of type-theoretic

⁸It is not difficult to generalize the construction to make sure that s can be interpreted as a full set-theoretic universe, but this would require an additional strongly inaccessible cardinal.

⁹This is the hybrid set-theoretic/computational semantics which justifies Howe’s classical variant of Nuprl that we discussed in Chapter 7.

¹⁰We should emphasize that this is a conservative extension of the set-theoretic notion of function *application*, and should not be confused with an extension of the set-theoretic *functions* themselves, which actually remain completely unchanged.

¹¹without inspecting the set-theoretic representation of the function, e.g. using $\text{Dom}(\beta)$.

functions and function types in the semantics that we introduce below.

Another interesting observation is that given a set γ , a subset $\beta \subseteq \gamma$, and its characteristic boolean function $\hat{\beta} : \gamma \rightarrow \mathbb{B}$ (recall that $\mathbb{B} = \{\emptyset, \{\emptyset\}\}$), we have an isomorphism $\beta \cong \Downarrow \hat{\beta}$, i.e., working with “normal forms” eliminates the gap between sets and their functional representations as characteristic predicates. This last observation also suggests that a set of the form $\Downarrow(\gamma \rightarrow \mathbb{B})$ should be regarded as the set $\mathcal{PS}(\gamma)$ of subsets as opposed to the set $\gamma \rightarrow \mathbb{B}$ of functions, a subtle point which is important for the interpretation of equality.

Given an OCC frame we have to define the interpretation of OCC terms, OCC contexts, and finally OCC judgements. A technical aspect of our approach, is that we eliminate the need for environments, i.e. variable assignments, by using a notion of terms and substitutions with embedded semantic objects, which are of set-theoretic nature in our case. Furthermore, thanks to the use of CINNI, we can define the semantics of OCC in a way which does not require renaming or identification of α -equivalent terms. Indeed, in our entire treatment there is no need for such an assumption.¹² To realize these ideas we first extend the original definition of OCC terms to enriched OCC terms by adding a new term constructor σ with $\bigcup \mathcal{U}$ as its domain so that *enriched OCC terms* are defined by the same syntax as OCC terms but extended by constructs $\sigma(\alpha)$ for all $\alpha \in \bigcup \mathcal{U}$. We usually leave σ implicit, writing α instead of $\sigma(\alpha)$, if there is no danger of confusion. The definition of *enriched OCC contexts* agrees with the definition of OCC contexts, but uses enriched OCC terms instead of terms. Also, we define enriched $\text{CINNI}_{\text{OCC}}$ terms/contexts in complete analogy to $\text{CINNI}_{\text{OCC}}$ terms/contexts, again introducing the *convention* to identify them if they are equivalent by virtue of $\text{CINNI}_{\text{OCC}}$.

We first define a partial interpretation $\llbracket - \rrbracket$. It is a partial function on closed enriched terms which is subsequently extended to enriched contexts, and we generally assume that it is undefined for all cases not covered by the definitions below. We also make use of the standard convention that a set-theoretic formula can only hold and a set-theoretic expression can only be defined if all its subexpressions are defined. We begin with the definition of $\llbracket - \rrbracket$ for closed enriched OCC terms:

$$\begin{aligned} \llbracket \alpha \rrbracket &= \alpha \\ \llbracket s \rrbracket &= \mathcal{U}_s \\ \llbracket M N \rrbracket &= \llbracket M \rrbracket(\llbracket N \rrbracket) \\ \llbracket [X : S]M \rrbracket &= \Downarrow \lambda \alpha \in \llbracket S \rrbracket . \llbracket [X := \alpha]M \rrbracket \text{ if } \llbracket S \rrbracket \in \mathcal{U} \\ \llbracket \{X : S\}T \rrbracket &= \Downarrow \Pi \alpha \in \llbracket S \rrbracket . \llbracket [X := \alpha]T \rrbracket \text{ if } \llbracket S \rrbracket \in \mathcal{U} \\ \llbracket M : S \rrbracket &= \llbracket M \rrbracket \text{ if } \llbracket M \rrbracket \in \llbracket S \rrbracket \in \mathcal{U} \end{aligned}$$

¹²A similar technique can be used to eliminate the need for environments in operational semantics, the specification of the active network language in Appendix B being a good example.

$$\begin{aligned}
\llbracket M = N \rrbracket &= \mathbb{T} \text{ if } \llbracket M \rrbracket = \llbracket N \rrbracket \text{ and } \llbracket M \rrbracket, \llbracket N \rrbracket \in \bigcup \mathcal{U} \\
\llbracket M = N \rrbracket &= \mathbb{F} \text{ if } \llbracket M \rrbracket \neq \llbracket N \rrbracket \text{ and } \llbracket M \rrbracket, \llbracket N \rrbracket \in \bigcup \mathcal{U} \\
\llbracket \epsilon A \rrbracket &= \text{a unique element of } \llbracket A \rrbracket \text{ if } \llbracket A \rrbracket \neq \emptyset \text{ and } \llbracket A \rrbracket \in \mathcal{U} \\
\llbracket \tau P \rrbracket &= \llbracket P \rrbracket \text{ if } \llbracket P \rrbracket \in \mathcal{U}
\end{aligned}$$

On the right hand side of the equations for $\llbracket [X : S]M \rrbracket$ and $\llbracket \{X : S\}T \rrbracket$ we have used the set-theoretic λ -abstraction and the set-theoretic dependent function space, respectively, as defined in Section 2.1. The function and the set of functions that we obtain in these cases is then passed through \Downarrow and \Downarrow , respectively, to obtain our “set-theoretic normal-form”. These operators are needed in our general setting of arbitrary OCC signatures to achieve the strong closure properties of impredicative universes, as we will see in the soundness proof. Recall that the terms $[X:=\alpha]M$ and $[X:=\alpha]T$, used on the left hand side, are equal to enriched OCC terms without substitutions by our earlier convention. It is furthermore important that the definition of $\llbracket MN \rrbracket$ as $\llbracket M \rrbracket(\llbracket N \rrbracket)$ uses the extended notion of function application introduced earlier, which is always defined. The uniqueness requirement in the definition of ϵA can be realized by choosing an element that is minimal w.r.t. a fixed well-founded total order.¹³ The term ϵA is used to denote an unspecified proof of A if A can be proved by means of the operational semantics. Obviously, if \mathbf{Prop} is an impredicative universe and A is a proposition in \mathbf{Prop} , then $\llbracket \epsilon A \rrbracket$ can only be defined if it is an element of \mathbb{T} , which is necessarily \emptyset in our proof-irrelevance semantics. Concerning the interpretation $\llbracket \tau P \rrbracket$ of operational propositions, we can see that the set-theoretic semantics abstracts from their operational nature which is specified by τ .

The partial interpretation $\llbracket - \rrbracket$ of closed enriched OCC contexts is defined by sets of tuples of the form $(\alpha_1, \alpha_2, \dots, \alpha_n) = (\alpha_1, (\alpha_2, (\dots, (\alpha_n, ())))$ as follows:

$$\begin{aligned}
\llbracket [] \rrbracket &= \{()\} \\
\llbracket [X : S, \Gamma] \rrbracket &= \Sigma \alpha \in \llbracket [S] \rrbracket . \llbracket [X:=\alpha] \Gamma \rrbracket \text{ if } \llbracket [S] \rrbracket \in \mathcal{U}
\end{aligned}$$

This concludes the definition of $\llbracket - \rrbracket$ which suggests to call an OCC term M or an OCC context Γ *meaningful* iff $\llbracket M \rrbracket$ or $\llbracket \Gamma \rrbracket$, respectively, is defined. This notion of meaningful terms is essential for the soundness of the formal system presented in the next section. According to our definition, a universe s is always meaningful, and a term $M N$, $M : S$, $M = N$, ϵA , or τP , can only be meaningful if its immediate subterms are meaningful. Furthermore, a term $[X : S] M$ or $\{X : S\} T$ can only be meaningful if S is meaningful and $[X:=\alpha]M$ or $[X:=\alpha]T$, respectively, is meaningful for all $\alpha \in \llbracket [S] \rrbracket$. Similarly, a context $X : S, \Gamma$ can only be meaningful if S is meaningful and $[X:=\alpha]\Gamma$ is meaningful for all $\alpha \in \llbracket [S] \rrbracket$.

¹³It is actually a special case of Hilbert’s ϵ -operator.

Given an *OCC specification* (Σ, Γ) with $\Gamma = Z^n : S^n$, a *model* of (Σ, Γ) consists of an OCC frame over Σ and a tuple $\gamma^n \in \llbracket \Gamma \rrbracket$.

Based on the partial interpretation $\llbracket - \rrbracket$ of enriched OCC terms we define *validity* of OCC judgements. We again make use of the standard convention that a set-theoretic formula can only hold and a set-theoretic expression can only be defined if all its subexpressions are defined. For arbitrary set-theoretic expressions α , we use the abbreviation $\downarrow \alpha$ to express that α is defined. We assume that the validity predicate is false in all cases not covered by the following definition. Validity is a unary predicate over OCC judgements, but we write $\Gamma \models J$ instead of the more cumbersome $\models \Gamma \vdash J$.

$$\begin{aligned}
& \models M \rightarrow : S \text{ if } \llbracket M \rrbracket \in \llbracket S \rrbracket \\
& \models M : S \text{ if } \downarrow \llbracket S \rrbracket \text{ implies } \llbracket M \rrbracket \in \llbracket S \rrbracket \\
& \models ?? A \text{ if } \downarrow \llbracket A \rrbracket \text{ implies } \llbracket A \rrbracket \neq \emptyset \\
& \models ?? (M = N) \text{ if } (\downarrow \llbracket M \rrbracket \text{ and } \downarrow \llbracket N \rrbracket) \text{ implies } \llbracket M \rrbracket = \llbracket N \rrbracket \\
& \models ?? (S \leq T) \text{ if } (\downarrow \llbracket S \rrbracket \text{ and } \downarrow \llbracket T \rrbracket) \text{ implies } \llbracket S \rrbracket \subseteq \llbracket T \rrbracket \\
& \models || (M = N) \text{ if } (\downarrow \llbracket M \rrbracket \text{ or } \downarrow \llbracket N \rrbracket) \text{ implies } \llbracket M \rrbracket = \llbracket N \rrbracket \\
& \models !! (M = N) \text{ if } \downarrow \llbracket M \rrbracket \text{ implies } \llbracket M \rrbracket = \llbracket N \rrbracket \\
& \models !! (R_k M N P) \text{ if } \downarrow \llbracket M \rrbracket \text{ implies } \llbracket (R_k M N P) \rrbracket \neq \emptyset \\
& \models ?? (R_k M N P) \text{ if } \downarrow \llbracket M \rrbracket, \downarrow \llbracket N \rrbracket \text{ and } \downarrow \llbracket P \rrbracket \text{ implies } \llbracket (R_k M N P) \rrbracket \neq \emptyset \\
& \Gamma \models J \text{ if } \downarrow \llbracket \Gamma \rrbracket \text{ implies } \models [Z^n := \gamma^n] J \text{ for all } \gamma^n \in \llbracket \Gamma \rrbracket
\end{aligned}$$

where Γ is of the form $Z^n : S^n$.

Concerning this definition of valid judgements it should be noted that the judgements are semantically rather weak. As an example, the validity of the typing judgements $\Gamma \vdash M : S$ *assumes* that Γ and S are meaningful and asserts under this condition that in each model of Γ the term M is meaningful and the interpretation of M is contained in the interpretation of S . This is slightly different from the type inference judgement $\Gamma \vdash M \rightarrow : S$, which *only assumes* that Γ is meaningful and asserts that in each model of Γ both terms M and S are meaningful and the interpretation of M is contained in the interpretation of S . Furthermore, we should point out that none of these judgements explicitly requires that the interpretation of S is an element of some universe \mathcal{U}_s . The rationale behind the definedness conditions will become clear with the introduction of the operational meaning of these judgements. The general idea is to keep the judgements semantically as weak as possible to justify an efficient formal system without redundant deductions, which is presented in the next section.¹⁴ Obviously, it is easy to define

¹⁴The idea of explaining judgements under the assumption that the context is meaningful has been already used in Nuprl and is a deviation from Martin-Löf's explanation of judgements and from the explanation of judgements in CC and other PTSs, which all imply that the context is meaningful.

stronger judgements on the basis of existing ones if the need arises.

Finally, we should point out that the need for our notion of a “set-theoretic normal form of functions” only arises when we consider OCC signatures with impredicative universes. In the case, where no impredicative universes are used, we can omit \Downarrow , $\bar{\Downarrow}$ and use the standard set-theoretic function application in the interpretation of OCC terms. The result is the entirely straightforward set-theoretic semantics which has already been used for Martin-Löf’s type theory [Dyb91, DS99] and for predicative universes of ECC [Luo94] and CIC [Wer97].

8.1.3 Formal System

An essential feature of type theories with dependent types in the family of pure types systems (PTS) is that the typing judgements are defined on the basis of the operational semantics of the type theory. In the standard presentations of PTSs, and of CC in particular, the operational semantics, which is based on a notion of reduction, is first defined for the untyped language, and then by employing a metatheoretic argument, the subject reduction property to be precise, it is shown that the operational semantics respects the type system and therefore constitutes in fact a typed operational semantics. In OCC the connection between the typing judgements and the operational semantics is bidirectional and more complex than in PTSs and in existing extensions of CC. It therefore requires a different approach, which is reflected in the formal system of OCC which simultaneously defines both, the operational semantics and well-typed terms with their types. Since the deductive judgements of logics such as MEL and RWL correspond to typing judgements in OCC by virtue of the propositions-as-types interpretation, we can also say that the formal system of OCC represents both the computational system and the deductive system, which cannot be separated from each other. A particular aspect of our approach is that the judgements of OCC are very liberal in the sense that they do not require well-typedness of their constituents but are justified instead on purely semantic grounds.

Compared with the model-theoretic semantics, the formal system gives an independent and more refined explanation of all OCC judgements by making essential distinctions between typing judgements and type inference judgements as well as between the remaining judgements. It especially makes explicit the distinction in the operational behaviour of the equality judgements that all have a very similar model-theoretic semantics in terms of set-theoretic equality. To make the presentation of the formal system of OCC and the operational semantics, which is explained on the basis of this formal system, more accessible for the reader we first give a brief informal explanation of all judgements and their intuitive operational meaning.

- The *type inference judgement* $\Gamma \vdash M \rightarrow: S$ asserts that the term M is an

element of the *inferred type* S in the context Γ . Operationally, Γ and M are given and S is obtained by syntax-directed type inference and possible reduction using computational equations modulo the structural equations of Γ . Hence, we are interested in solutions to incomplete goals of the form $\Gamma \vdash M \rightarrow: ?$.

- The *typing judgement* $\Gamma \vdash M : S$ asserts that M is an *element of type* S in the context Γ . Operationally, Γ , M and S are given and verifying $\Gamma \vdash M : S$ amounts to type checking. In the inference system given below, type checking is always reduced to type inference and the verification of an assertional subtyping judgement, which subsumes the assertional equality judgement (see below).
- The *structural equality judgement* $\Gamma \vdash || (M = N)$ is used to express that M and N are considered to be structurally equal elements in the context Γ . Operationally, structural equality is realized by a suitable term representation so that structurally equal terms cannot be distinguished when they participate in computations.
- The *computational equality judgement* $\Gamma \vdash !! (M = N)$ is the judgement that defines the notion of reduction for the simplification of terms. The judgement states that the element M can be reduced to the element N in the context Γ . Operationally, Γ and M are given and N is the result of reducing M using the computational equations in Γ modulo the structural equations in Γ . Hence, operationally we are interested in solutions to incomplete goals of the form $\Gamma \vdash !! (M = ?)$.
- The *assertional judgement* $\Gamma \vdash ?? A$ states that A is provable by means of the operational semantics in the context Γ . Operationally, Γ and A are given and the judgement is verified by a combination of reduction using the computational equations and exhaustive goal-oriented search using the assertional propositions in Γ . Both processes take place modulo the structural equations in Γ . Just as in the case of structural and computational equality, the verification of assertional judgements is absorbed by the computational system of OCC and hence does not involve explicit proof objects.
- The *assertional equality judgement* $\Gamma \vdash ?? (M = N)$ states that M and N are assertionally equal in Γ , a notion that treats equality as a predicate and subsumes the structural and computational equality judgements. Operationally, Γ , M and N are given and the judgement is verified like other assertional judgements in a goal-oriented fashion.
- The *assertional subtyping judgement* $\Gamma \vdash ?? (S \leq T)$ states that S is a subtype of T in Γ as a consequence of the cumulativity of the universe

hierarchy. Operationally, Γ , S and T are given and the judgement is verified like other assertional judgements in a goal-oriented fashion.

- The *computational rewrite judgement* $\Gamma \vdash !! (R_k M M' P)$ expresses that by means of the computational rewrite rules specified in Γ for the rewrite predicate R_k the element M can be rewritten to the element M' and this rewrite is labeled by the element P . Operationally, Γ and M are given and M' is computed by the application of a computational rewrite rule in Γ modulo the computational and structural equations in Γ . In addition, an abstract witness P for this rewrite is constructed. Hence, operationally we are interested in solutions to incomplete goals of the form $\Gamma \vdash !! (R_k M ? ?)$.
- The *assertional rewrite judgement* $\Gamma \vdash ?? (R_k M M' P)$ is a special case of the assertional judgement and subsumes the computational rewrite judgement. Operationally, Γ , M , M' , and P are given, and the judgement can be verified by means of a corresponding computational rewrite judgement.

In the remainder of this section we introduce the *formal system of OCC*, which defines *derivability* of OCC judgements, for a given OCC signature. The formal system is also the basis for the *operational semantics* of OCC, which is a generalization of the operational semantics of MEL and RWL presented in Chapter 2. It is the partial specification of an algorithm that realizes the judgements in the sense explained earlier within the possibilities admitted by the formal system. We should emphasize that the intention of the formal system is not to require implementations to realize it in full generality, but the formal system should be seen as a framework opening a space in which different implementations of various degrees of generality are possible. The operational semantics which we describe in the following together with the formal system explains on a suitably abstract level how the inference rules are implemented and also imposes some important restrictions on how they are applied, but on the other hand leaves considerable freedom for possible implementations.

To emphasize the operational goal-oriented view of the formal system we have grouped the inference rules according to the form of their conclusions. Since operationally the context of each conclusion is an input, which can be independently checked for well-typedness, all our judgements are *relative judgements* in the sense that well-typedness of the context is assumed rather than implied. As in Chapter 6, it is straightforward to add rules for well-typed context judgements and for a corresponding stronger typing judgement.

Although OCC is inspired by PTSs, it is important to stress already at this point that OCC and PTSs are quite different systems, as indicated by the rich collection of OCC judgements and their explanation given earlier. Already the admission of judgements involving terms that are not well-typed (and possibly not even meaningful) is a strong deviation of the underlying philosophy of PTSs (and

especially CC), but is very natural in semantics-based non-foundational approach to type theory. Another important difference to PTSs is the presence of fully cumulative universe subtyping in OCC in the style of ECC [Luo94] (which has also been adopted in CIC) instead of the simple cumulativity of universes used in Martin-Löf's type theory [ML74, ML84] and in GCC [Coq86]. This leads not only to a less restrictive type system, but more importantly it avoids some the difficulties with type checking in GCC, which was the main motivation to introduce full cumulativity in [Luo94, Luo91].

Finally, we should point out that the formal system of OCC has strong computational flavor, and is in this respect similar to the operational version of PTSs that we introduced in Chapter 6. Although a more abstract formal system for OCC can be useful for metatheoretic investigations, we think that in a very liberal type theory like OCC the computational aspects are so important for the user that they should receive an explicit representation in the formal system itself. The decision to especially make explicit the algorithmic nature of type checking in the formal system (as for instance witnessed by the use of type inference judgements) is quite different from the abstract presentations of other type theories. It reflects our view, that the type checking process itself is of utmost importance and not less important than the computational system itself. We feel furthermore that typechecking should be as simple and as explicit as possible, because it cannot be completely hidden from the user (e.g. by reference to completeness w.r.t. a more abstract formal system) in computationally very expressive systems like OCC.

The *type inference judgements* $\Gamma \vdash M \rightarrow : S$ are closed under the following rules. Most of these rules are similar to the corresponding rules for the optimized version of uniform pure types systems of Chapter 6. Especially, we define the function *lookup* as in Section 6.2.4.

$$\frac{}{\Gamma \vdash s_1 \rightarrow : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{A} \quad (\text{Ax})$$

$$\frac{}{\Gamma \vdash Z_m \rightarrow : \text{lookup}(\Gamma, Z_m)} \quad \text{if } \text{lookup}(\Gamma, Z_m) \text{ is defined} \quad (\text{Lookup})$$

$$\frac{\Gamma \vdash S \rightarrow : s_1 \quad \Gamma, X : S \vdash T \rightarrow : s_2}{\Gamma \vdash \{X : S\}T \rightarrow : s_3} \quad \text{if } (s_1, s_2, s_3) \in \mathcal{R} \quad (\text{Pi})$$

$$\frac{\Gamma \vdash S \rightarrow : s_1 \quad \Gamma, X : S \vdash M \rightarrow : T}{\Gamma \vdash [X : S]M \rightarrow : \{X : S\}T} \quad (\text{Lda})$$

$$\frac{\Gamma \vdash M \rightarrow : \{X : S\}T \quad \Gamma \vdash N : S}{\Gamma \vdash (M N) \rightarrow : [X := N] T} \quad (\text{App})$$

$$\frac{\Gamma \vdash M \rightarrow: \tau \{X : S\}T \quad \Gamma \vdash N : S}{\Gamma \vdash (M N) \rightarrow: [X:=N] T} \quad (\text{AppTag})$$

$$\frac{\Gamma \vdash T \rightarrow: s \quad \Gamma \vdash T' \rightarrow: s' \quad \text{if } s \sqcup s' \text{ is defined}}{\Gamma \vdash (T = T') \rightarrow: \text{Prop}} \quad (\text{Eq1})$$

$$\frac{\Gamma \vdash M \rightarrow: T \quad \Gamma \vdash M' \rightarrow: T' \quad \Gamma \vdash T \rightarrow: s \quad \Gamma \vdash T' \rightarrow: s' \quad \Gamma \vdash ?? (T = T')}{\Gamma \vdash (M = M') \rightarrow: \text{Prop}} \quad \text{if } s \sqcup s' \text{ is defined} \quad (\text{Eq2})$$

$$\frac{\Gamma \vdash A \rightarrow: s \quad \Gamma \vdash ?? A}{\Gamma \vdash \epsilon A \rightarrow: A} \quad (\text{Epsilon})$$

$$\frac{\Gamma \vdash A \rightarrow: s}{\Gamma \vdash \tau A \rightarrow: s} \quad (\text{Tag})$$

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash S \rightarrow: s}{\Gamma \vdash (M : S) \rightarrow: S} \quad (\text{Cast})$$

$$\frac{\Gamma \vdash M \rightarrow: T \quad \Gamma \vdash !! (T = T')}{\Gamma \vdash M \rightarrow: T'} \quad (\text{TypeRed})$$

Since our judgements do not imply that their context is well-typed or even meaningful, it is easy to see already from the rule **Lookup** (and also from **Lda** in comparison with **Pi**) that derivability of $\Gamma \vdash M \rightarrow: S$ does not generally imply that S is well-typed, because it exists in some universe. Instead, the type inference judgements define the subset of well-typed types and well-typed elements as follows: We say that T is a *well-typed type* in the context Γ iff $\Gamma \vdash T \rightarrow: s$ is derivable for some universe s . We furthermore say the term M is a *well-typed element* in the context Γ iff $\Gamma \vdash M \rightarrow: T$ is derivable and T is a well-typed type in Γ .

Operationally, all rules for type inference judgements are applied in a goal-oriented fashion. All the rules for the type inference judgement are syntax-directed, except for **Eq1**, **Eq2**, and **TypeRed**. The latter allows reduction of the inferred type using computational equations modulo structural equations. Operationally, we require that this rule is implemented by an arbitrary reduction without backtracking, which must be nontrivial in the sense defined below to avoid trivial nontermination.

Since reduction preserves semantic equality, it follows that the inferred type for a well-typed term in a given context is semantically unique, which is why OCC is a *monomorphic type theory*. As we will discuss later, the context Γ is, however, even if it is well-typed not necessarily operationally strong enough to ensure well-

typedness of the inferred type or uniqueness w.r.t. to assertional, structural, or computational equality. Another noteworthy point, which is a consequence of the context-dependent operational semantics, is that in the rule **Lda** the term S in the conclusion $\Gamma \vdash [X : S]M \rightarrow: \{X : S\}T$ can be an operational proposition, which under the goal-oriented reading of this rule becomes part of the context of the new goal $\Gamma, X : S \vdash M \rightarrow: T$ and can then be operationally involved in the type inference process for M . A similar possibility exists in the rule **Pi**, where the goal $\{X : S\}T \rightarrow: s_3$ gives rise to a new goal $\Gamma, X : S \vdash T \rightarrow: s_2$, and S can again be an operational proposition.

The *typing judgement* $\Gamma \vdash M : T$ is always reduced to the type inference judgement and a single assertional subtyping judgement according to the following rule, which again is implemented in a goal-directed fashion.

$$\frac{\Gamma \vdash M \rightarrow: S \quad \Gamma \vdash ?? (S \leq T)}{\Gamma \vdash M : T} \quad (\text{Conv})$$

The typing judgements $\Gamma \vdash M : T$ are more liberal than the corresponding type inference judgements $\Gamma \vdash M \rightarrow: T$. This is reasonable, because it allows us to exploit the fact that more information is available for type checking than for type inference. Indeed, in the context Γ both M and T are available as input for the type checking algorithm, so that assertional subtyping can be used in **Conv**. Since, **Conv** is the only rule which allows us to derive typing judgements of the form $\Gamma \vdash M : T$, we can infer from this judgement that $\Gamma \vdash M \rightarrow: S$ and $\Gamma \vdash S \leq T$ for some S by inverting the rule. Hence, the operational meaning of $\Gamma \vdash M : T$ is that the inferred type of M in Γ is a subtype of T , where assertional subtyping subsumes assertional equality as we will see in the subsequent rules.

The *structural equality judgements* $\Gamma \vdash \parallel (M = N)$ are closed under the following rules.

$$\frac{}{\Gamma \vdash \parallel (M = M)} \quad (\text{StrRefl})$$

$$\frac{\Gamma \vdash \parallel (M = N)}{\Gamma \vdash \parallel (N = M)} \quad (\text{StrSym})$$

$$\frac{\Gamma \vdash \parallel (P = Q) \quad \Gamma \vdash \parallel (Q = R)}{\Gamma \vdash \parallel (P = R)} \quad (\text{StrTrans})$$

$$\frac{\Gamma \vdash Z \rightarrow: \parallel \{X^n : T^n\}(M = M') \quad \Gamma \vdash N_i^n : [X^{i-1} := N^{i-1}]T_i^n \text{ for } i \in \{1 \dots n\}}{\Gamma \vdash \parallel [X^n := N^n](M = M')} \quad (\text{Str})$$

In addition, structural equality is closed under the following context rules.

$$\frac{\Gamma \vdash \parallel (A = A')}{\Gamma \vdash \parallel ((\tau A) = (\tau A'))} \quad (\text{StrTag})$$

$$\frac{\Gamma \vdash \parallel (A = A')}{\Gamma \vdash \parallel ((\epsilon A) = (\epsilon A'))} \quad (\text{StrEpsilon})$$

$$\frac{\Gamma \vdash \parallel (M = M') \quad \Gamma \vdash \parallel (S = S')}{\Gamma \vdash \parallel ((M : S) = (M' : S'))} \quad (\text{StrCast})$$

$$\frac{\Gamma \vdash \parallel (M = M') \quad \Gamma \vdash \parallel (N = N')}{\Gamma \vdash \parallel ((M = N) = (M' = N'))} \quad (\text{StrEq})$$

$$\frac{\Gamma \vdash \parallel (M = M') \quad \Gamma \vdash \parallel (N = N')}{\Gamma \vdash \parallel ((M N) = (M' N'))} \quad (\text{StrApp})$$

$$\frac{\Gamma \vdash \parallel (S = S')}{\Gamma \vdash \parallel ([X : S]M = [X : S']M)} \quad (\text{StrLda})$$

$$\frac{\Gamma \vdash \parallel (S = S')}{\Gamma \vdash \parallel (\{X : S\}T = \{X : S'\}T)} \quad (\text{StrPi})$$

According to the rule **Str**, the structural equality in a context Γ is generated by all the equations in Γ that are designated by \parallel as structural equations. Operationally, structural equality judgements are realized by the choice of a suitable term representation depending on the given context. More precisely, when used in computational or assertional judgements with a context Γ the terms M and M' are computationally indistinguishable iff $\Gamma \vdash \parallel (M = M')$. The structural equality judgement can be seen as an instantaneous reversible transformation in arbitrary subterms, which justifies reflexivity, symmetry, transitivity, and the context rules above.

As in the operational semantics of MEL and RWL, we now adopt the general convention that in a given context Γ we identify structurally equal terms in all computational and assertional judgements, which are generated by the remaining rules of the formal system.

The *computational equality judgements* $\Gamma \vdash !! (M = N)$ are closed under the following rules.

$$\frac{}{\Gamma \vdash !! (M = M)} \quad (\text{RedRef})$$

$$\frac{\Gamma \vdash !! (P = Q) \quad \Gamma \vdash !! (Q = R)}{\Gamma \vdash !! (P = R)} \quad (\text{RedTrans})$$

$$\frac{\begin{array}{l} \Gamma \vdash Z \rightarrow: !! \{X^n : T^n\} C^m \rightarrow (M = M') \\ \Gamma \vdash N_i^n : [X^{i-1} := N^{i-1}] T_i^n \text{ for } i \in \{1 \dots n\} \\ \Gamma \vdash ?? [X^n := N^n] C_i^m \text{ for } i \in \{1 \dots m\} \end{array}}{\Gamma \vdash !! [X^n := N^n] (M = M')} \quad (\text{Red})$$

$$\frac{\Gamma \vdash N : S}{\Gamma \vdash !! ((([X : S]M)N) = [X := N] M)} \quad (\text{RedBeta})$$

$$\frac{\Gamma \vdash M : S}{\Gamma \vdash !! ((M : S) = M)} \quad (\text{RedCast})$$

In addition, computational equality judgements enjoy context rules corresponding to those for structural equality judgements, which we have omitted here.

The rule **Red** expresses that computational equality in a context Γ is generated by those equations of Γ designated by $!!$ as computational. The rule is of a more general form than **Str**, because computational equations can be conditional. More precisely, the rule **Red** generates a computational equality judgement $\Gamma \vdash !! [X^n := N^n] (M = M')$ using the proposition $\{X^n : T^n\} C^m \rightarrow (M = M')$, which must be designated by $!!$ as a computational equation in the context Γ according to the first premise of the rule. By the propositions-as-types interpretation the dependencies $\{X_i^n : T_i^n\}$ represent universal quantifiers and the non-proper dependencies C_i^m represent logical conditions. The most typical case is that the computational equation $\{X^n : T^n\} C^m \rightarrow (M = M')$ takes the form of a universally quantified Horn clause. On the other hand, we do not exclude universal quantifiers and implications inside conditions, a possibility which leads us beyond Horn clause logic and has a well-defined operational meaning thanks to the rule **AssGen** given later.

Operationally, the computational equality judgement $\Gamma \vdash !! (M = M')$ is realized by reduction using the computational equations in Γ by virtue of **Red**, and the fixed computation rules **RedBeta** and **RedCast**. By our earlier convention, reduction should take place modulo the structural equality. As in MEL, we use the term *reduction* to refer to rewriting *without backtracking*, and we do not fix the method that is used to find applicable instances of axioms.

We next present the general rules **Ass**, **AssRed**, and **AssGen** for assertional judgements $\Gamma \vdash ?? A$. These rules apply in particular to assertional equality judgements $\Gamma \vdash ?? (M = N)$, assertional subtyping judgements $\Gamma \vdash ?? (M \leq N)$, and assertional rewrite judgements $\Gamma \vdash ?? (R_m M M' P)$.

$$\frac{\begin{array}{l} \Gamma \vdash Z \rightarrow: ?? \{X^n : T^n\} C^m \rightarrow A \\ \Gamma \vdash N_i^n : [X^{i-1} := N^{i-1}] T_i^n \text{ for } i \in \{1 \dots n\} \\ \Gamma \vdash ?? [X^n := N^n] C_i^m \text{ for } i \in \{1 \dots m\} \end{array}}{\Gamma \vdash ?? [X^n := N^n] A} \quad (\text{Ass})$$

$$\frac{\Gamma \vdash ?? A' \quad \Gamma \vdash !! (A = A')}{\Gamma \vdash ?? A} \quad (\text{AssRed})$$

$$\frac{\Gamma, X : T \vdash ?? A}{\Gamma \vdash ?? \{X : T\}A} \quad (\text{AssGen})$$

The rule **Ass** proves the assertional goal $[X^n := N^n]A$ using the proposition $\{X^n : T^n\} C^m \rightarrow A$, which must be designated by $??$ as an assertion in the context Γ according to the first premise of the rule. Universal quantifiers and conditions can be used in assertional axioms in the same way as they can be used in computational equations, that is, again the dependencies $\{X_i^n : T_i^n\}$ represent universal quantifiers and the non-proper dependencies C_i^m represent logical conditions. The rule **AssRed** allows the reduction of the current goal, and finally, there is a rule **AssGen** to prove both quantified and conditional goals. Similar to the rule **Lda**, in the conclusion $\Gamma \vdash ?? \{X : T\}A$ of **AssGen** the type T can be an operational proposition, which becomes part of the context of the new goal $\Gamma, X : T \vdash ?? A$ and can then be operationally involved in the proof of A .

As in the operational semantics of MEL, we require that the rules for assertional judgements are implemented in a goal-oriented fashion by an exhaustive strategy *with backtracking* with the exception of the rule **AssRed** which uses reduction and hence does not involve backtracking. In particular, this ensures that all possibilities to apply assertional axioms admitted by the formal system are explored, provided this is not prevented by nonterminating conditions. To avoid trivial nontermination, we furthermore require that the reduction in **AssRed** is nontrivial. In summary, the entire process of verifying assertions is based on exhaustive goal-oriented search, which can be interleaved with equational simplification of the goal.

In addition to the previous rules for general assertional judgements, the *assertional equality judgements* $\Gamma \vdash ?? (M = N)$ are closed under the following rules.

$$\frac{}{\Gamma \vdash ?? (M = M)} \quad (\text{EqRefl})$$

$$\frac{\Gamma \vdash ?? (M = N)}{\Gamma \vdash ?? (N = M)} \quad (\text{EqSym})$$

$$\frac{\Gamma \vdash ?? (A = B)}{\Gamma \vdash ?? ((\tau A) = B)} \quad (\text{EqTag})$$

$$\frac{\Gamma \vdash ?? (A = A')}{\Gamma \vdash ?? ((\epsilon A) = (\epsilon A'))} \quad (\text{EqEpsilon})$$

$$\frac{\Gamma \vdash ?? (M = M') \quad \Gamma \vdash ?? (N = N')}{\Gamma \vdash ?? ((M = N) = (M' = N'))} \quad (\text{EqEq})$$

$$\frac{\Gamma \vdash ?? (M = M') \quad \Gamma \vdash ?? (N = N')}{\Gamma \vdash ?? ((M N) = (M' N'))} \quad (\text{EqApp})$$

$$\frac{\Gamma \vdash ?? (S = S') \quad \Gamma, Y : S' \vdash ?? ([X:=Y] \uparrow_Y T = T')}{\Gamma \vdash ?? ([X : S]T = [Y : S']T')} \quad (\text{EqLda})$$

$$\frac{\Gamma \vdash ?? (S = S') \quad \Gamma, Y : S' \vdash ?? ([X:=Y] \uparrow_Y T = T')}{\Gamma \vdash ?? (\{X : S\}T = \{Y : S'\}T')} \quad (\text{EqPi})$$

As we see from the rules **EqSym** and **EqRefl**, assertional equality is symmetric and reflexive w.r.t. to structural equality, and enjoys more general context closure rules than structural and computational equality in the cases **EqLda** and **EqPi**, where variable binding constructs are involved. In particular, we can see that α -conversion is admitted in both of these rules. The lack of a transitivity rule for assertional equality judgements is on purpose, because such a rule is not sound under our semantics for assertional equality judgements, and furthermore it is not suitable for the operational semantics for assertional judgements, which is based on exhaustive goal-oriented search.

Operationally, we are still concerned with the assertional judgements which are implemented in an exhaustive goal-oriented fashion as explained before, but as an exception we require that the rule **EqSym**, which would lead to trivial non-termination if implemented as a goal-oriented rule, is implemented directly by a representation of assertional equality judgements which does not distinguish between $\Gamma \vdash ?? (M = N)$ and $\Gamma \vdash ?? (N = M)$. This amounts to the use of **EqSym** as a structural equivalence in the logical sense.

In addition to the rules for general assertional judgements, the rules for the *assertional subtyping judgements* $\Gamma \vdash ?? (S \leq T)$ are the following.

$$\frac{}{\Gamma \vdash ?? (s \leq s')} \quad \text{if } s \leq s' \quad (\text{SubUniv})$$

$$\frac{\Gamma \vdash ?? (S = T)}{\Gamma \vdash ?? (S \leq T)} \quad (\text{SubEq})$$

$$\frac{\Gamma \vdash ?? (A \leq B)}{\Gamma \vdash ?? ((\tau A) \leq B)} \quad (\text{SubTag1})$$

$$\frac{\Gamma \vdash ?? (A \leq B)}{\Gamma \vdash ?? (A \leq (\tau B))} \quad (\text{SubTag2})$$

$$\frac{\Gamma \vdash ?? (S = S') \quad \Gamma, Y : S' \vdash ?? ([X:=Y] \uparrow_Y T \leq T')}{\Gamma \vdash ?? (\{X : S\}T \leq \{Y : S'\}T')} \quad (\text{SubPi})$$

The only use of subtyping judgements in this presentation of OCC is *universe subtyping*.¹⁵ We follow the proposal of [Luo94] for ECC of extending the sub-universe relation \leq to a relation between types, which is here represented by subtyping judgements. Observe, however, that there is no transitivity rule for assertional subtyping judgements, for the same reasons that we have not included such a rule for assertional equality judgements. Furthermore, the rule **SubPi** only admits covariant subtyping as in ECC. Obviously, contravariant subtyping would not be compatible with our set-theoretic semantics which interprets subtyping as inclusion.

In addition to the rules for general assertional judgements, the *assertional rewrite judgement* $\Gamma \vdash ?? (R_m M M' P)$ is closed under the following rule.

$$\frac{\Gamma \vdash !! (R_m M N P) \quad \Gamma \vdash ?? (P = P') \quad \Gamma \vdash ?? (N = N')}{\Gamma \vdash ?? (R_m M N' P')} \quad (\text{RewAss})$$

Operationally, this rule is again implemented in a goal-oriented fashion, and simply provides an assertional counterpart for the computational rewrite judgement. Typically, assertional rewrite goals arise from rewrite conditions in computational rewrite axioms, which are generated by the subsequent rule.

The *computational rewrite judgements* $\Gamma \vdash !! (R_k M N P)$ are closed under the following rules.

$$\frac{\begin{array}{l} \Gamma \vdash R_k \rightarrow: \{Y, Y' : S\} T \rightarrow \text{Prop} \\ \Gamma \vdash Z \rightarrow: !! \{X^n : T^n\} C^m \rightarrow (\uparrow_{X^n} R_k M M' P) \\ \Gamma \vdash N_i^n : [X^{i-1} := N^{i-1}] T_i^n \text{ for } i \in \{1 \dots n\} \\ \Gamma \vdash ?? [X^n := N^n] C_i^m \text{ for } i \in \{1 \dots m\} \end{array}}{\Gamma \vdash !! [X^n := N^n] (\uparrow_{X^n} R_k M M' P)} \quad (\text{Rew})$$

$$\frac{\begin{array}{l} \Gamma \vdash !! (R_k M' M'' P) \\ \Gamma \vdash !! (M = M') \quad \Gamma \vdash !! (M'' = M''') \end{array}}{\Gamma \vdash !! (R_k M M''' P)} \quad (\text{RewModulo})$$

Operationally, the computational rewrite judgement $\Gamma \vdash !! (R_k M N P)$ is realized by rewriting using the computational rewrite axioms in Γ . By our earlier convention, rewriting takes place modulo structural equality. The rule **RewModulo** expresses, furthermore, that rewriting takes place modulo computational equality, and we impose the usual operational requirement to avoid trivial nontermination, namely that at least one of the two reductions involved is nontrivial. As in the

¹⁵A quite general form of (covariant) subtyping could be realized by means of a few additional rules and in full analogy to assertional equality.

case of RWL, the operational semantics does not specify the rewrite strategy, which is in general application-dependent, and it does not fix the method used to find instances of applicable axioms, because we do not want to exclude the most general case where this can be a matter of the rewrite strategy. Furthermore, since OCC does not have fixed closure rules for rewriting like those of RWL, such rules have to be explicitly stated as computational rewrite axioms in Γ , and rewriting is required to respect these closure rules rather than using them as rewrite rules. We feel that stating explicitly the closure properties that are actually needed in OCC specifications does not only improve clarity, but it also leads to a higher degree of flexibility and to more economic induction principles.

A final restriction, that the operational semantics should satisfy is that the sub-goals arising from the conditions of operational axioms in the rules **Ass**, **Red**, and **Rew** are solved in the order in which they are given in the axiom. Recall that a corresponding property has been required in the operational semantics of MEL and RWL to deal with conditions which only terminate if certain other conditions are satisfied.¹⁶

This concludes the presentation of the formal system of OCC and its operational semantics. It is worth emphasizing once again that exactly the same operational semantics that is used for the execution of programs and of specifications is used for type checking and type inference. In the remainder of this section we continue with a discussion of the formal system of OCC and also try to shed light on some of the design decisions involved.

Similar to MEL and RWL, it is clear that already due to the possibility of using arbitrary structural equations the operational semantics of OCC cannot be implemented in the general case. Therefore we discuss in the following a number of sufficient conditions under which an implementation is feasible, which are also among the restrictions imposed by the OCC prototype. First of all, the structural equations should be universally quantified equations (without conditions) of a standard form which allows a representation of the induced equivalence classes by suitable data structures so that matching algorithms modulo structural equality are available.¹⁷ The rules **Red**, **Ass**, and **Rew** are then implemented using matching modulo structural equality to determine all potentially applicable instances of axioms, which leads to similar restrictions on the use of variables as those discussed in the context of MEL and RWL. It should however be mentioned that more general implementations are possible that defer the instantiation of universally quantified variables when they cannot be determined by matching

¹⁶This and other restrictions could be made explicit in the formal system of OCC, but for our high-level presentation it seems to be clearer to impose it at the strategy level.

¹⁷The structural equations currently supported by the OCC prototype are all combinations of associativity, commutativity and identity laws. Not surprisingly, these are the kinds of structural equations currently supported by Maude, which serves as an underlying rewriting engine.

and use the conditions to find potential solutions. Conditions with matching equations (cf. Section 2.4.4) as they are supported by the most recent version of Maude are an example of such a generalization. Concerning the implementation of rewrite judgements, rewriting has to be restricted to certain subterms for which it can be justified by the closure axioms that are present in the current context.¹⁸ Hence, closure axioms are not treated as rewrite rules, but should be, similar to structural equations, of a standard form which can be recognized and directly supported by the implementation. The remaining rewrite rules should satisfy a variable restriction similar to that imposed for rules in RWL, but again more general implementations are possible. For instance, the variable restriction could be relaxed, if the instantiation of axioms is determined by a rewrite strategy or based on a rewrite counterpart of matching equations.

We conclude this section with a discussion of the formal system of OCC and its operational semantics. First of all, we would like to recall that the formal system introduces *different notions of equality* which are related in a hierarchical way. Computational equality is reflexive and transitive, but is not required to be symmetric, since it is intended to reflect directed computation. By **RedRefl** and by our convention to identify structurally equal terms, computational equality subsumes structural equality. Furthermore, assertional equality subsumes computational equality by our earlier rule **AssRed** and, last but not least, all equalities are subsumed by the propositional equality, which itself implies semantic equality. This provides the user with an entire hierarchy ranging from finer to coarser equalities with very different operational capabilities. The right choices of the appropriate equalities are important decisions made by the user, since they determine the computational suitability of the specification for his particular application. Regarding the different notions of equality, it is interesting to remark that the built-in rules for structural and computational equality always preserve the names of bound variables thanks to the underlying CINNI calculus. This can be especially observed in the rule **RedBeta**, which preserves names in OCC, but is defined modulo α -conversion in other type theories. Also the type inference judgement, which is closed under computational equality, preserves names as we can see in the rule **Lda**. Assertional equality and subtyping on the other hand are coarser and subsume α -conversion, which is indispensable for type checking (rule **Conv**). Our key point is that α -conversion can be completely avoided at the level of the directed notion of computation defined by computational equality.

A major difference between the formal systems of OCC and CC is that OCC is based on a *context-dependent operational semantics*, rather than on a fixed notion

¹⁸The OCC prototype currently supports only a single rewrite predicate and specifications have to contain closure axioms that justify sequential rewriting at the top, which is the default rewrite strategy of Maude. As in the current version of Maude, rewrite proofs are presently not constructed.

of reduction. Furthermore the operational semantics is *inherently untyped*,¹⁹ in contrast to that of CC, which is intended to be used only with well-typed terms. The operational semantics of OCC is given by the structural and computational equality/rewrite judgements and by the assertional judgements which subsume especially the assertional equality and subtyping judgements. All these judgements cannot be defined independently of the typing and type inference judgements, as we can see in the rules **Ass**, **Str**, **Red**, **RedBeta**, and **RedCast**, where type checking conditions can be generated that are themselves verified using the operational semantics. Since these type checking conditions are generated during the execution of OCC specifications we also refer to them as *dynamic type checking conditions*.

The design decision to use an *open computational system*, which is fully controlled by the user, has a number of important implications. Similar to specifications in MEL and RWL, we have explained the operational semantics without enforcing OCC specifications to be confluent, normalizing, or computationally complete w.r.t. certain fragments of its logic. Instead, our applications of OCC suggest that often specifications are only partially terminating or partially confluent and are still computationally useful. In addition, our examples indicate that the simple syntactic notion of confluence is unnecessarily strong in view of the fact that nonconfluent specifications can still be confluent w.r.t. assertional equality or other equivalences that are very specific to the application. Still it is important to mention however that in addition to the potential sources of nonconfluence and nontermination known from first-order languages like MEL or RWL, there are new, more subtle, potential sources of nonconfluence and nontermination in OCC that the user has to be aware of. For instance, in terms $[X : S] M$ or $\{X : S\} T$ where S is an operational proposition, e.g. a computational equation, the reduction of M or T may be in conflict with a reduction of S . Furthermore, the lack of transitivity rules for assertional equality and subtyping requires the user to take care of sufficiently strong transitivity properties himself for the relevant cases, which is clearly a potential source of computational incompleteness. Useful specifications should exhibit good operational properties in scenarios that are relevant for the application, but it may be difficult, impossible, or even irrelevant to achieve these properties in general. For instance, compared with normalizing type theories, we can use the full expressivity of nonterminating equational computation in OCC, but it is the responsibility of the user that the concrete terms used in the relevant application scenarios are actually normalizing w.r.t. the relevant notion of equality. Since type inference and type checking are based on the operational semantics, an obvious drawback of this flexibility is that there is no guarantee that type inference and type checking are confluent, terminating, or complete either. In fact, the computational system associated with each context

¹⁹One may also say that the operational semantics is typed in a very liberal sense on the basis of a very general notion of type that is not made explicit in the formal system we have given in this thesis.

ultimately determines what are the well-typed terms in the given context and hence what can be expressed in the type theory, and any incompleteness of the computational system can potentially lead to an incompleteness of the type inference and type checking process, although it will never be a source of unsoundness. Having discussed only equational computation so far, it is important to keep in mind that OCC has also a notion of non-equational computation, which is given by the computational rewrite judgements. For typical RWL applications such as the representation of concurrent and deductive systems, nonterminating and even non-confluent computations are the most typical case, which is why no particular restrictions are imposed by OCC at this point either.

The second important point mentioned above is that the formal system and especially the operational semantics of OCC is *inherently untyped* in a similar sense as the operational semantics of the Nuprl type theory, which goes back to Martin-Löf's ideas. Since computation with non-well-typed terms can be semantically justified, there is in particular no need for a syntactic notion of subject reduction in our treatment. Actually, this would be an unnecessarily restrictive notion in our context, where computation can discard (type) information that is essential for type checking but semantically irrelevant, and where the computational system is potentially incomplete.²⁰ For instance, since reduction of a meaningful term preserves semantic equality, the result is always meaningful, but it may not be well-typed in the given context, because the context could be computationally incomplete. In this situation, the user has the opportunity to complete the context accordingly, a task that typically involves the enrichment by operational propositions, which can often be proved as theorems inside the type theory. To be more specific, there are two closely related reasons why non-well-typed terms can be introduced. First of all, in several rules, **RedBeta** is an example, a variable X of type T in a given term M is replaced by a subterm N of type T , i.e. with an inferred type that is a subtype of T . A computationally useful context for this scenario should be sufficiently complete to verify that the inferred type of the resulting term $[X:=N] M$ is a subtype of the original type of M , so that M and $[X:=N] M$ can be equipped with equal types. On the other hand, it is easy to violate this property by a computationally incomplete specification of assertional equality/subtyping. A second more subtle reason, why non-well-typed terms can be generated is of a different nature, but again can be explained using the **RedBeta** rule as an example. If the term M is well-typed in the context $\Gamma, X : S$ then $[X:=N] M$ with N as above is not necessarily well-typed in the context Γ , because S may be an operational proposition that is not present in Γ . In other words, the context Γ may be operationally less powerful than $\Gamma, X : S$. Again, the user has to decide if it is relevant for the application to complete the context

²⁰Although subject reduction does not make sense for our computational notion of types, it would be a natural property of a more liberal notion of types, which could also be made explicit in the formal system.

to ensure that well-typedness of $[X:=N] M$ can be verified. Other rules, where a variable is substituted by a concrete term, and where the operational context can change accordingly, are **Ass**, **Str**, and **Red**, and also the rule **App**, which is used to derive type inference judgements.

In summary, we see that OCC imposes a considerably weaker typing discipline than existing type theories in the line of CC. In our view, this is an inevitable consequence of the openness and the flexibility of the underlying computational system. In the following we show that the typing discipline is still sufficiently strong to ensure soundness and consistency of the type theory as a logic. It is especially important to emphasize that soundness or consistency is never compromised by the incompleteness of the computational system.

8.1.4 Soundness and Consistency

In this section we show that the formal system of OCC is sound w.r.t. to the classical set-theoretic semantics given earlier, a result that immediately implies consistency of the formal system as a logic, i.e. under the propositions-as-types interpretation.

Let $\Sigma = (\mathcal{S}, \mathcal{S}^p, \mathcal{S}^i, \preceq, \mathbf{Prop}, \mathcal{A}, \mathcal{R}, \leq)$ be an arbitrary OCC signature, and let $(\mathcal{U}_s)_{s \in \mathcal{S}}$ be an arbitrary OCC frame for Σ .

The soundness result states that each judgement $\Gamma \vdash J$ that is derivable is also valid. Since the semantics of OCC is defined using the CINNI calculus the proof of this theorem uses the algebraic properties of explicit substitutions that have been previously proved in Chapter 5. Due to the simplicity of the set-theoretic semantics the soundness of most rules is straightforward to verify. There are only a few rules for which soundness is not obvious, namely those which depend on the particularities of our “set-theoretic normal form” of functions.

Theorem 8.1.1 (Soundness)

For each derivable OCC judgement $\Gamma \vdash J$ we have $\Gamma \models J$.

Proof Sketch.

Straightforward by induction over derivable judgements. The only interesting cases are the rules **Pi**, **Lda**, **App**, and **RedBeta**. We treat each of these rules in detail below.

Soundness of Pi

If $\Gamma \vdash J$ has been derived using **Pi** it is of the form $\Gamma \vdash \{X : S\}T \rightarrow: s_3$ and the side condition $(s_1, s_2, s_3) \in \mathcal{R}$ holds. By the induction hypothesis $\Gamma \models S \rightarrow: s_1$ and $\Gamma, X : S \models T \rightarrow: s_2$. To verify $\Gamma \models \{X : S\}T \rightarrow: s_3$ it is sufficient to show that $\llbracket [Z^n := \gamma^n] \{X : S\}T \rrbracket \in \llbracket s_3 \rrbracket$ assuming $\gamma^n \in \llbracket \Gamma \rrbracket$. Observe that our

assumptions imply (A) $\llbracket [Z^n := \gamma^n] S \rrbracket \in \llbracket s_1 \rrbracket$ and (B) $\llbracket [Z^n := \gamma^n] [X := \alpha] T \rrbracket \in \llbracket s_2 \rrbracket$ for all $\alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket$. We now distinguish two cases:

1. Case $s_2 \notin \mathcal{S}^i$. This implies $s_1, s_2 \preceq s_3 \in \mathcal{S}^p$, so that $\llbracket s_3 \rrbracket$ is a set-theoretic universe with $\llbracket s_1 \rrbracket, \llbracket s_2 \rrbracket \subseteq \llbracket s_3 \rrbracket$. By definition $\llbracket [Z^n := \gamma^n] \{X : S\}T \rrbracket = \bar{\downarrow} \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket$. Using the fact that $\llbracket s_3 \rrbracket$ is closed under all set-theoretic constructions we obtain $\bar{\downarrow} \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket \in \llbracket s_3 \rrbracket$, since $\llbracket [Z^n := \gamma^n] S \rrbracket \in \llbracket s_3 \rrbracket$ and $\llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket \in \llbracket s_3 \rrbracket$ for all $\alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket$.
2. Case $s_2 \in \mathcal{S}^i$. This implies $s_2 = s_3 \in \mathcal{S}^i$, so that $\llbracket s_2 \rrbracket = \llbracket s_3 \rrbracket = \{\emptyset, \{\emptyset\}\}$. In view of (B) we have $\llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket \in \llbracket s_2 \rrbracket$ for all $\alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket$. Hence, we can verify $\llbracket [Z^n := \gamma^n] \{X : S\}T \rrbracket \in \llbracket s_3 \rrbracket = \llbracket s_2 \rrbracket$ by considering the following cases:
 - (a) If $\llbracket [Z^n := \gamma^n] S \rrbracket = \emptyset$ then we have $\llbracket [Z^n := \gamma^n] \{X : S\}T \rrbracket = \bar{\downarrow} \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket = \{\emptyset\}$, since the only function with an empty domain is \emptyset , and $\bar{\downarrow} \{\emptyset\} = \{\emptyset\}$.
 - (b) If $\llbracket [Z^n := \gamma^n] S \rrbracket \neq \emptyset$ and for each $\alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket$ we have $\llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket = \{\emptyset\}$ then it follows that $\llbracket \{X : [Z^n := \gamma^n] S\}T \rrbracket = \bar{\downarrow} \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket = \{\emptyset\}$, since $\Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket$ can only contain a constant function β yielding \emptyset for all arguments, so that $\bar{\downarrow} \{\beta\} = \{\emptyset\}$.
 - (c) If $\llbracket [Z^n := \gamma^n] S \rrbracket \neq \emptyset$ and there is an $\alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket$ with $\llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket = \emptyset$ then $\llbracket \{X : [Z^n := \gamma^n] S\} \uparrow_X [Z^n := \gamma^n] T \rrbracket = \bar{\downarrow} \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket = \emptyset$, because we have $\Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket = \emptyset$, which holds because each function in this set should map α to an element of $\llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket = \emptyset$, which is impossible.

Soundness of Lda

If $\Gamma \vdash J$ has been derived using **Lda** it is of the form $\Gamma \vdash [X : S]M \rightarrow: \{X : S\}T$. By the induction hypothesis $\Gamma \models S \rightarrow: s_1$ and $\Gamma, X : S \models M \rightarrow: T$. To verify that $\Gamma \models [X : S]M \rightarrow: \{X : S\}T$ it is sufficient to show that $\llbracket [Z^n := \gamma^n] [X : S]M \rrbracket \in \llbracket [Z^n := \gamma^n] \{X : S\}T \rrbracket$ assuming $\gamma^n \in \llbracket \Gamma \rrbracket$. Observe that our assumptions imply (A) $\llbracket [Z^n := \gamma^n] S \rrbracket \in \llbracket s_1 \rrbracket$ and (B) $\llbracket [Z^n := \gamma^n] [X := \alpha] M \rrbracket \in \llbracket [Z^n := \gamma^n] [X := \alpha] T \rrbracket$ for all $\alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket$.

We have to prove $\llbracket [Z^n := \gamma^n] [X : S]M \rrbracket \in \llbracket [Z^n := \gamma^n] \{X : S\}T \rrbracket$ which in view of (A) is equivalent to $\bar{\downarrow} \lambda \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] M \rrbracket \in \bar{\downarrow} \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket$. By the definition of $\bar{\downarrow}$ it is suffi-

cient to show that $\lambda \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] M \rrbracket \in \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] T \rrbracket$, which in turn follows from (B).

Soundness of App

If $\Gamma \vdash J$ has been derived using **App** it is of the form $\Gamma \vdash (M N) \rightarrow: [X := N] T$. By the induction hypothesis $\Gamma \models M \rightarrow: \{X : S\}T$ and $\Gamma \models N : S$. To verify that $\Gamma \models (M N) \rightarrow: [X := N] T$ it is sufficient to prove $\llbracket [Z^n := \gamma^n] (M N) \rrbracket \in \llbracket [Z^n := \gamma^n] [X := N] T \rrbracket$ assuming $\gamma^n \in \llbracket \Gamma \rrbracket$. Observe that our assumptions imply that (A) $\llbracket [Z^n := \gamma^n] N \rrbracket \in \llbracket [Z^n := \gamma^n] S \rrbracket$ (here we have used that $\llbracket [Z^n := \gamma^n] S \rrbracket$ is defined, which follows from $\Gamma \models M \rightarrow: \{X : S\}T$) and (B) $\llbracket [Z^n := \gamma^n] M \rrbracket \in \llbracket [Z^n := \gamma^n] \{X : S\} T \rrbracket = \Downarrow \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . [X := \alpha] \uparrow_X [Z^n := \gamma^n] T$.

By the definition of \Downarrow there is a function $\beta \in \Pi \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . [X := \alpha] \uparrow_X [Z^n := \gamma^n] T$, so that $\llbracket [Z^n := \gamma^n] M \rrbracket = \Downarrow \beta$, and $\llbracket [Z^n := \gamma^n] M \rrbracket (\llbracket [Z^n := \gamma^n] N \rrbracket) = \beta(\llbracket [Z^n := \gamma^n] N \rrbracket)$. Hence, $\llbracket [Z^n := \gamma^n] (M N) \rrbracket = \llbracket [Z^n := \gamma^n] M \rrbracket (\llbracket [Z^n := \gamma^n] N \rrbracket) \in \llbracket [X := \llbracket [Z^n := \gamma^n] N \rrbracket] \uparrow_X [Z^n := \gamma^n] T \rrbracket = \llbracket [Z^n := \gamma^n] [X := N] T \rrbracket$.

Soundness of RedBeta

If $\Gamma \vdash J$ has been derived using **RedBeta** it is of the form $\Gamma \vdash !! ((([X : S]M)N) = [X := N] M)$. From the induction hypothesis we obtain $\Gamma \models N : S$, and we have to verify that $\Gamma \models !! ((([X : S]M)N) = [X := N] M)$. To this end it is sufficient to show that $\llbracket [Z^n := \gamma^n] ((([X : S]M)N) = [X := N] M) \rrbracket = \llbracket [Z^n := \gamma^n] [X := N] M \rrbracket$ assuming that $\llbracket [Z^n := \gamma^n] ((([X : S]M)N) = [X := N] M) \rrbracket$ is defined and $\gamma^n \in \llbracket \Gamma \rrbracket$. Definedness of $\llbracket [Z^n := \gamma^n] ((([X : S]M)N) = [X := N] M) \rrbracket$ obviously implies definedness of $\llbracket [Z^n := \gamma^n] S \rrbracket$, which allows us to apply our induction hypothesis $\Gamma \models N : S$ to infer $\llbracket [Z^n := \gamma^n] N \rrbracket \in \llbracket [Z^n := \gamma^n] S \rrbracket$. Using this fact we obtain $\llbracket [Z^n := \gamma^n] ((([X : S]M)N) = [X := N] M) \rrbracket = \llbracket ([X : S] \uparrow_X [Z^n := \gamma^n] M) [Z^n := \gamma^n] N \rrbracket = (\Downarrow \lambda \alpha \in \llbracket [Z^n := \gamma^n] S \rrbracket . \llbracket [X := \alpha] \uparrow_X [Z^n := \gamma^n] M \rrbracket) (\llbracket [Z^n := \gamma^n] N \rrbracket) = \llbracket [X := \llbracket [Z^n := \gamma^n] N \rrbracket] \uparrow_X [Z^n := \gamma^n] M \rrbracket = \llbracket [X := [Z^n := \gamma^n] N] \uparrow_X [Z^n := \gamma^n] M \rrbracket = \llbracket [Z^n := \gamma^n] [X := N] M \rrbracket$, where we have used our earlier observation that \Downarrow respects weak extensionality, i.e., does not change the applicative behavior of a function. □

As a simple corollary we obtain consistency of the formal system, i.e., that we cannot prove the absurd proposition $\perp_s = \{X : s\}X$ for any universe s in the empty context.

Corollary 8.1.2 (Consistency) The formal system of OCC is *consistent*, i.e., $\square \vdash M : \perp_s$ is not derivable for any M, s .

Proof By the soundness theorem $\square \vdash M : \perp_s$ implies $\square \models M : \perp_s$. Furthermore, $\square \models M : \{X : s\}X$ implies $\forall \gamma \in \{()\} . \Downarrow \Pi \alpha \in \llbracket s \rrbracket . \alpha \neq \emptyset$ which is clearly false, because our definition of OCC frames requires $\emptyset \in \llbracket s \rrbracket$ and therefore $\Pi \alpha \in \llbracket s \rrbracket . \alpha = \emptyset$. □

8.2 Examples in Specification and Programming

In tutorial form we will now present a number of small examples to demonstrate the pragmatics and the capabilities of OCC in the context of specification and programming. By giving a number of executable specifications we demonstrate the computational system of OCC which generalizes membership equational logic to higher-order logic with dependent types. We furthermore demonstrate the interplay between dependent types and the computational system. Similar to our introduction to membership equational logic in Section 2.4 we restrict our attention to initial models in the case of nonparameterized specifications and more generally to free models in the case of parameterized specifications if nothing else is mentioned. We use the notions of initiality and freeness informally until we have introduced the right tools to express such constraints using the higher-order logic of OCC in Section 8.2.7. Finally, we should say that for all examples we assume an OCC signature with an ECC-style universe hierarchy as defined at the end of Section 8.1.1.

8.2.1 Executable Equational Specifications

To begin with a simple example of an executable equational specification, we repeatedly use the following specification of booleans, which according to the above convention has an implicit initiality constraint implying that `bool` is a type with two distinct elements `true` and `false`.

```
( bool : Type )
( true : bool )
( false : bool )
```

This is the syntax used by the OCC prototype to extend the current context by additional declarations. Here and in the following we adhere to the convention of the OCC prototype that every user input is surrounded by parentheses. For sake of brevity, we have usually omitted the response by the system in all cases where it is a simple confirmation or where it is not relevant for our explanations.

The standard functions `not` and, `or`, and `xor` are now specified using computational equations, i.e. equational axioms that are interpreted as reduction rules in the operational semantics.

```
( not : bool -> bool )
( not_eq_1 : !! (EQ (not false) true) )
( not_eq_2 : !! (EQ (not true) false) )

( and : bool -> bool -> bool )
```

```

( and_eq_1 : !! {b : bool} (EQ (and false b) false) )
( and_eq_2 : !! {b : bool} (EQ (and b false) false) )
( and_eq_3 : !! {b : bool} (EQ (and true true) true) )

( or : bool -> bool -> bool )
( or_eq_1 : !! {b : bool} (EQ (or true b) true) )
( or_eq_2 : !! {b : bool} (EQ (or b true) true) )
( or_eq_3 : !! {b : bool} (EQ (or false false) false) )

( xor : bool -> bool -> bool )
( xor_eq_1 : !! {b : bool} (EQ (xor true b) (not b)) )
( xor_eq_2 : !! {b : bool} (EQ (xor b true) (not b)) )
( xor_eq_3 : !! {b : bool} (EQ (xor false b) b) )
( xor_eq_4 : !! {b : bool} (EQ (xor b false) b) )
( xor_eq_5 : !! {b : bool} (EQ (xor b b) false) )

```

The operational semantics allows us to evaluate terms in this context using the `Red` command:

```

( Red (not (and (or true false) (or false true) ) ) )
false

```

In a similar way we can specify natural numbers together with a total function `nat_minus`.

```

( nat : Type )
( 0 : nat )
( suc : nat -> nat )

( 1 = (suc 0) )
( 2 = (suc 1) )
( 3 = (suc 2) )
( 4 = (suc 3) )

( nat_minus : nat -> nat -> nat )

( nat_minus_eq_1 : !! {i : nat}
  (EQ (nat_minus i 0) i) )
( nat_minus_eq_2 : !! {i,j : nat}
  (EQ (nat_minus (suc i) (suc j)) (nat_minus i j)) )

( Red (nat_minus 4 2) )
( suc ( suc 0 ) )

```

Notice that `nat_minus` is left unspecified for some arguments to reflect undefinedness of its set-theoretic counterpart. This is one possible way to deal with partial functions in logics with total functions such as OCC. An alternative approach, which is discussed later and often used in Maude specifications, is to represent undefinedness explicitly using subsorts (see Section 8.2.1).

Structural vs. Computational Equality

So far we have used only computational equations. Given the previous specification of `bool` we sometimes want to evaluate terms with variables such as `unknown` and `unknown'` in the following example:

```
( unknown : bool )
( unknown' : bool )

( Red (not (and (or true unknown) (or false true) ) ) )
false

( Red (xor unknown' (xor (xor unknown unknown') unknown)) )
(xor unknown' (xor (xor unknown unknown') unknown))
```

None of the computational equations is applicable in the last case, although we could prove by induction (for examples of inductive theorem proving see Section 8.3.3) that the term above is equal to `false`. Computational equality can be often enhanced with structural equations. For instance, we could use the following alternative specification for `xor` which specifies commutativity and associativity as structural equations. We can also see that structural equations can capture natural symmetries thereby reducing the number of computational equations.

```
( xor : bool -> bool -> bool )

( xor_comm : || {b,b' : bool}
  (EQ (xor b b') (xor b' b)) )
( xor_assoc : || {b,b',b'' : bool}
  (EQ (xor (xor b b') b'') (xor b (xor b' b''))) )
( xor_eq_1 : !! {b : bool} (EQ (xor false b) b) )
( xor_eq_2 : !! {b : bool} (EQ (xor true b) (not b)) )
( xor_eq_3 : !! {b : bool} (EQ (xor b b) false) )
```

Returning to our previous example we can now evaluate the following term with variables to a useful result:

```
( Red (xor unknown' (xor (xor unknown unknown') unknown)) )
false
```

Continuing our specification of `nat`, we can specify `nat_plus` using structural equations in the following way:

```
( nat_plus : nat -> nat -> nat )

( nat_plus_comm : || {i,j : nat}
  (EQ (nat_plus i j) (nat_plus j i)) )
( nat_plus_assoc : || {i,j,k : nat}
  (EQ (nat_plus i (nat_plus j k)) (nat_plus (nat_plus i j) k)) )

( nat_plus_eq_1 : !! {i : nat}
  (EQ (nat_plus i 0) i) )
( nat_plus_eq_2 : !! {i,j : nat}
  (EQ (nat_plus i (suc j)) (suc (nat_plus i j))) )

( Red (nat_plus 3 4) )
( suc 0 ) ) ) ) ) ) ) )

( n : nat )

( Red (nat_plus 3 (nat_plus n 4)) )
( suc n ) ) ) ) ) ) ) )
```

Another example that we reuse later is multiplication and its use in an equational definition of the factorial function:

```
( nat_mult : nat -> nat -> nat )

( nat_mult_comm : || {i,j : nat}
  (EQ (nat_mult i j) (nat_mult j i)) )
( nat_mult_assoc : || {i,j,k : nat}
  (EQ (nat_mult i (nat_mult j k)) (nat_mult (nat_mult i j) k)) )

( nat_mult_eq_1 : !! {i : nat}
  (EQ (nat_mult 0 i) 0) )
( nat_mult_eq_2 : !! {i : nat}
  (EQ (nat_mult (suc 0) i) i) )
( nat_mult_eq_3 : !! {i,j : nat}
  (EQ (nat_mult i (suc j)) (nat_plus i (nat_mult i j))) )
```

```
( nat_fact : nat -> nat )

( nat_fact_eq_1 : !!
  (EQ (nat_fact 0) (suc 0)) )
( nat_fact_eq_2 : !! {n : nat}
  (EQ (nat_fact (suc n)) (nat_mult (suc n) (nat_fact n))) )

( Red (nat_fact 3) )
( suc ( suc ( suc ( suc ( suc ( suc 0 ) ) ) ) ) ) )
```

Definition by Pattern Matching Modulo

An alternative specification of `nat` can be given by viewing `nat` as a commutative monoid with identity element. Instead of constructors `0` and `suc` as in the standard specification above, we use `0`, `1` and `nat_plus` as constructors together with suitable structural equations stating that `nat_plus` is associative and commutative with left and right identity `0`.

```
( nat : Type )

( 0 : nat )
( 1 : nat )

( nat_plus : nat -> nat -> nat )

( nat_plus_comm : || {i,j : nat}
  (EQ (nat_plus i j) (nat_plus j i)) )
( nat_plus_assoc : || {i,j,k : nat}
  (EQ (nat_plus i (nat_plus j k)) (nat_plus (nat_plus i j) k)) )
( nat_plus_right_id : || {i : nat}
  (EQ (nat_plus i 0) i) )
( nat_plus_left_id : || {i : nat}
  (EQ (nat_plus 0 i) i) )

( 2 = (nat_plus 1 1) )
( 3 = (nat_plus 2 1) )
( 4 = (nat_plus 3 1) )
```

This unusual representation of natural numbers has different operational properties. For instance, it allows us to define functions by pattern matching modulo the structural equations above.

```

( nat_half : nat -> nat )
( nat_half_eq_0 : !! {n : nat}
  (EQ (nat_half (nat_plus n n)) n) )
( nat_half_eq_1 : !! {n : nat}
  (EQ (nat_half (nat_plus n (nat_plus n 1))) n) )

( Red (nat_half 4) )
( nat_plus 1 1 )
( Red (nat_half 3) )
1
( Red (nat_half 0) )
0

```

Membership Equational Specifications

Membership equational logic together with its operational semantics based on reduction modulo for equational axioms and exhaustive goal-oriented search for membership axioms forms a natural sublogic of OCC. As an example consider the following theory from Section 2.4.3:

```

sort Nat NzNat .

subsort NzNat < Nat .

op 0 : -> Nat .
op suc : Nat -> NzNat .
op pred : NzNat -> Nat .

var n : Nat .
eq pred(suc(n)) = n .

```

Recall (from Section 2.4.4) that this Maude theory is syntactic sugar for the following MEL theory with a single kind `Nat?`, which in Maude syntax can be written as:

```

sort Nat NzNat .

var n : Nat? .

cmb n : Nat if n : NzNat .

op 0 : -> Nat? .

```

```

mb 0 : Nat .

op suc : Nat? -> Nat? .
mb suc(n) : NzNat if n : Nat .

op pred : Nat? -> Nat? .
cmb pred(n) : Nat if n : NzNat .

ceq pred(suc(n)) = n if n : Nat .

```

Now each MEL kind is represented as a type in OCC. Sorts are represented as unary predicates. Equational and membership axioms have a direct representation as Horn clauses with operational meaning, i.e. marked by !! or ??, respectively.

```

( Nat? : Type )

( Nat : Nat? -> Prop )
( NzNat : Nat? -> Prop )
( subsort_as : ?? {n : Nat?} (NzNat n) -> (Nat n) )

( 0 : Nat? )
( 0_as : ?? (Nat 0) )

( suc : Nat? -> Nat? )
( suc_as : ?? {n : Nat?} (Nat n) -> (NzNat (suc n)) )

( pred : Nat? -> Nat? )
( pred_as : ?? {n : Nat?} (NzNat n) -> (Nat (pred n)) )

( pred_eq : !! {i : Nat?} (Nat i) -> (EQ (pred (suc i)) i) )

( 1 = (suc 0) )
( 2 = (suc 1) )
( 3 = (suc 2) )

( Verify (NzNat 2) )
verification succeeded

( Red (pred 2) )
( suc 0 )

```

Notice that `pred` is the representation of a partial set-theoretic function which is undefined for 0. In our specification this is reflected by the fact that `(pred 0)`

is an element of the type `Nat?`, but not an element of the subsort represented by `Nat`. In fact, we are concerned with a general approach, which has already been employed in the context of membership equational logic (and earlier in OBJ3, see [GM93a]), to explicitly represent partial set-theoretic functions.

A slightly more involved example which nicely demonstrates the possible interaction between computational membership and equational axioms and also involves structural axioms is the example of paths in a graph that been given in Section 2.4.4 and can be represented in OCC as follows:

```
( Path? : Type )
( Node? : Type )

( Edge : Path? -> Prop )
( Path : Path? -> Prop )
( Node : Node? -> Prop )

( edge_subsort_path_as : ?? {p : Path?} (Edge p) -> (Path p) )

( concat : Path? -> Path? -> Path? )

( concat_assoc : !! {p1,p2,p3 : Path?}
  (EQ (concat p1 (concat p2 p3)) (concat (concat p1 p2) p3)) )

( source : Path? -> Node? )
( target : Path? -> Node? )

( source_as : ?? {p : Path?} (Path p) -> (Node (source p)) )
( target_as : ?? {p : Path?} (Path p) -> (Node (target p)) )

( length : Path? -> Nat? )
( length_as : ?? {p : Path?} (Path p) -> (Nat (length p)) )

( concat_as : ?? {E : Path?}{P : Path?}
  (Edge E) -> (Path P) ->
  (EQ (target E) (source P)) -> (Path (concat E P)) )

( source_eq : !! {E : Path?}{P : Path?}
  (Edge E) -> (Path P) -> (Path (concat E P)) ->
  (EQ (source (concat E P)) (source E)) )

( target_eq : !! {E : Path?}{P : Path?}
  (Edge E) -> (Path P) -> (Path (concat P E)) ->
  (EQ (target (concat P E)) (target E)) )
```

```
( length_eq_1 : !! {E : Path?}
  (Edge E) -> (EQ (length E) 1) )

( length_eq_2 : !! {E : Path?}{P : Path?}
  (Edge E) -> (Path P) -> (EQ (target E) (source P)) ->
  (EQ (length (concat E P)) (suc (length P))) )
```

In the following we fix a concrete graph and give some example computations in the extended context:

```
( n1 : Node? ) ( node_n1_as : ?? (Node n1) )
( n2 : Node? ) ( node_n2_as : ?? (Node n2) )
( n3 : Node? ) ( node_n3_as : ?? (Node n3) )
( n4 : Node? ) ( node_n4_as : ?? (Node n4) )

( a : Path? ) ( edge_a_as : ?? (Edge a) )
( b : Path? ) ( edge_b_as : ?? (Edge b) )
( c : Path? ) ( edge_c_as : ?? (Edge c) )
( d : Path? ) ( edge_d_as : ?? (Edge d) )

( source_a_eq : !! (EQ (source a) n1) )
( target_a_eq : !! (EQ (target a) n2) )
( source_b_eq : !! (EQ (source b) n1) )
( target_b_eq : !! (EQ (target b) n3) )
( source_c_eq : !! (EQ (source c) n3) )
( target_c_eq : !! (EQ (target c) n4) )
( source_d_eq : !! (EQ (source d) n4) )
( target_d_eq : !! (EQ (target d) n2) )

( Red (length (concat (concat b c) d)) )
( suc ( suc ( suc 0 ) ) )

( Verify (Path (concat (concat a b) c)) )
verification failed

( Red (length (concat (concat a b) c)) )
( length ( concat ( concat a b ) c ) )
```

General Equational Specifications

In the representation above the sorts of MEL are regarded as unary predicates. It is however worth to point out that OCC directly supports equational logic in

its general form, i.e. Horn clause logic with equality and arbitrary predicates. The operational semantics is again based on an exhaustive goal-oriented search in combination with reduction by computational equations. For instance, the following examples show the use of typical binary predicates on natural numbers and their operational behavior.

```
( nat_le : nat -> nat -> Prop )

( nat_le_1 : ?? {j : nat}
  (nat_le 0 j) )
( nat_le_2 : ?? {i,j : nat}
  (nat_le i j) -> (nat_le (suc i) (suc j)) )

( Verify (nat_le 2 3) )
verification succeeded

( nat_max : nat -> nat -> nat )

( nat_max_1 : !! {i,j : nat}
  (nat_le i j) -> (EQ (nat_max i j) j) )
( nat_max_2 : !! {i,j : nat}
  (nat_le j i) -> (EQ (nat_max i j) i) )

( Red (nat_max 2 3) )
( suc ( suc ( suc 0 ) ) )
```

Beyond equational logic, OCC admits propositional functions such as `Not` in the following example, which in a positive way introduces equality on `nat` and its negation as well as an order on `nat` given by `nat_le` and `nat_lt`.

```
( Not : Prop -> Prop)

( nat_eq : nat -> nat -> Prop )

( nat_eq_1 : ??
  (nat_eq 0 0) )
( nat_eq_2 : ?? {i,j : nat}
  (nat_eq i j) -> (nat_eq (suc i) (suc j)) )

( not_nat_eq_1 : ?? {j : nat}
  (Not (nat_eq 0 (suc j))) )
( not_nat_eq_2 : ?? {j : nat}
  (Not (nat_eq (suc j) 0)) )
```

```
( not_nat_eq_3 : ?? {i,j : nat}
  (Not (nat_eq i j)) -> (Not (nat_eq (suc i) (suc j))) )

( nat_lt : nat -> nat -> Prop )

( nat_lt_1 : ?? {i,j : nat}
  (nat_le i j) -> (Not (nat_eq i j)) -> (nat_lt i j) )

( Verify (nat_lt 2 3) )
verification succeeded
```

We can further introduce propositional functions

```
( And : Prop -> Prop -> Prop )

( And_intro : ?? { A,B : Prop } A -> B -> (And A B) )

( Or : Prop -> Prop -> Prop )

( Or_intro_l : ?? { A,B : Prop } A -> (Or A B) )
( Or_intro_r : ?? { A,B : Prop } B -> (Or A B) )
```

and write specifications such as the following:

```
( person : Type )

( peter : person )
( paul : person )
( ana : person )
( james : person )

( male : person -> Prop )
( female : person -> Prop )
( parent_of : person -> person -> Prop )
( mother_of : person -> person -> Prop )
( father_of : person -> person -> Prop )

( fact_1 : ?? (male james) )
( fact_2 : ?? (male paul) )
( fact_3 : ?? (female ana) )
( fact_4 : ?? (male peter) )
( fact_5 : ?? (father_of peter paul) )
```

```

( fact_6 : ?? (mother_of ana paul) )
( fact_7 : ?? (parent_of james peter) )
( fact_8 : ?? {A,B : person}
  (Or (father_of A B) (mother_of A B)) -> (parent_of A B) )
( fact_9 : ?? {A,B : person}
  (father_of A B) -> (Not (father_of B A)) )
( fact_10 : ?? {A,B : person}
  (mother_of A B) -> (Not (mother_of B A)) )
( fact_11 : ?? {A,B : person}
  (And (Not (father_of A B)) (Not (mother_of A B))) ->
  (Not (parent_of A B)) )
( fact_12 : ?? {A,B : person}
  (Not (female A)) -> (Not (mother_of A B)) )
( fact_13 : ?? {A,B : person}
  (Not (male A)) -> (Not (father_of A B)) )
( fact_14 : ?? {A : person}
  (male A) -> (Not (female A)) )
( fact_15 : ?? {A : person}
  (female A) -> (Not (male A)) )

( Verify (parent_of paul peter) )
verification failed
( Verify (Not (parent_of paul peter)) )
verification succeeded

```

In principle, we could impose structural or computational equations on propositions, and hence introduce propositional functions which are not just constructors as in the given examples. Computation with propositions is also an essential ingredient in the quite general form of higher-order equational specifications that are introduced after the next section. First, however, we explain the use of assertional equality, a distinguished binary predicate, which has an operational behavior quite different from the structural and computational equality we have employed so far.

Computational vs. Assertional Equality

OCC supports three different levels of equality with different operational properties: Structural equality is directly implemented by the term representation itself, computational equality is used for reduction (modulo structural equality), and assertional equality is a binary predicate and is like other predicates verified by exhaustive goal-oriented search.

To demonstrate the use of assertional equality in the specification of data types

we introduce the enumeration type `id` which contains precisely the identifiers `a`, `b`, `c`, `d`, `e`, and `i`. According to the convention introduced earlier we have an implicit initiality constraint for `id` with its elements. We only make explicit the implied disequality properties, since these are computationally relevant in the following.

```
( id : Type )

( a : id ) ( b : id ) ( c : id ) ( d : id ) ( e : id ) ( i : id )

( not_eq_a_b : ?? (Not (EQ a b)) )
( not_eq_a_c : ?? (Not (EQ a c)) )
( not_eq_a_d : ?? (Not (EQ a d)) )
...
```

In an asymmetric way, we now introduce an alias for `a` denoted by `a'` by saying that `a'` is computationally equal to `a`. We can consider `a'` as being defined by `a`. The asymmetry is caused by the fact that computational equality has a clear direction. Since `a'` reduces to `a`, properties of `a` are implicitly inherited by `a'`.

```
( a' : id )

( eq_a'_a : !! (EQ a' a) )
```

We can also introduce another alias `a''` in a symmetric way using assertional equality `EQ`. In this case we should explicitly state the properties that are inherited from `a` such as the equalities and nonequalities below.

```
( a'' : id )

( eq_a''_a : ?? (EQ a'' a) )
( not_eq_a''_b : ?? (Not (EQ a' b)) )
( not_eq_a''_c : ?? (Not (EQ a' c)) )
( not_eq_a''_d : ?? (Not (EQ a' d)) )
...
```

The rationale for using a computational or an assertional equality instead of a structural equality in some cases is that each implementation supports only certain forms of structural equations. The present OCC prototype supports exactly the forms of structural equations admitted by Maude, namely commutativity, associativity and the laws of left and right identities. In principle, it is possible to support further structural equations as long as they are of a form that can be

efficiently implemented by the choice of a suitable term representation for which matching algorithms are available.

Another important difference is that computational and assertional equality differ in their built-in closure properties. Computational equality is closed under contexts and transitivity, reflecting its purpose to be used for reduction in arbitrary contexts:

```
( f : id -> id )
( Red (f a') )
( f a )
```

Assertional equality on the other hand is closed under reflexivity and symmetry, but not under transitivity, since, as we pointed out, a transitivity rule would not be not syntax-directed and hence not suitable for exhaustive goal-oriented search.

```
( Verify (EQ (f a'') (f a)) )
verification succeeded
( Verify (EQ (f a) (f a'')) )
verification succeeded
```

Since goal-oriented search is combined with reduction, assertional equality subsumes computational equality so that we have:

```
( Verify (EQ (f a') (f a)) )
verification succeeded
```

Apart from the advantage of assertional equality to be less rigid than structural equality, another reason for sometimes preferring assertional equality over structural or computational equality is that both equality and nonequality can be specified as assertions, i.e. on the basis of the same operational mechanism. Furthermore, equality often comes together with other relations (e.g. orderings), which can also be specified as assertions in this way.

Higher-Order Equational Specifications

As an important step beyond equational logic OCC supports higher-order equational specifications such as the following specification of lists over natural numbers. We first introduce the type `list_nat` together with constructors `nil` and `cons` and a conventional first-order equational specification of `append`.

```

( list_nat : Type )

( nil : list_nat )
( cons : nat -> list_nat -> list_nat )

( append : list_nat -> list_nat -> list_nat )

( list_append_eq_1 : !! {l : list_nat}
  (EQ (append nil l) l) )

( list_append_eq_2 : !! {l : list_nat}{h : nat}{t : list_nat}
  (EQ (append (cons h t) l) (cons h (append t l)))) )

```

By three examples we demonstrate different uses of higher-order equational specifications. The function `list_map` is higher-order in the sense that it has a functional argument, and the functions `list_in` and `list_select` are higher-order in the sense that they have predicates as arguments. Furthermore, `list_in` is itself a (higher-order) predicate, in contrast to `list_map` and `list_select`.

The higher-order function `list_map` applies a given function to all elements of a given list, returning the list of results.

```

( list_map : (nat -> nat) -> list_nat -> list_nat )

( list_map_eq_1 : !! {f : (nat -> nat)}
  (EQ (list_map f nil) nil) )

( list_map_eq_2 : !! {f : (nat -> nat)}{x : nat}{l : list_nat}
  (EQ (list_map f (cons x l)) (cons (f x) (list_map f l)))) )

( Red (list_map suc (cons 0 (cons 1 (cons 2 nil)))) )
( cons ( suc 0 ) ( cons ( suc ( suc 0 ) )
  ( cons ( suc ( suc ( suc 0 ) ) ) nil ) ) ) )

```

The higher-order predicate `list_in` takes an equivalence predicate²¹ and verifies if a given element is a member of the list using the given equality.

```

( list_in : (nat -> nat -> Prop) -> list_nat -> nat -> Prop )

```

²¹By means of a stronger type, it would be possible to express that this function accepts only predicates that are reflexive, symmetric, and transitive. One possibility is to use the algebras-as-objects approach of Section 8.2.7.

```

( list_in_eq_1 : ?? {eq : nat -> nat -> Prop}
  {l : list_nat}{x,y : nat}
  (eq x y) -> (list_in eq (cons x l) y) )

( list_in_eq_2 : ?? {eq : nat -> nat -> Prop}
  {l : list_nat}{x,y : nat}
  (list_in eq l y) -> (list_in eq (cons x l) y) )

( Verify (list_in eq (cons 0 (cons 1 (cons 2 nil)))) 1) )
verification succeeded

```

The function `list_select` filters the given list according to a given predicate, returning only the elements satisfying the predicate.

```

( list_select : (nat -> Prop) -> list_nat -> list_nat )

( list_select_eq_1 : !! {P : (nat -> Prop)}
  (EQ (list_select P nil) nil) )

( list_select_eq_2 : !! {P : (nat -> Prop)}
  {x : nat}{l : list_nat} (P x) ->
  (EQ (list_select P (cons x l)) (cons x (list_select P l)))) )

( list_select_eq_3 : !! {P : (nat -> Prop)}
  {x : nat}{l : list_nat} (Not (P x)) ->
  (EQ (list_select P (cons x l)) (list_select P l)) )

( Red (list_select ([n : nat] (eq n 1))
  (cons 1 (cons 2 (cons 1 nil)))) )
( cons ( suc 0 ) ( cons ( suc 0 ) nil ) )

```

The specification of `list_select` nicely demonstrates the interplay between the two computational mechanisms of reduction and goal-oriented search.

Equationally Defined Datatypes

Several standard data types such as lists, sets, and multisets, can be naturally specified using structural equations of a simple form. We demonstrate this issue using the following alternative specification of lists which uses `nil`, `single` and `append` as constructors.

```

( list_nat : Type )

```

```

( nil : list_nat )

( single : nat -> list_nat )

( append : list_nat -> list_nat -> list_nat )

( append_assoc : || {l1,l2,l3 : list_nat}
  (EQ (append l1 (append l2 l3)) (append (append l1 l2) l3)) )

( append_right_id : || {l : list_nat}
  (EQ (append l nil) l) )

( append_left_id : || {l : list_nat}
  (EQ (append nil l) l) )

```

The functions `list_map`, `list_select` and `list_in` could be specified as before (based on a decomposition of lists in head and tail), but we structure their specification according to the three new constructors. We first introduce an auxiliary predicate to verify emptiness of lists.

```

( list_is_empty : list_nat -> Prop )

( list_is_empty_as : ?? (list_is_empty nil) )

( list_not_is_empty_as : ?? {l : list_nat}{u : nat}
  (Not (list_is_empty (append l (single u)))) )

```

The specification of `list_map` takes now the following form:

```

( list_map : (nat -> nat) -> list_nat -> list_nat )

( list_map_eq_1 : !! {f : (nat -> nat)}
  (EQ (list_map f nil) nil) )

( list_map_eq_2 : !! {f : (nat -> nat)}{x : nat}
  (EQ (list_map f (single x)) (single (f x))) )

( list_map_eq_3 : !! {l,l' : list_nat}{f : (nat -> nat)}
  (Not (list_is_empty l)) -> (Not (list_is_empty l')) ->
  (EQ (list_map f (append l l'))) )

```

```

      (append (list_map f l) (list_map f l')))) )

( Red (list_map suc
      (append (single 0) (append (single 1) (single 2)))) )
( append ( append ( single ( suc 0 ) ) )
          ( single ( suc ( suc 0 ) ) )
          ( single ( suc ( suc ( suc 0 ) ) ) ) ) )

```

The function `list_in` is specified by:

```

( list_in : (nat -> nat -> Prop) -> list_nat -> nat -> Prop )

( list_in_as_1 : ?? {eq : (nat -> nat -> Prop)}
  {l : list_nat}{x,y : nat}
  (eq x y) -> (list_in eq (append l (single x)) y) )

( list_in_as_2 : ?? {eq : (nat -> nat -> Prop)}
  {l,l' : list_nat}{x : nat}
  (Not (list_is_empty l)) -> (Not (list_is_empty l')) ->
  (list_in eq l x) -> (list_in eq (append l l') x) )

( list_in_as_3 : ?? {eq : (nat -> nat -> Prop)}
  {l,l' : list_nat}{x : nat}
  (Not (list_is_empty l)) -> (Not (list_is_empty l')) ->
  (list_in eq l' x) -> (list_in eq (append l l') x) )

( Verify (list_in eq
  (append (single 0) (append (single 1) (single 2))) 1) )
verification succeeded

```

Finally, the function `list_select` has the following specification:

```

( list_select : (nat -> Prop) -> list_nat -> list_nat )

( list_select_eq_1 : !! {P : (nat -> Prop)}
  (EQ (list_select P nil) nil) )

( list_select_eq_2 : !! {P : (nat -> Prop)}{x : nat}
  (P x) -> (EQ (list_select P (single x)) (single x)) )

( list_select_eq_3 : !! {P : (nat -> Prop)}{x : nat}
  (Not (P x)) -> (EQ (list_select P (single x)) nil) )

```

```
( list_select_eq_4 : !! {l,l' : list_nat}{P : (nat -> Prop)}
  (Not (list_is_empty l)) -> (Not (list_is_empty l')) ->
  (EQ (list_select P (append l l'))
      (append (list_select P l) (list_select P l')))) )

( Red (list_select ([n : nat] (eq n 1))
  (append (single 1) (append (single 2) (single 1)))) )
( append ( single ( suc 0 ) ) ( single ( suc 0 ) ) ) )
```

Above we have employed an auxiliary predicate `list_is_empty` and its negation to make the specification terminating. If we omit the corresponding premises in say `list_map_eq_3` the equation could be instantiated to

```
( list_map_eq_3 : !! {f : (nat -> nat)}
  (EQ (map f nil) (append (map f nil) (map f nil)))) )
```

leading to an undesired source of nontermination.

In a similar way, we specify finite multisets below. The functions `list_map`, `list_in` and `list_select` can be introduced as above and are omitted here.

```
( fms_nat : Type )

( fms_empty : fms_nat )

( fms_single : nat -> fms_nat )

( fms_union : fms_nat -> fms_nat -> fms_nat )

( fms_union_comm : || {m,m' : fms_nat}
  (EQ (fms_union m m') (fms_union m' m)) )

( fms_union_assoc : || {m,m',m'' : fms_nat}
  (EQ (fms_union m (fms_union m' m''))
      (fms_union (fms_union m m') m'')) )

( fms_union_id : || {m : fms_nat}
  (EQ (fms_union m fms_empty) m) )
```

Instead of using structural equations we could also employ computational equations, e.g., we could replace the previous structural equation for the identity element by the corresponding computational equation below. Of course, we loose

matching power, but for many purposes matching modulo the identity laws is not needed. Sometimes it is even advisable to avoid it, since it is a potential source of nontermination.

```
( fms_union_right_id : !! {m : fms_nat}
  (EQ (fms_union m fms_empty) m) )
```

An operationally meaningful equality on data types can not only be given by structural and computational equations but also by assertional equality. Below we develop a specification of finite sets over the type `id` introduced above. It shows how to specify finite sets over a type where logical equality of ground terms does not coincide with structural equality but is given by a coarser assertional equality.

```
( fs_id : Type )

( fs_empty : fs_id )

( fs_single : id -> fs_id )

( fs_union : fs_id -> fs_id -> fs_id )

( fs_union_comm : || {s,s' : fs_id}
  (EQ (fs_union s s') (fs_union s' s)) )

( fs_union_assoc : || {s,s',s'' : fs_id}
  (EQ (fs_union s (fs_union s' s''))
    (fs_union (fs_union s s') s'')) )

( fs_union_id : || {s : fs_id}
  (EQ (fs_union s fs_empty) s) )
```

To specify idempotence of `fs_union` we add a computational equation such as:

```
( fs_union_idemp : !! {x : id}
  (EQ (fs_union (fs_single x) (fs_single x)) (fs_single x)) )

( Red (fs_union (fs_single a)
  (fs_union (fs_single b) (fs_single a))))
( fs_union ( fs_single a ) ( fs_single b ) )
```

A general disadvantage of using linear patterns as `(fs_union (fs_single x) (fs_single x))` in `fs_union_idemp` is that the terms substituted for the two

occurrences of x are checked only for structural equality, which is coarser than assertional equality EQ. This is fine for datatypes with an equality that coincides with structural equality, but not for our example of `id` where we identify `a` and `a''` using assertional equality. For instance, the following term is irreducible:

```
( Red (fs_union (fs_single a)
              (fs_union (fs_single b) (fs_single a''))) )
( fs_union (fs_single a)
  ( fs_union (fs_single b) (fs_single a'') ) )
```

To enhance the operational behavior we use the following operationally more general but logically equivalent equation:

```
( fs_union_idemp : !! {x,y : id} (EQ x y) ->
  (EQ (fs_union (fs_single x) (fs_single y)) (fs_single x)) )

( Red (fs_union (fs_single a)
              (fs_union (fs_single b) (fs_single a''))) )
( fs_union ( fs_single a ) ( fs_single b ) )
( fs_union ( fs_single b ) ( fs_single a'' ) )
```

The last evaluation has the two possible results shown above (apart from variants modulo structural equality) from which only one is actually computed. The results are not structurally equal, but they are assertionally equal which can be verified as follows:

```
( Verify (EQ (fs_union (fs_single a) (fs_single b))
             (fs_union (fs_single b) (fs_single a''))) )
verification succeeded
```

An alternative to the use of idempotence as a computational equation is to omit this axiom, i.e. to work with a less abstract representation, and to define a coarser assertional equality on finite sets, which can be done as follows using an auxiliary predicate for emptiness. Recall that we have used a similar technique in our specification of lists to avoid problems with nontermination.

```
( fs_is_empty : fs_id -> Prop )

( fs_is_empty_as : ?? (fs_is_empty fs_empty) )

( fs_not_is_empty_as : ?? {s : fs_id}{u : id}
  (Not (fs_is_empty (fs_union s (fs_single u)))) )
```

We next specify a set membership function `fs_in` and its negation. The function `fs_is_empty` is used in some premises to avoid nontermination in way similar to the specification of lists.

```
( fs_in : fs_id -> id -> Prop )

( fs_in_as_1 : ?? {s : fs_id}{x,y : id}
  (EQ x y) -> (fs_in (fs_union s (fs_single x)) y) )

( fs_in_as_2 : ?? {s,s' : fs_id}{x : id}
  (Not (fs_is_empty s)) -> (Not (fs_is_empty s')) ->
  (fs_in s' x) -> (fs_in (fs_union s s') x) )

( fs_not_in_as_1 : ?? {x : id}
  (Not (fs_in fs_empty x)) )

( fs_not_in_as_2 : ?? {x,y : id}
  (Not (EQ x y)) -> (Not (fs_in (fs_single y) x)) )

( fs_not_in_as_3 : ?? {s,s' : fs_id}{x : id}
  (Not (fs_is_empty s)) -> (Not (fs_is_empty s')) ->
  (Not (fs_in s x)) -> (Not (fs_in s' x)) ->
  (Not (fs_in (fs_union s s') x)) )
```

Now we define the standard ordering on sets, i.e. the subset predicate, and use to define set equality in the standard way.

```
( fs_le : fs_id -> fs_id -> Prop )

( fs_le_as_1 : ?? {s : fs_id}
  (fs_le fs_empty s) )

( fs_le_as_2 : ?? {s,s' : fs_id}{x,y : id}
  (fs_le s s') -> (EQ x y) ->
  (fs_le (fs_union (fs_single x) s) (fs_union (fs_single y) s')) )

( fs_le_as_3 : ?? {s : fs_id}{x : id}
  (Not (fs_in s x)) -> (fs_le s (fs_union (fs_single x) s)) )

( fs_eq : fs_id -> fs_id -> Prop )
```

```
( fs_eq_as : ?? {s,s' : fs_id}
  (fs_le s s') -> (fs_le s' s) -> (fs_eq s s') )
```

Finally, we identify sets which are equivalent in terms of `fs_eq`.

```
( fs_eq_imp_EQ : ?? {s,s' : fs_id}
  (fs_eq s s') -> (EQ s s') )

( Verify (EQ (fs_union (fs_single a) (fs_single b))
             (fs_union (fs_single b) (fs_single a''))) )
verification succeeded
```

In this way we obtain a specification of finite sets with the same logical equality as before, but with a different representation and operational behavior.

We conclude this section with the following specification of the cardinality of finite sets, which depends on the equality of its elements in an essential way:

```
( fs_card : fs_id -> nat )

( fs_card_eq_1 : !!
  (EQ (fs_card fs_empty) 0) )

( fs_card_eq_2 : !! {x : id}
  (EQ (fs_card (fs_single x)) 1) )

( fs_card_eq_3 : !! {x : id}{s : fs_id}
  (Not (fs_in s x)) ->
  (EQ (fs_card (fs_union (fs_single x) s)) (suc (fs_card s))) )

( fs_card_eq_4 : !! {x : id}{s : fs_id}
  (fs_in s x) ->
  (EQ (fs_card (fs_union (fs_single x) s)) (fs_card s)) )

( Red (fs_card (fs_union (fs_single a)
                        (fs_union (fs_single b) (fs_single a'')))) )
( suc ( suc 0 ) )
```

Notice that the premise `(Not (fs_in m x))` in `fs_card_eq_3` cannot be dropped, since even if we would add the computational idempotence equation `union_idemp`, we cannot assume that finite sets are always in normal form, i.e. containing exactly one occurrence of the form `fs_single` for each element.

8.2.2 Dependent Types and Universes

Type Operators and Polymorphism

The datatypes of lists, finite multisets, and finite sets given earlier are examples of parameterized datatypes that have been instantiated by `nat` or `id`. The availability of dependent types and universes in OCC allows us to naturally express type operators and explicitly polymorphic functions. The following is a specification of the parameterized inductive datatype of lists given by a type operator `list` with explicitly polymorphic constructors `nil` and `cons`. By our convention, there is an implicit freeness constraint stating that the specification of `(list T)` is free over `T` (see Section 8.2.7 for details).

```
( list : Type -> Type )

( nil : {T : Type} (list T) )

( cons : {T : Type} T -> (list T) -> (list T) )
```

We also specify `list_map` as an example of an equationally defined explicitly polymorphic function.

```
( list_map : {U,V : Type} (U -> V) -> (list U) ->(list V) )

( list_map_eq_1 : !! {U,V : Type}{f : (U -> V)}
  (EQ (list_map U V f (nil U)) (nil V)) )

( list_map_eq_2 : !! {U,V : Type}{f : (U -> V)}
  {x : U}{l : (list U)}
  (EQ (list_map U V f (cons U x l))
    (cons V (f x) (list_map U V f l)))) )
```

In the OCC prototype explicit type parameters can often be abbreviated as `?`. Each occurrence of `?` introduces a new existential metavariable that has to be solved by the system.

```
( list_map_eq_1 : !! {U,V : Type}{f : (U -> V)}
  (EQ (list_map ? ? f (nil U)) (nil V)) )

( list_map_eq_2 : !! {U,V : Type}{f : (U -> V)}
  {x : U}{l : (list U)}
  (EQ (list_map ? ? f (cons U x l))
    (cons ? (f x) (list_map ? ? f l)))) )
```

Similar to the approach adopted in LEGO [Pol90], arguments of functions can be specified to be implicit. For instance, the following specification of lists is equivalent to the previous one. The *implicitly dependent type* $\{x \mid A\} B$ is syntactic sugar for the (explicitly) dependent type $\{x : A\} B$, and denotes the type of functions with a result of type B and an implicit argument of type A . In addition, each application $(M P)$ of a function M of type $\{x \mid A\} B$ is syntactic sugar for $((M ?) P)$, that is by default, implicit arguments in function applications are omitted and inferred using type inference whenever possible. As an exception, the previously implicit argument N can be explicitly given using the function application $(M \mid N)$, which is syntactic sugar for $(M N)$ in this case.

```
( list : Type -> Type )

( nil : {T | Type} (list T) )

( cons : {T | Type} T -> (list T) -> (list T) )

( list_map : {U,V | Type} (U -> V) -> (list U) -> (list V) )

( list_map_eq_1 : !! {U,V : Type}{f : (U -> V)}
  (EQ (list_map f (nil | U)) (nil | V)) )

( list_map_eq_2 : !! {U,V : Type}{f : (U -> V)}
  {x : U}{l : (list U)}
  (EQ (list_map f (cons x l))
      (cons (f x) (list_map f l)))) )
```

Another example of a parameterized nonrecursive datatype is the cartesian product type which can be specified as follows. Here `prod` is a binary type operator, and there is only one polymorphic constructor `pair`.

```
( prod : Type -> Type -> Type )

( pair : {U,V | Type } U -> V -> (prod U V) )

( fst : {U,V | Type } (prod U V) -> U )

( fst_eq : {U,V : Type}{u : U}{v : V}
  (EQ (fst (pair u v)) u) )

( snd : {U,V | Type} (prod U V) -> V )
```

```

( snd_eq : {U,V : Type}{u : U}{v : V}
  (EQ (snd (pair u v)) v) )

( pair_not_eq : ?? {U,V : Type}{u,u' : U}{v,v' : V}
  (Or (Not (EQ u u')) (Not (EQ v v')))) ->
  (Not (EQ (pair u v) (pair u' v'))) )

```

We will also need the parameterized disjoint sum type `sum` which is generated by its constructors `left` and `right`. The function `decide` is used to inspect elements of this type by case analysis.

```

( sum : Type -> Type -> Type )

( left : {U,V | Type } U -> (sum U V) )

( right : {U,V | Type} V -> (sum U V) )

( decide : {U,V | Type}{T | Type}
  (U -> T) -> (V -> T) -> (sum U V) -> T )

( decide_eq_1 : !! {U,V : Type}{T : Type}
  {f : U -> T}{g : V -> T}{u : U}
  (EQ (decide f g (left u)) (f u)) )

( decide_eq_2 : !! {U,V : Type}{T : Type}
  {f : U -> T}{g : V -> T}{v : V}
  (EQ (decide f g (right v)) (g v)) )

```

In an entirely analogous way we generalize our specifications of lists, finite multisets and finite sets (the versions with structural equations) to parameterized data types.

In summary, OCC supports (explicitly) dependent types and universes, but the OCC prototype additionally implemented a concept of implicitly dependent types similar to that known from LEGO, which as explained in [Pol90] can be regarded as syntactic sugar that can always be eliminated, and is just used to increase readability. In contrast to the quite liberal approach of LEGO, however, we designate arguments of functions as implicit only in the case where this can be justified by semantic means, i.e. where the ultimate meaning of a suitably complete function application does not depend on its implicit arguments. We think that it would be desirable and leave it as future work to make explicit this restricted notion of implicitly dependent types in the formal system and in the semantics of OCC, in order to justify, for instance, the erasure of implicit arguments for computational purposes.

General Dependent Types

In the previous example `list` is a type operator. More generally, dependent types together with universes allows us to express types parameterized not only by types but by arbitrary elements. A good example is a type of vectors, i.e. a type of lists of a given length over a given type, which can be specified as follows. Due to the additional information available, the function `list_map` can be equipped with a more specific type (compared with the previous version) which expresses preservation of the length of its argument list.

```
( list : Type -> nat -> Type )

( nil : {T : Type} (list T 0) )

( cons : {T : Type}{n : nat}
  T -> (list T n) -> (list T (suc n)) )

( list_map : {U,V : Type}{n : nat}
  (U -> V) -> (list U n) -> (list V n) )

( list_map_eq_1 : !! {U,V : Type}{f : (U -> V)}
  (EQ (list_map U V 0 f (nil U)) (nil V)) )

( list_map_eq_2 : !! {U,V : Type}{n : nat}{f : (U -> V)}
  {x : U}{l : (list U n)}
  (EQ (list_map U V (suc n) f (cons U n x l))
    (cons V n (f x) (list_map U V n f l))) )
```

Again we can employ implicitly dependent types and obtain the following equivalent but more readable specification:

```
( list : Type -> nat -> Type )

( nil : {T | Type} (list T 0) )

( cons : {T | Type}{n | nat}
  T -> (list T n) -> (list T (suc n)) )

( list_map : {U,V | Type}{n | nat}
  (U -> V) -> (list U n) ->(list V n) )

( list_map_eq_1 : !! {U,V : Type}{f : (U -> V)}
  (EQ (list_map f (nil | U)) (nil | V)) )
```

```
( list_map_eq_2 : !! {U,V : Type}{n : nat}{f : (U -> V)}
  {x : U}{l : (list U n)}
  (EQ (list_map f (cons x l))
    (cons (f x) (list_map f l)))) )
```

Type Classes and Inheritance

The functional programming language Haskell is well-known for its type system which extends the Hindley-Milner system by type classes [WB89, HHJW96] and certain forms of qualified types as they are called in [Jon92b, Jon92a]. Type classes are a means to express common features of types, and qualified types of the form $\Gamma \Rightarrow T$ can express that a polymorphic type T , i.e. a type involving type variables, is subject to certain assumptions Γ , which restrict the type variables of T to range over certain classes of types. Inheritance is provided in the sense that type classes can be defined in terms of existing type classes by inheriting their features and possibly requiring additional ones.

In OCC we can express type classes as predicates over a type universe and inheritance using Horn clauses with operational meaning. Qualified types can then be regarded as a special case of dependent types. The following example specifies two type classes `has_eq` and `has_order`, where `has_order` has more features and hence is a subclass of `has_eq`.

```
( has_eq : Type -> Prop )
( has_order : Type -> Prop )

( subclass_as : ?? {T : Type} (has_order T) -> (has_eq T) )

( eq : {T | Type} {e | (has_eq T)} T -> T -> Prop )
( le : {T | Type} {e | (has_order T)} T -> T -> Prop )
```

An instance of both of these classes is `nat` together with its equality and order as given below. Notice that the assertional axiom `?? (has_order nat)` is essential to typecheck the occurrences of both `eq` and `le`.

```
( nat_has_order : ?? (has_order nat) )

( eq_as_0 : ?? (eq 0 0) )
( eq_as_1 : ?? {i,j : nat} (eq i j) -> (eq (suc i) (suc j)) )

( le_as_1 : ?? {j : nat} (le 0 j) )
( le_as_2 : ?? {i,j : nat} (le i j) -> (le (suc i) (suc j)) )
```

```
( Verify (le 2 3))
verification succeeded
```

```
( Verify (eq 2 2))
verification succeeded
```

A polymorphic version of list membership which can be instantiated for all equality types can be specified as follows. Again, notice that the assertion `?? (has_eq T)` is needed to typecheck the occurrences of `eq` and `list_in`.

```
( list_in : {T | Type}{e | (has_eq T)} (list T) -> T -> Prop )
```

```
( list_in_as_1 : ?? {T : Type}{e : ?? (has_eq T)}
  {l : (list T)}{x,y : T}
  (eq x y) -> (list_in (cons x l) y) )
```

```
( list_in_as_2 : ?? {T : Type}{e : ?? (has_eq T)}
  {l : (list T)}{x,y : T}
  (list_in l y) -> (list_in (cons x l) y) )
```

```
( nat_has_eq : ?? (has_eq nat) )
```

```
( Verify (list_in (cons 0 (cons 1 (cons 2 nil))) 1) )
verification succeeded
```

Observe that it is the interplay between dependent types, type universes and operational Horn clauses which makes this representation of type classes and qualified types possible in OCC.

Multiple Parameter Classes and Type Relations

The introduction of typeclasses in [WB89] and their success in practice, especially via their inclusion in the Haskell language, has initiated several directions of research including multiple parameter classes [CHO92] and type relations [Jon92b, Jon92a]. See also [JJM97] for a survey of different extensions and their pragmatic motivations. All these approaches can be naturally represented in OCC. Type relations, for instance, can be represented as predicates of the type `Type -> ... -> Type -> Prop`.

Another recent generalization are type classes with functional dependencies as proposed in [Jon00]. A similar way of representing functional dependencies is possible via higher-order type predicates in OCC. These are type predicates which

may have type operators as arguments as we can see in the following specification of abstract collections:

```
( collection : (Type -> Type) -> Type -> Prop )

( col_empty : {C | Type -> Type}{A | Type}{p | (collection C A)}
  (C A) )

( col_insert : {C | Type -> Type}{A | Type}{p | (collection C A)}
  A -> (C A) -> (C A) )

( col_union : {C | Type -> Type}{A | Type}{p | (collection C A)}
  (C A) -> (C A) -> (C A) )
```

An instance of this generalized type class is given by the type `list` introduced previously:

```
( list_is_collection : ?? {A : Type} (collection list A) )

( col_empty_eq : !! {A : Type}
  (EQ (col_empty | list | A | ?) (nil | A)) )

( col_insert_eq : !! {A : Type}{x : A}{l : (list A)}
  (EQ (col_insert x l) (cons | A x l)) )

( col_union_eq : !! {A : Type}{l,l' : (list A)}
  (EQ (col_union l l') (append l l')) )

( Red (col_insert 0 (col_empty | list | nat | ?)) )
( cons 0 nil )
```

In summary, we conjecture that OCC can serve as a framework for various approaches to type classes. We furthermore think that this framework could be helpful to find systematic solutions to the problems discussed in [JJM97, Jon00], which have been serious enough to prevent the incorporation of generalizations of type classes into the Haskell language standard.

Type Equality and Type Casting

From the viewpoint of the OCC type checker, computational and assertional equality provide the interface to the underlying computational system. More precisely, type checking, i.e. verifying $M : A'$ in a given context is reduced to

type inference yielding the inferred type of M , say A , followed by a check of assertional equality $A = A'$. As all operational notions, assertional equality gives only an incomplete picture of the logical world. Furthermore, the standard logical equality (see Section 8.2.6) is transitive, but assertional equality is not required to satisfy this property, as we see in the following example.

```
( A : Type )
( B : Type )
( C : Type )

( eq_A_B : ?? (EQ A B) )
( eq_B_C : ?? (EQ B C) )

( Verify (EQ A C) )
verification failed

( a : A )
( c = (a : B) : C )
all goals solved

( c = a : C )
failed
```

Above we use type assertions of the form $M : A$, which can be used to ensure that M has the desired type. Since $(M : A)$ is itself a term of type A , type assertions simultaneously provide a means for controlled type casting.

Enhanced Typing Capabilities

The general problem with very restrictive type theories such as CIC is that the type checker cannot make use of logical facts, which could be either axioms or theorems proved in the type theory. In OCC such facts can be made available to the type checker via the open operational semantics, which itself reflects a part of the logical world. To illustrate the advantages, we give an OCC specification of the module `var_state_ops` from Chapter 4, which clearly shows the difficulties in the CIC approach. Recall that the module `var_state_ops` defines operations for an abstract state space with typed variables, which has been introduced by the module `var_abstract` as follows.

```
( var : Type )

( type_name : Type )
```

```

( type : type_name -> Type )

( var_type_name : var -> type_name )
( var_type = [v : var] (type (var_type_name v)) )

( st = {v : var}(var_type v) )
( act : Type )
( trans : act -> st -> st -> Prop )

( view = act -> Prop )

```

Recall further that the first difficulty in the module `var_state_ops` was the definition of `val_eq`, an equality predicate for values of variables. Although this function is only used to compare values of variables with the same type, this fact cannot be conveyed to the type checker, so that a dependent equality had to be used instead of the standard polymorphic equality of CIC.

```

Definition val_eq :=
  [n:type_name] [x:(type n)] [n':type_name] [x':(type n')]
  (eqT_dep type_name type n x n' x').

```

For similar reasons casting between types poses a problem and requires the use of `eqT_rec` for which the inductive definition of `eqT` does not provide any computation rules.

```

Definition cast := [n,n':type_name] [e:(n==n')] [x:(type n)]
  (eqT_rec type_name n type x n' e).

```

In OCC value equality and casting do not pose any problems. We can use the standard polymorphic equality for value equality, since the type checker takes into account the context, which in our case will always contain operational assertions implying that the types on both sides of the equality are equal. For a similar reason, casting can simply be specified by the following computational equation, which in the given context is well-typed due to the presence of the operational assertion `{e : ?? (EQ n n')}`.

```

( cast : {n,n' : type_name}{e : (EQ n n')}{x : (type n)}(type n') )

( cast_eq : !! {n,n' : type_name}{e : ?? (EQ n n')}{x : (type n)}
  (EQ (cast n n' e x) x) )

```

Similarly, we can replace the cumbersome definition of the `assign` function

```

Definition assign := [s:st][v:var][x:(var_type v)]
  [v':var] <var_type v'>Case (var_eq_dec v v') of
  (*v==v'*) [p:(v==v')] (cast (var_type_name v) (var_type_name v')
    (var_eq_to_type_name_eq v v' p) x)
  (*~(v==v')*) [_](s v')
end.

```

which used `cast` and required us to explicitly pass proofs around by the following much more natural abstract specification. Notice again that well-typedness of $(EQ ((assign\ s\ v\ x)\ v')\ x)$ is ensured by the operational premise $(??\ (EQ\ v\ v'))$.

```

( assign : st -> {v : var}(var_type v) -> st )

( assign_eq_1 : !! {s : st}{v : var}{x : (var_type v)}{v' : var}
  (?? (EQ v v')) -> (EQ ((assign s v x) v') x) )

( assign_eq_2 : !! {s : st}{v : var}{x : (var_type v)}{v' : var}
  (Not (EQ v v')) -> (EQ ((assign s v x) v') (s v')) )

```

Furthermore, OCC allows us to prove the following properties of `assign` from this specification. We can see that most of these theorems (or axioms, depending on the point of view) can be directly used as computational equations.

```

( assign_unchanged : !! {s : st}{v : var}
  (EQ (assign s v (s v)) s) )

( assign_changed : !! {s : st}{v : var}{x : (var_type v)}
  (EQ ((assign s v x) v) x) )

( assign_indep : !! {s : st}{v,v' : var}{x : (var_type v)}
  (Not (EQ v v')) ->
  (EQ ((assign s v x) v') (s v')) )

( assign_twice : !! {s : st}{v : var}{x,y : (var_type v)}
  (EQ (assign (assign s v x) v y) (assign s v y)) )

( assign_commutates : {s : st}
  {v,v' : var}{x : (var_type v)}{y : (var_type v')}
  (Not (EQ v v')) ->
  (EQ (assign (assign s v x) v' y) (assign (assign s v' y) v x)) )

```

In the context of programming and specification, computational equations (and their interplay with structural equations and assertional propositions) are the key for abstract execution, i.e. execution without reference to a concrete interpretation or implementation. In the context of interactive theorem proving, which will be addressed in Section 8.3, abstract execution increases the degree of automation and reduces the complexity of proofs.

Heterogenous Data Structures

Recently the combination of dependent types and universes has been proposed as a powerful foundation for new functional programming languages [Aug99, Pol01]. The first programming language specifically designed to address this use of a dependent type theory is Cayenne [Aug99], a language based on a variant of Martin-Löf's constructive type theory. On the practical side, however, it has been found that programming with dependent types leads to rather complicated formulations, due to the overhead of computing types and passing proofs around. A good illustration of these difficulties, which have been pointed out to us by Randy Pollack, is contained in [Pol01], where different formalizations of an important data structure, namely heterogenous association lists, are presented in LEGO (making essential use of nonstandard computation rules). To sum up, none of the formalizations presented have been found to be very satisfactory, and it seems that programming with dependent types needs fundamentally new ideas to scale up to even more complex problems.

As we already explained, we consider the too conservative character of today's type theories with dependent types as a major source of difficulties. It should be no surprise that a high-level specification is often clearer and more concise than a rather low-level implementation in terms of a purely functional program. Indeed, addressing the particular examples of heterogenous association list in the equational programming style favored by OCC leads to a quite natural formalization of this concept as we can see below.

To simplify the presentation, we consider heterogenous association lists with labels (keys) of the type `id` introduced earlier. The auxiliary predicate `occ` verifies if a given label occurs in a given list. The function `assoc` looks up a given label in a given list and returns the element associated with this label. Similarly, the function `tassoc` returns the type of this elements. A remarkable point is the two-level structure (reflected by `tassoc` and `assoc` respectively) and the similarity in the patterns of their specification. Notice also that in OCC the context-dependent computational system is essential to typecheck the conditional equation `assoc_eq_2` below.

```
( hal : Type )
```

```

( nil : hal )
( cons : {r : id}{A : Type}{s : A}{l : hal} hal )

( occ : {q : id}{l : hal} Prop )

( occ_as_1 : ?? {q,r : id}{A : Type}{a : A}{l : hal}
  (EQ q r) -> (occ q (cons r A a l)) )

( occ_as_2 : ?? {q,r : id}{A : Type}{a : A}{l : hal}
  (Not (EQ q r)) -> (occ q l) -> (occ q (cons r A a l)) )

( tassoc : {n : id}{l : hal}{p : (occ n l)} Type )

( tassoc_eq_1 : !! {n,r : id}{A : Type}{a : A}{l : hal}
  {p : (occ n (cons r A a l))}
  (EQ n r) ->
  (EQ (tassoc n (cons r A a l) p) A) )

( tassoc_eq_2 : !! {n,r : id}{A : Type}{a : A}{l : hal}
  {p : (occ n (cons r A a l))}
  (Not (EQ n r)) -> (?? (occ n l)) ->
  (EQ (tassoc n (cons r A a l) p) (tassoc n l ?)) )

( assoc : {n : id}{l : hal}{p : (occ n l)}(tassoc n l p) )

( assoc_eq_1 : !! {n,r : id}{A : Type}{a : A}{l : hal}
  {p : (occ n (cons r A a l))}
  (?? (EQ n r)) ->
  (EQ (assoc n (cons r A a l) p) a) )

( assoc_eq_2 : !! {n,r : id}{A : Type}{a : A}{l : hal}
  {p : (occ n (cons r A a l))}
  (?? (Not (EQ n r))) -> (?? (occ n l)) ->
  (EQ (assoc n (cons r A a l) p) (assoc n l ?)) )

```

Given this specification we have the following executions:

```

( l = (cons a bool true (cons b nat 3 (cons c bool false nil))) )

( Verify (occ b l) )
verification succeeded

```

```
( Red (tassoc b 1 ?) )
nat

( Red (assoc b 1 ?) )
( suc ( suc ( suc 0 ) ) )
```

Typical Ambiguity

We have already encountered existential metavariables, introduced and prefixed by `?`, that the OCC type checker tries to solve as soon and as far as possible. The remaining constraints, if any, are posted back as goals to the user. As we shall see later, the user can interact with the system to solve metavariables that cannot be solved automatically. Another kind of metavariables, that we call universal metavariables, introduced and prefixed by `!`, are never automatically solved by the type checker, and can therefore be used to defer certain decisions, such as the choice of universes in the following example.

```
( id = [X : !][x : X] x )
( id' = [X : !][x : X] x )

( res = (id ? id') : ?)
```

These definitions give rise to the following context

```
{ id = [ X : !2291 ][ x : X ] x }
{ id' = [ X : !2302 ][ x : X ] x }
{ res = ( id ( { X : !2302 } { x : X } X ) id' )
      : { X : !2302 } { x : X } X }
```

in which we can evaluate terms, e.g.,

```
( Red res )
( [ X : !2302 ] [ x : X ] x )
```

Up to this point OCC has accumulated a number of constraints on metavariables under which this context and the evaluation are well-typed. Usually, constraints arising from typical ambiguity are kept and accumulated as goals. Eventually, however, they should be completely solved by a choice of universes such as

```
(!2302 := Type2)
(!2291 := ?)
```

yielding a context

```
{ id  = [ X : Type3 ][ x : X ] x }
{ id' = [ X : Type2 ][ x : X ] x }
{ res = ( id ( { X : Type2 }{ x : X } X ) id' )
      : { X : Type2 }{ x : X } X }
```

This last step can be automated and hence avoided by a check of satisfiability for all the accumulated constraints (similar to [HP91, CAB⁺86, Hue87]), but such a check, which would be specific for this particular application of universal metavariables, has not been implemented in the present version of the OCC prototype.

Typical ambiguity, i.e. leaving open the choice of the concrete universes, should not be confused with *universe polymorphism*, which allows the implicit instantiation of a single definition for different universe levels, and would allow us to use a single definition of the identity function above.²² We consider universe polymorphism an important feature that should be added to OCC in the future. However, in contrast to its view as a metalevel concept, like e.g. in [HP91], we believe that it should have a semantic basis inside the type theory and it should not be confined to definitions, because OCC favors equational specifications which are more powerful than definitional extensions. Indeed, several difficulties with the notion of universe polymorphism have motivated an extension of ECC [Cou02], which explicitly supports universe expressions (denoting the level of predicative universes) inside the theory. A related but more radical possibility is an extension of OCC with *universe types*, which contain (only) universes as elements and hence provide a new abstraction for collections of universes.²³ In such a setting universe polymorphism would emerge a special case of implicit/explicit polymorphism inside the type theory.

8.2.3 Executable Behavioral Specifications

It has been widely argued that reasoning about structure is usually not the appropriate level of abstraction to deal with today's complex software systems which are often designed as a collection of communicating objects. An early categorical approach to object behavior [Rei95] proposes the use of final coalgebras as a model for abstract objects as opposed to initial algebras, which are used to model

²²In this context it might be interesting to point out that universe polymorphism is supported by LEGO and COQ, although not in a fully satisfactory way as observed in [Pol01, Cou02], because, for instance, the identity function cannot be applied to itself.

²³It should, however, be noted that such a generalization is logically not unproblematic in connection with impredicative universes as indicated by the similar idea of supertypes in [Coq86].

objects with a visible structure (see also the tutorial [JR97] for a good introduction to the subject). In a similar spirit, the program of hidden algebra initiated by an earlier version of [GM00] advocates the idea of behavioral specifications as an extension of algebraic specifications by so-called hidden sorts. As demonstrated by many examples, e.g. in [JR97], both algebraic and coalgebraic methods are attractive for mathematical modeling and reasoning. Typically, both methods are used in an intertwined way, i.e., behavioral specifications involve ordinary specifications and vice versa.

The following small example is a behavioral specification of flags. It uses the previously introduced type `bool`. The example is taken from [GM00] which designates `flag` as hidden. In contrast to our previous examples, we do not use an initial semantics for `flag`. Instead, the intended semantics of `flag` is the final one, i.e. the model which identifies all objects of type `flag` that are not behaviorally distinguishable (see Section 8.2.7 for the formal notion of final algebra). In behavioral specifications we can distinguish two kinds of functions: *observers*, i.e. functions from a hidden type to a visible type, and *modifiers*, i.e. functions from a hidden type (and possibly additional arguments) to itself. In the example, `flag` is intended to be a hidden type, `up?` is the only observer, and `down`, `up`, and `rev` are the modifiers.

```
( flag : Type )

( up? : flag -> bool )

( down : flag -> flag )
( up   : flag -> flag )
( rev  : flag -> flag )

( up1_as : !! {f : flag} (EQ (up? (up f)) true) )
( up2_as : !! {f : flag} (EQ (up? (down f)) false) )

( rev_as : !! {f : flag} (EQ (up? (rev f)) (not (up? f))) )

( Red (up? (rev (rev (rev (down f)))))) )
true
```

We can see from this example that, thanks to the computational equations, it is possible to compute with abstract objects *without* referring to their structure, which is completely hidden at this level.

Parameterized Behavioral Specifications

A slightly more involved example of a behavioral specification in OCC is the following specification of infinite streams over an arbitrary parameter type. We have a single observer `head` and a single modifier `tail`. For each choice of the parameter the intended semantics is the final one, and more generally the cofree semantics (see Section 8.2.7 for details).

```
( stream : Type -> Type )

( head : {T | Type} (stream T) -> T )

( tail : {T | Type} (stream T) -> (stream T) )
```

We furthermore specify equationally two functions `odd` and `even` on streams, which generate the substreams of odd- and even-indexed elements, respectively.

```
( odd : {T | Type} (stream T) -> (stream T) )

( odd_eq_1 : !! {T : Type}{s : (stream T)}
  (EQ (head (odd s)) (head s)) )

( odd_eq_2 : !! {T : Type}{s : (stream T)}
  (EQ (tail (odd s)) (odd (tail (tail s)))) )

( even : {T | Type} (stream T) -> (stream T) )

( even_eq : !! {T : Type}{s : (stream T)}
  (EQ (even s) (odd (tail s))) )
```

We also introduce a function `merge` which merges two streams and a function `ith` which extracts an element at a given index from a stream.

```
( merge : {T | Type} (stream T) -> (stream T) -> (stream T) )

( merge_eq_1 : !! {T : Type}{s1,s2 : (stream T)}
  (EQ (head (merge s1 s2)) (head s1)) )

( merge_eq_2 : !! {T : Type}{s1,s2 : (stream T)}
  (EQ (tail (merge s1 s2)) (merge s2 (tail s1))) )

( ith : {T | Type} (stream T) -> nat -> T )
```

```
( ith_eq_1 : !! {T : Type}{s : (stream T)}
  (EQ (ith s 0) (head s)) )

( ith_eq_2 : !! {T : Type}{s : (stream T)}{i : nat}
  (EQ (ith s (suc i)) (ith (tail s) i)) )
```

All these examples show how the same equational specification style that we used for the definition of functions on inductive datatypes can be used to state concise and readable specifications in the coinductive case. To demonstrate, furthermore, the computation with stream objects, we introduce the stream of all natural numbers, which again can be equationally specified.

```
( nats : nat -> (stream nat) )

( nats_eq_1 : !! {n : nat}
  (EQ (head (nats n)) n) )

( nats_eq_2 : !! {n : nat}
  (EQ (tail (nats n)) (nats (suc n))) )

( Red (ith (merge (odd (nats 0)) (even (nats 0))) 0) 0 )
0
( Red (ith (merge (odd (nats 0)) (even (nats 0))) 1) 1 )
( suc 0 )
( Red (ith (merge (odd (nats 0)) (even (nats 0))) 2) 2 )
( suc ( suc 0 ) )
```

This example shows another benefit of behavioral specifications, namely that we can represent and compute with potentially infinite objects. This is only possible, because we abstract from their internal structure, and interact with such objects via a finitary interface.

In summary, we can say that behavioral specifications provide us with a *means* to deal with potentially infinite objects, but, on the other hand, one should not consider this as the main *purpose* of behavioral specifications, which is to introduce a level that is more abstract than the essentially syntactic level of standard algebraic data types with their initial or free semantics.

8.2.4 Representation of Object Languages

Higher-Order Abstract Syntax

Higher-order abstract syntax [PE88] has been proposed as a systematic approach to represent the abstract syntax of languages with binding constructs in logical frameworks such as LF [HHP87]. The main idea is to represent the binding constructs of the object language using binding constructs of the metalanguage, i.e. to aim at a shallow embedding concerning this issue. In the case of type theory such binding constructs are represented using λ -abstractions. In this case, a major convenience of higher-order abstract syntax is that α -equality is inherited from the metalanguage and substitution corresponds to β -reduction. In the following we use terms of the untyped λ -calculus with a single constant as an example. Using the original approach [PE88] a type of terms would be specified as follows:

```
( term : Type )

( const : term )
( app : term -> term -> term )
( abs : (term -> term) -> term )
```

For instance, the untyped λ -terms $\lambda x.x$ and $\lambda x.\lambda y.x$ would be represented as `(abs ([x : term] x))` and `(abs ([x : term] (abs ([y : term] x))))`, respectively, in the context of this specification.

It is important to emphasize that we do not impose any constraints at this point. In particular we cannot consider this specification as an inductive definition, because of the negative occurrence of `term` in the argument type of `abs`. A variant of higher-order abstract syntax in which `abs` has a type `(var -> term) -> term` for a type of names `var` has been proposed in [DH94, DFH95] (see also [GM96b] for a similar approach) to address this problem, but this approach seems to be less natural, since substitution does not correspond to β -reduction, a feature which makes the original approach to higher-order abstract syntax computationally very attractive.

Another well-known problem with the specification above is that the assumption of a standard elimination principle would allow us to construct elements in `term` that do not represent terms in the λ -calculus, and the same problem occurs with the induction principle if we assume the axiom of unique choice [Hof99]. We feel that this is a sign that the function space `term -> term` is too rich to provide a faithful representation of λ -abstraction bodies.

In view of these difficulties it is not immediately clear how the idea of higher-order abstract syntax could be fruitfully used under a classical set-theoretic semantics

in a computationally useful way. Obviously the full set-theoretic function space ($\mathbf{term} \rightarrow \mathbf{term}$), which is used as a domain of \mathbf{abs} , contains functions that do not represent the body of a lambda abstraction. An even more serious problem with the above specification appears if we need β -equality for the represented λ -terms. The addition of β -equality as an axiom would require \mathbf{abs} to be injective, which is clearly impossible under a classical set-theoretic semantics.

If we wish to maintain the computational benefits of higher-order abstract syntax, a possible solution is to refine the above specification and to be precise about what elements of \mathbf{term} and $\mathbf{term} \rightarrow \mathbf{term}$ are suitable representatives of terms and abstraction bodies. To capture this information we specify two predicates $\mathbf{term?}$ and $\mathbf{body?}$ as follows. The main idea behind the last axiom below is that a function in $\mathbf{term} \rightarrow \mathbf{term}$ represents a body if it is uniform in the sense that it respects each conceivable (partial) congruence \mathbf{eq} .

```
( term? : term -> Prop )
( body? : (term -> term) -> Prop )

( term?_ax_1 : ?? (term? const) )

( term?_ax_2 : ?? {M,N : term}
  (term? M) -> (term? N) -> (term? (app M N)) )

( term?_ax_3 : ?? {B : term -> term}
  (body? B) -> (term? (abs B)) )

( body?_ax : ?? {B : term -> term}
  ({X,Y : term}{eq : term -> term -> Prop}
  (?? (eq const const)) ->
  (?? {M,N,M',N' : term}
    (eq M M') -> (eq N N') ->
    (eq (app M N) (app M' N')))) ->
  (?? {B,B' : term -> term}
    ({X,Y : term}(eq X Y) -> (eq (B X) (B' Y))) ->
    (eq (abs B) (abs B')))) ->
  (?? (eq X Y)) -> (eq (B X) (B Y))) ->
  (body? B) )

( Verify (term? (abs ([x : term] (abs ([y : term] x)))))) )
verification succeeded
```

Notice that the condition ($\{X,Y : \mathbf{term}\}\{\mathbf{eq} : \mathbf{term} \rightarrow \mathbf{term} \rightarrow \mathbf{Prop}\} \dots$) of the operational assertion $\mathbf{body?_ax}$ has a universal quantifier which operationally amounts to the generation of fresh variables X,Y , and \mathbf{eq} , for which the

condition ... is verified. Unfortunately, the current OCC prototype does not implement operational axioms with universally quantified conditions, so that this example cannot be actually executed as specified.

Initial models of this specification can be constructed, e.g. using an approach similar to [GM96b] based on de Bruijn indices or using a model based on a CINNI representation with α -equality (see Chapter 5 and the subsequent section).

To accommodate for the equational nature of the λ -calculus, the specification can be extended by computational equations for β - and η -conversion as below.

```
( beta : !! {B : (term -> term)}{M : term}
  (body? B) -> (term? M) ->
  (EQ (app (abs B) M) (B M)) )

( Red (app (abs ([x : term] x)) (abs ([x : term] x))) )
( abs ( [ x : term ] x ) )

( eta : ?? {B : (term -> term)}{F : term}
  (body? B) -> (term? F) ->
  ({P : term} (EQ (B P) (app F P))) -> (EQ (abs B) F) )

( Verify (EQ (abs ([y : term] (app const y))) const) )
verification succeeded
```

Alternatively, we could specify η -conversion using a computational equation rather than using assertional equality:

```
( eta : !! {B : (term -> term)}{F : term}
  (body? B) -> (term? F) ->
  ({P : term} (EQ (B P) (app F P))) -> (EQ (abs B) F) )

( Red (abs ([y : term] (app const y))) )
const
```

Notice that the conclusion $(EQ (abs B) F)$ of `eta` has a variable `F` on the right which does not appear on the left hand side of the conclusion. Operationally, `F` has to be determined by the condition $(\{P : term\} (EQ (B P) (app F P)))$ via pattern matching, i.e. by reducing $(B P)$ and matching against the pattern $(app F P)$. However, like universally quantified conditions, such matching conditions are not supported in the current OCC prototype.

It is remarkable that in all these cases the use of conditional equations allows us to avoid λ -abstractions or, more generally, variable binding constructs in patterns, which was the problem originally addressed by combinatory reduction systems

[Klo80] and higher-order rewrite systems [Nip91]. The patterns that we used above on the left hand side of computational equations and for assertional equations are obviously not first-order, but constitute a simple form of higher-order patterns without variable binding constructs. In fact, patterns involving variable binding constructs are nearly useless in OCC, since structural equality does not subsume α -equality. It seems that our rather modest approach to higher-order rewriting, which clearly has a first-order flavor, constitutes an interesting alternative to the approaches mentioned above, although the exact relationship still remains to be investigated.

First-Order Representations via CINNI

The equational flexibility of OCC opens a first-order alternative to the use of higher-order abstract syntax, namely the use of a more concrete representation based on explicit substitutions, e.g. in form of the CINNI calculus introduced in Chapter 5. As an example we give an OCC specification of CINNI_λ , i.e. CINNI instantiated for the untyped λ -calculus.

```
( trm : Type )

( var : id -> nat -> trm )
( app : trm -> trm -> trm )
( lam : id -> trm -> trm )

( Subst : Type )

( subst : id -> trm -> Subst )
( shift : id -> Subst )
( lift : id -> Subst -> Subst )

( apply : Subst -> trm -> trm )
```

Recall that CINNI_λ has three groups of syntax-independent equations, namely those for the explicit substitution operators `subst`, `shift`, and `lift`, and in addition the group of syntax-specific equations:

```
( FVar : !! {X : id}{M : trm}
  (EQ (apply (subst X M) (var X 0)) M) )
( RVarEq : !! {X : id}{M : trm}{m : nat}
  (EQ (apply (subst X M) (var X (suc m))) (var X m)) )
```

```

( RVarNeq : !! {X,Y : id}{M : trm}{n : nat} (Not (EQ X Y)) ->
  (EQ (apply (subst X M) (var Y n)) (var Y n)) )
( VarShiftEq : !! {X : id}{m : nat}
  (EQ (apply (shift X) (var X m)) (var X (suc m))) )
( VarShiftNEq : !! {X,Y : id}{M : trm}{n : nat} (Not (EQ X Y)) ->
  (EQ (apply (shift X) (var Y n)) (var Y (suc n))) )

( FVarLift : !! {S : Subst}{X : id}
  (EQ (apply (lift X S) (var X 0))
    (var X 0)) )
( RVarLiftEq : !! {S : Subst}{X : id}{m : nat}
  (EQ (apply (lift X S) (var X (suc m)))
    (apply (shift X) (apply S (var X m)))) )
( RVarLiftNEq : !! {S : Subst}{X,Y : id}{m : nat}(Not (EQ X Y)) ->
  (EQ (apply (lift X S) (var Y m))
    (apply (shift X) (apply S (var Y m)))) )

( App : !! {S : Subst}{M,N : trm}
  (EQ (apply S (app M N)) (app (apply S M) (apply S N))) )
( Lam : !! {S : Subst}{X : id}{M : trm}
  (EQ (apply S (lam X M)) (lam X (apply (lift X S) M))) )

```

To this specification we could add computational β - and η -equations as follows. Again, note that universally quantified conditions, as used in `eta` below, are not supported in the current OCC prototype.

```

( beta : !! {M,N : trm}{X : id}
  (EQ (app (lam X M) N) (apply (subst X N) M)) )

( Red (app (lam a (var a 0)) (lam a (var a 0))) )
( lam a ( var a 0 ) )

( eta : !! {N,M : term}{X : id}
  ({P : term} (EQ (apply (subst X P) M) (app N P))) ->
  (EQ (lam X M) N) )

( Red (lam a (app b (var a 0))) )
b

```

The advantage of using a first-order representation rather than the original approach to higher-order abstract syntax [HHP87, PE88] is that it is independent of any assumptions about the semantics of function types, i.e., it is compatible with constructive as well as with classical models. Differently from the higher-order

abstract syntax approach and the variation presented in the previous section, we have access to variable names, since this information is explicitly represented in CINNI, provided that we do not add an equation for α -equality, which is another possibility (cf. Chapter 5).

Although we have chosen the λ -calculus as an object language for our examples, recall that the CINNI approach is of similar generality as the higher-abstract syntax approach. For instance, the representation of other programming languages, type theories and logics such as the ζ -calculus or π -calculus can be done as explained in Chapter 5, and pure type systems (and corresponding higher-order logics) can be represented as discussed in Chapter 6. The relevant capability of OCC to represent rewriting specifications will be demonstrated in Section 8.2.8.

Beyond the representation of object languages, OCC offers a higher-order logic framework for formal proofs about represented programs/systems, so that OCC can play the same role as CIC in our representation of UNITY (cf. Chapter 4 and Section 8.3.5).

8.2.5 Elimination and Coelimination Principles

Elimination principles are a general scheme to construct functions from inductive types to other types and in case of types without equations they can be generated from the inductive definition, i.e. from the inductive type and its constructors, in a systematic way, e.g. using the approaches of LEGO or COQ described in [CPM90] and [Luo94], respectively. OCC does not have built-in inductive types with fixed elimination principles. Instead, in OCC such principles are user-definable and hence do not have to obey fixed schemes. This is made possible by the openness of the computational system, a feature that we cannot find in type theories such as UTT²⁴ or CIC.

We have seen that in OCC functions on datatypes can be defined equationally in a quite natural way. This style of function definition has the advantage of extensibility, i.e. extending the data type with new elements just requires the addition of equations for these. For instance, the following definition of a polymorphic function `if` does not constrain the datatype `bool` to two elements, although it implies that `true` and `false` can be distinguished, since we can use `if` to perform a case analysis.

²⁴It is interesting that the LEGO implementation supports the ad hoc introduction of (unconditional) computation rules by the user (at the top level), but these rules are unconditional and not expressed in the logic of the type theory as it is the case in OCC. Although these rules are intended to specify the computational behavior of elimination operators (that are automatically generated by inductive definitions), the value of this flexibility in the hands of the user can be observed in [Pol01], where LEGO is used as a programming language and such rules are directly used to define functions via pattern matching.

```
( if : {T | Type} bool -> T -> T -> T )
( if_eq_1 : !! {T : Type}{x,y : T} (EQ (if true x y) x) )
( if_eq_2 : !! {T : Type}{x,y : T} (EQ (if false x y) y) )
```

The equational specification style of OCC also has the advantage that definitions of functions are easy to understand and can often be done by pattern matching modulo equational theories. On the the hand, the flexibility of equational definitions has also certain drawbacks, such as the possibility of ill-formed definitions, which could be either inconsistent or incomplete w.r.t. the given constructors or operationally ill-behaved, e.g. due to nonconfluence or nontermination.

These drawbacks can be avoided by giving up the flexibility of equational definitions and instead defining functions according to schemes that are known to be consistent, complete and operationally well-behaved. A particular common form of such a scheme is (*higher-order*) *primitive recursion*, which can be formulated in OCC as shown by the following example of a polymorphic primitive recursion operator for `nat`.

```
( nat_rec : {T | Type}
  T ->
  ( nat -> T -> T ) ->
  nat -> T )

( nat_rec_eq_1 : !! {T : Type}{z : T}{f : nat -> T -> T}
  (EQ (nat_rec z f 0) z) )

( nat_rec_eq_2 : !! {T : Type}{z : T}{f : nat -> T -> T}{n : nat}
  (EQ (nat_rec z f (suc n)) (f n (nat_rec z f n))) )

( nat_iter = [T | Type][z : T][f : T -> T]
  (nat_rec z ([_ : nat] f)) )

( nat_plus = [m,n : nat]
  (nat_iter n suc m) )

( nat_times = [m,n : nat]
  (nat_iter 0 (nat_plus n) m) : nat -> nat -> nat )

( nat_fact = [m : nat]
  ( nat_rec 1 ( [ n : nat ] [ nat_factn : nat ]
    ( nat_times (suc n) nat_factn ) ) m ) )

( Red (nat_fact 3) )
( suc ( suc ( suc ( suc ( suc 0 ) ) ) ) ) )
```

The simple primitive recursion principle `nat_rec`, which subsumes the iteration principle `nat_iter` as a special case, is sufficient to define (higher-order) primitive recursive functions from `nat` into a uniform domain `T`, but in a theory with dependent types we wish to be able to define dependent functions, where the domain depends on the actual argument, which is an element of `nat` in this case. Hence, there is a more general dependent recursion principle `nat_elim`, also called *elimination principle*, from which `nat_rec`, that we also call *simple elimination principle*, is obtained as a special case:

```
( nat_elim : {T | nat -> Type}
  (T 0) ->
  ({n : nat}(T n) -> (T (suc n))) ->
  {z : nat} (T z) )

( nat_elim_eq_1 : !! {T : nat -> Type}
  {z : (T 0)}{f : {n : nat}(T n) -> (T (suc n))}
  (EQ (nat_elim z f 0) z) )

( nat_elim_eq_2 : !! {T : nat -> Type}
  {z : (T 0)}{f : {n : nat}(T n) -> (T (suc n))}{n : nat}
  (EQ (nat_elim z f (suc n)) (f n (nat_elim z f n))) )

( nat_rec = [T | Type] (nat_elim | ([_ : nat] T))
  : {T | Type} T -> (nat -> T -> T) -> nat -> T )
```

We will show in Section 8.2.7 that the elimination principle is a consequence of our initiality constraint for the specification of `nat`. It expresses that `nat` is generated by its constructors and that, due to the lack of equations, no identifications are imposed between elements of `nat`. The first condition is reflected in the fact that `nat_elim` has one argument for each constructor, and the second condition is reflected in the specified equations for `nat_elim`, which allows us to distinguish occurrences of these two constructors. In other words, adding the elimination principle closes up the corresponding data type and its associated equality (which is the trivial syntactic equality in the case of `nat`).

In the case of `bool`, the elimination principle, that we also call *enumeration principle* in the case of enumeration types like this, is the one given below, and *the simple enumeration principle* is given by the function `if` which now arises as a special case.

```

( bool_elim : {T | bool -> Type}
  (T true) ->
  (T false) ->
  {b : bool} (T b) )

( bool_elim_ax_true : !! {T : bool -> Type}
  {t : (T true)}{f : (T false)}
  (EQ (bool_elim t f true) t) )

( bool_elim_ax_false : !! {T : bool -> Type}
  {t : (T true)}{f : (T false)}
  (EQ (bool_elim t f false) f) )

( if = [T | Type][x,y : T] (bool_elim | ([_ : bool] T) x y) )

```

We conclude this section with the standard elimination principle for the parameterized disjoint union type `sum`, i.e. dependent case analysis, which is used to define `decide` for simple case analysis, a function that will be used later.

```

( sum_elim : {U,V | Type}{T | (sum U V) -> Type}
  ({u : U} (T (left u))) ->
  ({v : V} (T (right v))) ->
  ({s : (sum U V)} (T s)) )

( sum_left : !! {U,V : Type}{T : (sum U V) -> Type}
  {f : {u : U} (T (left u))}{g : {v : V} (T (right v))}{u : U}
  (EQ (sum_elim f g (left u)) (f u)) )

( sum_right : !! {U,V : Type}{T : (sum U V) -> Type}
  {f : {u : U} (T (left u))}{g : {v : V} (T (right v))}{v : V}
  (EQ (sum_elim f g (right v)) (g v)) )

( decide = [U,V | Type][T | Type]
  [f : U -> T][g : V -> T][s : (sum U V)]
  (sum_elim | U | V | ([_ : (sum U V)] T) f g s) )

```

The cartesian product type `prod` specified earlier has the remarkable property that it is completely characterized by its projections `fst` and `snd`, which act as elimination operators. Hence, although an elimination principle according to the general scheme could be given, it is not needed in this case.

Another data type that we have defined before is the type of finite multisets over natural numbers with constructors `fms_empty`, `fms_single` and `fms_union`.

Differently from the previous examples, we are concerned with an equationally defined data type. This is reflected in the elimination principle given below by the additional assumption that the function to be used for elimination preserves the equational properties.

```
( fms_nat_elim : {T | fms_nat -> Type}
  { empty :
    (T fms_empty) }
  { single :
    ({x : nat} (T (fms_single x))) }
  { union : ({m,m' | fms_nat}
    (T m) -> (T m') -> (T (fms_union m m')) ) }
  { comm : {m,m' : fms_nat}{t : (T m)}{t' : (T m')}}
    (EQ (union t t') (union t' t)) }
  { assoc : {m,m',m'' : fms_nat}
    {t : (T m)}{t' : (T m')}{t'' : (T m'')}}
    (EQ (union t (union t' t'')) (union (union t t') t'')) }
  { id : {m : fms_nat}{t : (T m)}
    (EQ (union t empty) t) }
  { m : fms_nat } (T m) )
```

In addition we have the obvious computational equation for each constructor. Here we only give the equation for the recursive case, which makes again use of a nonemptiness predicate:

```
( fms_is_empty : fms_nat -> Prop )

( fms_not_is_empty_as : ?? {m : fms_nat}{n : nat}
  (Not (fms_is_empty (fms_union m (fms_single n)))) )

( fms_nat_elim_eq_3 : !! {T | fms_nat -> Type}
  { empty :
    (T fms_empty) }
  { single :
    ({x : nat} (T (fms_single x))) }
  { union : ({m,m' | fms_nat}
    (T m) -> (T m') -> (T (fms_union m m')) ) }
  { comm : {m,m' : fms_nat}{t : (T m)}{t' : (T m')}}
    (EQ (union t t') (union t' t)) }
  { assoc : {m,m',m'' : fms_nat}
    {t : (T m)}{t' : (T m')}{t'' : (T m'')}}
    (EQ (union t (union t' t'')) (union (union t t') t'')) }
  { m : fms_nat } (T m) )
```

```

{ id : {m : fms_nat}{t : (T m)}
  (EQ (union t empty) t) }
{ m,m' : fms_nat }
(Not (fms_is_empty m)) -> (Not (fms_is_empty m')) ->
(EQ (fms_nat_elim empty single union comm assoc id
     (fms_union m m'))
  (union
    (fms_nat_elim empty single union comm assoc id m)
    (fms_nat_elim empty single union comm assoc id m')))) )

```

It is straightforward to give corresponding parameterized elimination principles for a parameterized datatype of finite multisets by replacing `nat` by an arbitrary type parameter. Also, corresponding elimination principles for lists and finite sets can be given along the same lines.

As pointed out before we do not require elimination principles to follow fixed schemes. Instead they may be justified externally, e.g. be imported from other theorem provers or simply be justified by conventional mathematical means. A good example is the following elimination principle for our representation of the syntax of untyped λ -calculus (i.e. omitting β - and η -equality) based on CINNI given in Section 8.2.4. It can be justified by the metatheory of CINNI developed in Chapter 5, namely the strong normalization result, which ensures that substitutions can always be eliminated. For the sake of brevity we have omitted the obvious computational equations here.

```

( trm_elim : {T : trm -> Prop}
  ({X : id}{n : nat} (T (var X n))) ->
  ({M,N : trm} (T M) -> (T N) -> (T (app M N))) ->
  ({X : id}{M : trm}(T M) -> (T (lam X M))) ->
  {M : trm} (T M) )

```

In general, elimination principles are used to define functions from an inductive type into some other domain. Dually, coelimination principles should allow us to define functions from some domain into a coinductive type. A simple example of a coelimination principle is the following coiteration principle for our behavioral specification of streams.

```

( stream_coiter : {X | Type}{T | Type}
  {f : T -> X}{g : T -> T}
  T -> (stream X) )

```

```

( stream_coiter_eq_1 : !! {X : Type}{T : Type}
  {f : T -> X}{g : T -> T}{t : T}
  (EQ (head (stream_coiter | X | T f g t)) (f t)) )

( stream_coiter_eq_2 : !! {X : Type}{T : Type}
  {f : T -> X}{g : T -> T}{t : T}
  (EQ (tail (stream_coiter | X | T f g t))
      (stream_coiter | X | T f g (g t))) )

```

As shown below, it can be used to *define* the functions `odd`, `even` and `merge` that we *specified* in Section 8.2.3.

```

( odd = [T | Type]
  (stream_coiter | T | (stream T)
    ([s : (stream T)](head s))
    ([s : (stream T)](tail (tail s)))) )

( even = [T : Type][s : (stream T)] (odd (tail s)) )

( merge = [T : Type]
  (stream_coiter | T | (prod (stream T) (stream T))
    ([s1s2 : (prod (stream T) (stream T))]
      (head (fst s1s2)))
    ([s1s2 : (prod (stream T) (stream T))]
      (pair (snd s1s2) (tail (fst s1s2))))) )

```

The style of specifying coinductive datatypes that we employ here (and in Section 8.2.3) is inspired by the behavioral (i.e. hidden algebra) specifications of [GM00] and by the categorical approach to coalgebra and coinduction [JR97]. Both approaches use observer functions (the dual of constructor functions) to define the equality associated with a coinductive type. The approach adopted in COQ, which follows [Gim94] (continuing the approach of [Coq93]) has a notion of coinductive types without imposed equations, similar to the notions of inductive types [Luo94, PM93] without equations that is used in type theories such as UTT and CIC. Although, in such type theories suitable equivalence relations could replace a coarser equality, we feel that the OCC approach to directly support equationally defined datatypes is more natural and leads to better computational properties, and hence to a higher degree of automation in theorem proving.

An approach that is similar to inductive datatypes with equations are quotients of inductive datatypes as they can be constructed in Nuprl [CAB⁺86]. Quotient types would require us to define inductive types in three stages. First we

would define an inductive type without equations, then we would define a suitable equivalence, and finally we would take the quotient. OCC follows the membership equational logic approach, where constructors, equality and other predicates and functions can be specified simultaneously with possible interdependencies. Although OCC does not have built-in quotient types, we think that quotient types are very useful to support conservative constructions and to relate different levels of abstractions. A possible approach to an internal notion of quotient types for OCC could be based on setoids or better on categories, an issue that we touch upon in the following section.

8.2.6 Higher-Order Logic and Equality

In the following we use the universal-implicative fragment of higher-order logic, which is naturally provided by the propositions-as-types interpretation of the impredicative universe `Prop`, to introduce further logical constants and operators. For each logical operator we have the usual introduction and elimination operators that are known from natural deduction presentations. In fact, the operators can be seen as constructors of natural deduction proofs. Our axiomatization provides the same introduction and elimination principles as LEGO and COQ, but in contrast to these systems we do not impose any equational properties on proofs at this stage, i.e. we keep proofs as informative as possible.

```
( True : Prop )
( True_intro : ?? True )

( False : Prop )
( False_elim : False -> ( {A : Prop} A ) )

( Implies : Prop -> Prop -> Prop )
( Implies_intro : {A,B : Prop} (A -> B) -> (Implies A B) )
( Implies_elim : {A,B : Prop} (Implies A B) -> A -> B )

( And : Prop -> Prop -> Prop )
( And_intro : ?? {A,B : Prop} A -> B -> (And A B) )
( And_elim_l : {A,B : Prop} (And A B) -> A )
( And_elim_r : {A,B : Prop} (And A B) -> B )

( Or : Prop -> Prop -> Prop )
( Or_intro_l : ?? {A,B : Prop} A -> (Or A B) )
( Or_intro_r : ?? {A,B : Prop} B -> (Or A B) )
( Or_elim : {A,B,C : Prop} (Or A B) -> (A -> C) -> (B -> C) -> C )
```

```

( Iff : Prop -> Prop -> Prop )
( Iff_intro : {A,B : Prop} (A -> B) -> (B -> A) -> (Iff A B) )
( Iff_elim_lr : {A,B : Prop} (Iff A B) -> (A -> B) )
( Iff_elim_rl : {A,B : Prop} (Iff A B) -> (B -> A) )

( Not : Prop -> Prop )
( Not_intro : {A : Prop}(A -> False) -> (Not A) )
( Not_elim : {A : Prop}(Not A) -> A -> False )

( All : {T : Type} (T -> Prop) -> Prop )
( All_intro : {T : Type}{P : (T -> Prop)}
  ({x : T} (P x)) -> (All T P) )
( All_elim : {T : Type}{P : (T -> Prop)}
  (All ? P) -> {x : T} (P x) )

( Ex : {T : Type} (T -> Prop) -> Prop )
( Ex_intro : {T : Type}{x : T}{P : (T -> Prop)}
  (P x) -> (Ex T P) )
( Ex_elim : {T : Type}{P : (T -> Prop)}
  (Ex ? P) -> {A : Prop} ( { x : T } (P x) -> A ) -> A )

```

Axioms which have a natural computational interpretation, namely `True_intro`, `And_intro`, `Or_intro_l`, and `Or_intro_r`, are designated as operational by means of `??`.

Leibnitz Equality and Functional Extensionality

A general way to introduce equality in an impredicative higher-order logic is to define it as Leibnitz equality. This is for instance the approach adopted in LEGO and COQ. In OCC, equality is a built-in notion that can be enforced to be equivalent to Leibnitz equality by adding the axioms below. It is easy to see that these axioms are justified by the OCC semantics given in Section 8.1.2.

```

( eq_imp_leq : {U : Type}{x,y : U}
  (EQ x y) -> {P : U -> Prop}(P x) -> (P y) )

( leq_imp_eq : {U : Type}{x,y : U}
  ({P : U -> Prop}(P x) -> (P y)) -> (EQ x y) )

```

OCC does not have built-in η -rules or more general extensionality rules for equality of functions. Similar to [Luo94] we adopt the view that extensionality in the form of η -reduction rules should not be regarded as a computational equality.

Instead, we propose to add extensionality as assertional equality, which does not favor any direction of computation. This addition can be done selectively, i.e. only for functions between certain types. For instance, extensionality can be specified with operational assertions such as the following.

```
( extensionality : ?? {S : Type}{T : S -> Type}{f,g : {x : S} (T x)}
  ({x : S} (EQ (f x) (g x))) -> (EQ f g) )

( Verify (EQ suc ([m : nat] (suc m))) )
verification succeeded
```

This example is, however, not executable in the current OCC prototype, because it requires operational support for universally quantified conditions and a relaxed variable restriction that allows the instantiation of variables using type inference.

Classical Logic, Logical Extensionality and Proof Irrelevance

The internal intuitionistic higher-order logic of OCC can be tailored to our needs using any combination of the following axioms, the classical law of the excluded middle, logical extensionality, and proof irrelevance. Recall that we have assumed all these axioms in the CIC development of Chapter 4, but by no means we presuppose them for all developments.

```
( classic : {P : Prop}(Or P (Not P)) )

( log_extensionality : {P,Q : Prop}(Iff P Q) -> (EQ P Q) )

( proof_irrelevance : {P : Prop}{p,q : P}(EQ p q) )
```

Weak and Strong Subset Types

In higher-order logics sets are usually represented using characteristic predicates. To be able to talk about arbitrary sets rather than about sets over a fixed type we can represent a set as a type together with its characteristic predicate. This is done in the following specification which introduces the type (`subset T P`) to represent sets with the characteristic predicate `P` over `T` and a constructor `inj` for set elements. From such elements we can extract the actual element using `elem` and using `prf` a witness that the predicate holds for this element.

```
( subset : {T : Type}(T -> Prop) -> Type )

( inj : {T | Type}{P | T -> Prop}{x : T}(P x) -> (subset T P) )
```

```
( elem : {T | Type}{P | T -> Prop}(subset T P) -> T )
( prf  : {T | Type}{P | T -> Prop}{x : (subset T P)} (P (elem x)) )

( elem_eq : !! {T : Type}{P : T -> Prop}{e : T}{p : (P e)}
  (EQ (elem (inj e p)) e) )
```

Although we can extract the raw element used to construct the set element using `elem`, we have not specified that the witness provided by `prf` is related to the witness used for introducing the set. In order to identify set elements with possibly different witnesses, i.e. to abstract from the concrete witness, we impose the equational axiom

```
( abstract_prf : ?? {T : Type}{P : T -> Prop}
  {x,x' : T}{p,p' : (P x)}
  (?? (EQ x x')) -> (EQ (inj x p) (inj x' p')) )
```

Notice how we use assertional equality to make this axiom well-typed. Also, the axiom itself has the operational status of an assertional equality, since there is no reason to equip it with an orientation needed for computational equality.

In this way we have specified a weak form of sets, sometimes also called weak Σ -types, which is similar to the one used in Nuprl [CAB⁺86] and in the extension of Martin-Löf's type theory described in [PSN90].

Of course, instead of adopting the previous axiom we can enforce extractability and hence preservation of the witness using the following equation, which leads to a strong form of sets.

```
( prf_eq : !! {T : Type}{P : T -> Prop}{e : T}{p : (P e)}
  (EQ (prf (inj e p)) p) )
```

It is also instructive to compare subset types (`subset T P`) and existential types (`Ex T P`). The latter do not even allow us to extract the actual element, but according to the elimination principle we can assume the existence of a witness to construct an element of propositional type, i.e. a proof. Since the concrete structure of the witness is hidden, existential types are a common means to formalize the concept of abstract data types [MP85], where the concrete type is hidden but its abstract properties are visible.

In summary, the user-definable computational equations in OCC allow us to specify types with abstract inhabitants, i.e. elements with partially hidden structure. Depending on the strength of the elimination principles various degrees of hiding are possible.

Since we are concerned with a representation of sets in this section, it is worth pointing out that a more general approach is to add an equivalence relation on set elements to the representation. Such structures, sometimes called setoids, can be specified in a similar way. The setoid approach can be further generalized to categories (when the equivalences are replaced by morphisms), which also have a natural representation in OCC. Different from the examples above, the components of such concrete mathematical structures should be fully accessible, and hence are examples of strong Σ -types, which provide a general representation of algebras in type theories. Examples of algebras and categories specified in this style will be given in Section 8.2.7.

An Alternative Approach To Classical Logic

An alternative and quite different approach of embedding classical logic into OCC is to adopt the core idea of the HOL/Nuprl connection, that we reviewed and discussed in Chapter 7, namely to establish a correspondence between propositions and booleans. In fact, different from the pragmatics of CIC (as suggested by the libraries of COQ), Nuprl and OCC (as well as GCC and ECC) use only predicative universes to host data types.

In fact, we can roughly think of the universes \mathbf{Type}_i as a monomorphic, boiled-down version of Martin-Löf's type theory with only universes and dependent types (cf. [PSN90, Luo94]). Similar to the use of Martin-Löf's logical framework [PSN90], we can express all other types of Martin-Löf's monomorphic type theory. In addition, OCC also allows us to specify operational behavior, e.g. computation rules, in a very flexible way, a possibility that is beyond Martin-Löf's logical framework.

Subsequently, we transfer the approach adopted in the HOL/Nuprl connection and described in Chapter 7 from Nuprl to OCC. First we introduce casting functions between propositions and booleans. The functions from `bool` to `Prop` can be given immediately. The opposite direction relies on the strong law of the excluded middle, that we denoted by `inhabited` as in Chapter 7.

```
( bool_to_prop = [b : bool]
  (if b True False) )

( inhabited : {P : Prop} (sum P (Not P)) )

( prop_to_bool = [P : Prop]
  (decide ([x : P] true) ([x : (Not P)] false) (inhabited P)) )

( not = [x : bool] (if x true false) )
( and = [x,y : bool] (if x y false) )
```

```

( or  = [x,y : bool] (if x true y) )
( all = [T | Type][P : T -> bool]
      (prop_to_bool (All ? [x : T] (bool_to_prop (P x)))) )
( ex  = [T | Type][P : T -> bool]
      (prop_to_bool (Ex ? [x : T] (bool_to_prop (P x)))) )

```

Notice that, unlike in Nuprl, we can selectively address propositions, since they reside in the subuniverse `Prop`. In fact, we have restricted `inhabited` to propositions above, but we need the following stronger version for types below, if we want to introduce Hilbert's ϵ -operator as in Chapter 7. In our strongly typed setting of OCC its definition requires a notion of nonempty type, which comes with the axiom `arb` that allows us to pick an unspecified element from such a type, and therefore corresponds to the `arb` operator in Chapter 7.

```

( inhabited : {T : Type} (sum T (T -> False)) )

( non_empty : Type -> Prop )
( arb : {T : Type}{p | (non_empty T)} T )

```

Now the ϵ -operator can be defined in a ways similar to the Nuprl definition from Chapter 7, the only difference being that, since in OCC definitions must be strongly typed, we have to make the nonemptiness of `T` explicit in the definition.

```

( epsilon = [T | Type][p | (?? (non_empty T))][P : T -> bool]
  (decide
    ([x : (subset T ([x : T] (bool_to_prop (P x))))]
      (elem x))
    ([x : (subset T ([x : T] (bool_to_prop (P x)))) -> False]
      (arb T))
    (inhabited (subset T ([x : T] (bool_to_prop (P x)))))) )

```

Again, this is a nice example of how a witness of the nonemptiness assumption is passed to the `arb` function as an implicit argument without interaction by the user. In fact, if we use this approach to embed HOL into OCC (in the spirit of the approach described in Chapter 7), nonemptiness subgoals can be solved automatically thanks to the operational semantics of OCC. Another advantage of this approach to classical logic in OCC is that it does not require impredicativity of `Prop` so that it can be used in purely predicative instances of OCC. For practical purposes, the above construction would be replaced by an abstract axiomatization of the classical logic, e.g. as in [NSM01], because the justification in terms of the law of the excluded middle is irrelevant for its use.

Last but not least, we would like to point out a possibility that is offered by OCC thanks to the admission of universes that are neither predicative nor impredicative, but allow the same classical interpretation as impredicative universes. Assume that we have not used `Prop` but like in Nuprl a standard predicative hierarchy of universes `Typei` to introduce a classical logic exactly as above. Assume furthermore that we replace the type `bool : TYPE_0` by a new bottom universe `Prop : Type0`, a universe that is neither predicative nor impredicative, but has rules $(\text{Prop}, \text{Prop}, \text{Type}_0) \in \mathcal{R}$ and $(\text{Type}_i, \text{Prop}, \text{Type}_i) \in \mathcal{R}$ for all $i \in \mathbb{N}$ and is according to the cumulative hierarchy included in `Type0`. In other words, `Prop` allows the formation of dependent function types, but is not closed under this construction. We would still use casting functions from `Typei` to `Prop` (corresponding to `prop_to_bool` above), but we can omit the casting function from `Prop` to `Type` (corresponding to `bool_to_prop` above), because it just becomes the identity. An additional advantage is that we now have a classical propositional universe `Prop` (instead of just a classical type `bool`), so that OCC can be used almost as if `Prop` was an impredicative universe. This is quite convenient, because this allows us to maintain the identification of propositions with (a subset of) types, which is for instance exploited in the representation of mathematical structures as objects (see next Section) or in the uniform formulation of elimination and induction principles in the style of the ECC and CIC.

8.2.7 Algebras and Categories

Algebras

We have already seen how datatypes have natural equational specifications which can be expressed in OCC. The class of algebras which are monoids, for instance, is given by the following specification under the loose semantics, i.e. without any constraints.

```
( car : Type )
( id : car )
( op : car -> car -> car )
( right_id : {x : car} (EQ (op x id) x) )
( left_id : {x : car} (EQ (op id x) x) )
( assoc : {x,y,z : car} (EQ (op (op x y) z) (op x (op y z))) )
```

Thanks to the availability of universes and dependent types we can go one step further and use strong Σ -types to represent algebras as objects. The importance of strong Σ -types for abstract specifications has been pointed out in [Luo91, Luo94], which is why they have been incorporated into ECC as a major feature. Similar applications of dependent types, namely the representation of modules for programming in the large, have been investigated earlier

[LB88, Mac86], especially in the context of the functional programming languages Pebble and ML. In CIC strong Σ -types are not a primitive notion, but they are subsumed by inductive types,²⁵ and similarly in OCC strong Σ -types can be specified equationally as in the following example, which introduces a type of monoids together with its projections. Using the propositions-as-types interpretation, we can express that a monoid contains not only the standard components, namely the carrier set `car`, the identity `id`, and the monoidal operation `op`, but also corresponding proofs of all monoid axioms.

```
( Monoid : Type )

( monoid :
  {car : Type}
  {id : car}
  {op : car -> car -> car}
  {right_id : {x : car} (EQ (op x id) x)}
  {left_id : {x : car} (EQ (op id x) x)}
  {assoc : {x,y,z : car} (EQ (op (op x y) z) (op x (op y z)))}
  Monoid )
```

In a systematic way we can specify a projection for each component of a monoid. Observe that the type of a projection can depend on projections introduced earlier, so that introducing the computational equation for a projection, requires type checking the application of this projection, which involves computational equations of the projections it depends on.

```
( .car : Monoid -> Type )
( .id : {M : Monoid}
  (.car M) )
( .op : {M : Monoid}
  (.car M) -> (.car M) -> (.car M) )
( .right_id : {M : Monoid}{x : (.car M)}
  (EQ (.op M x (.id M)) x) )
( .left_id : {M : Monoid}{x : (.car M)}
  (EQ (.op M (.id M) x) x) )
( .assoc : {M : Monoid}{x,y,z : (.car M)}
  (EQ (.op M (.op M x y) z) (.op M x (.op M y z))) )

( .car_eq : !! {car : Type}{id : car}{op : car -> car -> car}
```

²⁵A subtle difference is that Σ -types as a primitive concept enjoy universe subtyping in ECC, whereas the (monomorphic) representation inside the type theory as an inductive type does not.

```

{right_id : {x : car} (EQ (op x id) x)}
{left_id  : {x : car} (EQ (op id x) x)}
{assoc   : {x,y,z : car} (EQ (op (op x y) z) (op x (op y z)))}
(EQ (.car (monoid car id op right_id left_id assoc)) car) )

( .id_eq : !! {car : Type}{id : car}{op : car -> car -> car}
  {right_id : {x : car} (EQ (op x id) x)}
  {left_id  : {x : car} (EQ (op id x) x)}
  {assoc   : {x,y,z : car} (EQ (op (op x y) z) (op x (op y z)))}
  (EQ (.id (monoid car id op right_id left_id assoc)) id) )
...

```

A particular example of a monoid is the monoid of natural numbers, a fact that can be verified by type checking only.

```

( nat_monoid = (monoid nat 0 nat_plus
  nat_plus_eq_1 nat_plus_eq_1 nat_plus_assoc) : Monoid )
all goals solved

```

Categories

The logic introduced above is a higher-order logic over the type theory of OCC. Dependent types and universes seem to provide a minimal basis for an adequate formalization of abstract mathematical concepts such as those studied in universal algebra and category theory. The following specification gives a flavor of how categories and important concepts such as initiality and finality, and the more general concepts of freeness and cofreeness, can be specified in the strongly typed language of OCC. The same representation of categories is used in the quite comprehensive formalization of constructive category theory [HS98], which has been carried out in COQ. We especially show in this section how the features of OCC, which go beyond type theories such as CIC and UTT, are essential to formalize the concepts mentioned above, and we also show how these features allow us to specify categories with additional algebraic structure, such as the enriched transition categories that appear in the semantics of rewriting logic.

The following is a specification of arbitrary categories under the loose semantics.

```

( Obj : Type )
( Mor : Obj -> Obj -> Type )

( id : {A | Obj} (Mor A A) )
( comp : {A,B,C | Obj}(Mor B C) -> (Mor A B) -> (Mor A C) )

```

```

( left_id : {A,B : Obj}{f : (Mor A B)}
  (EQ (comp id f) f) )

( right_id : {A,B : Obj}{f : (Mor A B)}
  (EQ (comp f id) f) )

( assoc : {A,B,C,D : Obj}
  {f : (Mor A B)}{g : (Mor B C)}{h : (Mor C D)}
  (EQ (comp h (comp g f) ) (comp (comp h g) f)) )

```

Among many other useful categorical concepts we can define initial and final objects using the embedded logic of OCC:

```

( initial = [I : Obj] {X : Obj}
  (And (Ex ? ([h : (Mor I X)] True))
    ({f,g : (Mor I X)}(EQ f g))) : Obj -> Prop )

( final = [F : Obj] {X : Obj}
  (And (Ex ? ([h : (Mor X F)] True))
    ({f,g : (Mor X F)}(EQ f g))) : Obj -> Prop )

```

Since categories are algebras, we can again use strong Σ -types to specify a type of all (small) categories, which can be further used to specified the (larger) category of all (small) categories, and so on.

A formal notion of categories in OCC can be of interest for various reasons, of which we would like to mention especially the following two:

1. One application is the internalization the fundamental concepts of initiality and finality and, more generally, freeness and cofreeness, which provide us with a uniform notion of inductive and coinductive data types inside the type theory, rather than having these notions as external concepts. In the next subsection we illustrate this possibility by means of data types that have been introduced in a more ad hoc way before.
2. Another quite different application of categories is the use of enriched transition categories as a semantic model for concurrent systems as suggested by the categorical semantics of rewriting logic (cf. Chapter 2). Our treatment of Petri nets in Chapter 3 is a very natural example in this spirit that we will further discuss in a subsequent subsection on enriched categories.

Initial and Final Algebras

In previous examples we have informally used initiality and finality constraints to express that we are not interested in all models satisfying a given OCC specification, but only in a very particular ones, which are unique up to isomorphism. As an expressive higher-order logic, OCC allows us to make explicit such constraints in the logic itself. We will briefly illustrate how this can be done systematically by reconsidering some of our previous specifications. We begin with the specification of the algebra of natural numbers. Since we will be concerned with different algebras, we consider them as first class objects using the approach of Section 8.2.7. This gives rise to the following specification of the type `Nat` of natural number algebras.

```
( Nat : Type )

( MkNat : {car : Type}
          {zero : car}
          {suc : car -> car}
          Nat )

( .car : Nat -> Type )
( .zero : {nat : Nat}(.car nat) )
( .suc : {nat : Nat}(.car nat) -> (.car nat) )

( .car_eq : !! {car : Type}{zero : car}{suc : car -> car}
  (EQ (.car (MkNat car zero suc)) car) )
( .zero_eq : !! {car : Type}{zero : car}{suc : car -> car}
  (EQ (.zero (MkNat car zero suc)) zero) )
( .suc_eq : !! {car : Type}{zero : car}{suc : car -> car}
  (EQ (.suc (MkNat car zero suc)) suc) )
```

To further make explicit the relevant structure of natural number algebras we follow the categorical approach and introduce morphisms which by definition preserve the essential properties. Again, we consider morphisms as first class objects, so that we have types `(NatMor nat1 nat2)` of morphisms from algebra `nat1` to `nat2`.

```
( NatMor : {nat1, nat2 : Nat} Type )
( MkNatMor : {nat1, nat2 : Nat}{h : (.car nat1) -> (.car nat2)}
  {mor_eq_1 :
    (EQ (h (.zero nat1)) (.zero nat2))}
  {mor_eq_2 : {n : (.car nat1)}
    (EQ (h ((.suc nat1) n)) ((.suc nat2) (h n)))}
```

```

(NatMor nat1 nat2) )

( .fun : {nat1, nat2 : Nat}
  (NatMor nat1 nat2) -> (.car nat1) -> (.car nat2) )

( .fun_eq : !! {nat1, nat2 : Nat}{h : (.car nat1) -> (.car nat2)}
  {mor_eq_1 :
    (EQ (h (.zero nat1)) (.zero nat2))}
  {mor_eq_2 : {n : (.car nat1)}
    (EQ (h ((.suc nat1) n)) ((.suc nat2) (h n)))}
  (EQ (.fun nat1 nat2
    (MkNatMor nat1 nat2 h mor_eq_1 mor_eq_2)) h) )

( .mor_eq_1 : !! {nat1,nat2 : Nat}{h : (NatMor nat1 nat2)}
  (EQ (.fun nat1 nat2 h (.zero nat1)) (.zero nat2)) )

( .mor_eq_2 : !! {nat1,nat2 : Nat}{h : (NatMor nat1 nat2)}
  {n : (.car nat1)} (EQ (.fun nat1 nat2 h (.suc nat1 n))
    (.suc nat2 (.fun nat1 nat2 h n))) )

```

It is easy to verify that the type `Nat` of natural number algebras together with these morphisms constitutes a category precisely in the formal sense of Section 8.2.7 (definitions of identities and composition are omitted here). Therefore, we can apply our previous definition of initial objects to axiomatize `nat` as an initial object in this category.

```

( nat : Nat )

( nat_initial_mor : {nat' : Nat} (NatMor nat nat') )

( nat_initial_mor_unique : {nat' : Nat}
  {h1,h2 : (NatMor nat nat')}(EQ h1 h2) )

```

Observe that the first line corresponds to our loose OCC specification of natural numbers and the two axioms make explicit our informal initiality constraint.

In the following we show that this specification is strong enough to allow us to derive the elimination principle for natural numbers, which as we know contains the induction principle as a special case.

First of all, we observe that thanks to the projection functions `.mor_eq_1` and `.nat_mor_eq_2`, which are designated as computational equations, all morphisms have a useful computational behavior. In the following we are especially interested in `nat_initial_mor`, for which the computational behavior is verified by:

```
( initial_mor_eq_1 = ? : {nat' : Nat}
  (EQ (.fun nat nat' (nat_initial_mor nat') (.zero nat))
      (.zero nat')) )
all goals solved
```

```
( initial_mor_eq_2 = ? : {nat' : Nat}{n : (.car nat)}
  (EQ (.fun nat nat' (nat_initial_mor nat') (.suc nat n))
      (.suc nat' (.fun nat nat' (nat_initial_mor nat') n))) )
all goals solved
```

Our goal is to define the elimination principle (`nat_elim T z f n`) as a morphism from `nat` to a suitable algebra, which can depend on `T`, `z` and `f`. For the carrier of this algebra we use a dependent type of pairs which is specified below.

```
( car' : {T : (.car nat) -> Type} Type )
( mkcar' : {T : (.car nat) -> Type}
  {fst : (.car nat)}{snd : (T fst)} (car' T) )
( .fst : {T : (.car nat) -> Type} (car' T) -> (.car nat) )
( .snd : {T : (.car nat) -> Type} {p : (car' T)}(T (.fst T p)) )

( .fst_eq : !! {T : (.car nat) -> Type}
  {fst : (.car nat)}{snd : (T fst)}
  (EQ (.fst T (mkcar' T fst snd)) fst) )
( .snd_eq : !! {T : (.car nat) -> Type}
  {fst : (.car nat)}{snd : (T fst)}
  (EQ (.snd T (mkcar' T fst snd)) snd) )
```

The operations of this algebra are as follows:

```
( zero' = [T : (.car nat) -> Type][z : (T (.zero nat))]
  (mkcar' T (.zero nat) z) )

( suc' = [T : (.car nat) -> Type]
  [s : {n : (.car nat)}(T n) -> (T (.suc nat n))]
  [p : (car' T)](mkcar' T (.suc nat (.fst T p))
    (s (.fst T p) (.snd T p)) ) )

( nat' = [T : (.car nat) -> Type]
  [z : (T (.zero nat))]
  [s : {n : (.car nat)}(T n) -> (T (.suc nat n))]
  (MkNat (car' T) (zero' T z) (suc' T s)) )
```

```

( lemma = ? : !! {T : (.car nat) -> Type}
  {z : (T (.zero nat))}
  {s : {n : (.car nat)}(T n) -> (T (.suc nat n))}
  {n : (.car nat)}(EQ (.fst T (.fun nat (nat' T z s)
    (nat_initial_mor (nat' T z s)) n)) n) )
...

```

The previous lemma expresses the fact that the first component of the pair resulting from the application of the morphism `(nat_initial_mor (nat' T z s))` just tracks the original argument. This is a straightforward consequence of the uniqueness part, `nat_initial_mor_unique`, of initiality, which implies that the morphism from `nat` to itself is unique and hence equal to the identity morphism. Similar to proof assistants such as LEGO and COQ, OCC supports goal-oriented interactive proofs of such logical statements, an issue that will be treated in Section 8.3.

The feature of OCC allowing us to designate the previous lemma as a computational equality is the key to derive a computational meaningful elimination principle, which can now be defined by:

```

( nat_elim = [T : (.car nat) -> Type]
  [z : (T (.zero nat))]
  [s : {n : (.car nat)}(T n) -> (T (.suc nat n))]
  [n : (.car nat)]
  (.snd T (.fun nat (nat' T z s)
    (nat_initial_mor (nat' T z s)) n))
  : {T : (.car nat) -> Type}
  {z : (T (.zero nat))}
  {s : {n : (.car nat)}(T n) -> (T (.suc nat n))}
  {n : (.car nat)} (T n) )

```

Indeed, our original computational equations for `nat_elim` can be proved without further interaction:

```

( nat_elim_eq_1 = ? : {T : (.car nat) -> Type}
  {z : (T (.zero nat))}
  {f : {n : (.car nat)}(T n) -> (T (.suc nat n))}
  (EQ (nat_elim T z f (.zero nat)) z) )

( nat_elim_eq_2 = ? : {T : (.car nat) -> Type}
  {z : (T (.zero nat))}
  {f : {n : (.car nat)}(T n) -> (T (.suc nat n))}
  {n : (.car nat)}
  (EQ (nat_elim T z f (.suc nat n)) (f n (nat_elim T z f n))) )

```

To sum up, we have derived the elimination principle from initiality, and the entire derivation has been carried out inside the logic of OCC. The derived elimination principles is computationally meaningful, thanks to the equational properties of morphisms between algebras, which can be specified as computational equations in OCC. No separate treatment is needed for the induction principle, since it is just a special case of the elimination principle thanks to the propositions-as-types interpretation.

Typically, specifications are not isolated but they can depend on previous specifications. Our earlier specification of dependent lists (vectors of specified length over a given type) is a good example of a such a parameterized specification. The two parameters are given as an algebra `nat` with `zero` and `suc` and an arbitrary algebra `T`. In the context of these parameters, which we assume to be fixed for the moment, we define a class of list algebras `List` and axiomatize `list` as the initial object in this class.

```
( nat : Type )
( zero : nat )
( suc : nat -> nat )

( T : Type )

( List : Type )

( MkList : {car : nat -> Type}
           {nil : (car zero)}
           {cons : {n : nat} T -> (car n) -> (car (suc n))}
           List )

( .car : List -> nat -> Type )
( .nil : {list : List}{.car list zero} )
( .cons : {list : List}{n : nat}
         T -> (.car list n) -> (.car list (suc n)) )
...

( ListMor : {list1, list2 : List} Type )
( MkListMor :
  {list1, list2 : List}
  {h : {n : nat}{.car list1 n} -> (.car list2 n)}
  {h_eq_1 : (EQ (h zero (.nil list1)) (.nil list2))}
  {h_eq_2 : {n : nat}{x : T}{l : (.car list1 n)}
            (EQ (h (suc n) (.cons list1 n x l))
                (.cons list2 n x (h n l)))}
  (ListMor list1 list2) )
```

...

```

( list : List )

( list_initial_mor : {list' : List} (ListMor list list') )

( list_initial_mor_unique : {list' : List}
  {h1,h2 : (ListMor list list')}}(EQ h1 h2) )

```

For brevity we have omitted the specification of some projection functions.

Our next step is to make the parameterization explicit. In practice, we often work with a fixed algebra `nat` making explicit only the parameterization in `T` as in our earlier ad hoc specification of `list`, but here we consider the most general case where `list` is parametric over both algebras `T` and `nat`. To this end, we abstract over `nat` and `T`, and furthermore we consider morphisms between these parameter algebras.

```

( List : {nat : Nat}{T : Type} Type )

( MkList : {nat : Nat}{T : Type}
  {lcar : (.car nat) -> Type}
  {nil : (lcar (.zero nat))}
  {cons : {n : (.car nat)}
    T -> (lcar n) -> (lcar (.suc nat n))}
  (List nat T))

( .lcar : {nat : Nat}{T : Type}
  (List nat T) -> (.car nat) -> Type )

( .nil : {nat : Nat}{T : Type}
  {list : (List nat T)}(.lcar nat T list (.zero nat)) )

( .cons : {nat : Nat}{T : Type}
  {list : (List nat T)}{n : (.car nat)}
  T -> (.lcar nat T list n) ->
  (.lcar nat T list (.suc nat n)) )

...

```

```

( ListMor : {nat1,nat2 : Nat} {hnat : (NatMor nat1 nat2)}
  {T1,T2 : Type} {hT : (T1 -> T2)}
  {list1 : (List nat1 T1)}{list2 : (List nat2 T2)} Type )
( MkListMor : {nat1,nat2 : Nat} {hnat : (NatMor nat1 nat2)}
  {T1,T2 : Type} {hT : (T1 -> T2)}
  {list1 : (List nat1 T1)}{list2 : (List nat2 T2)}

```

```

{h : {n : (.car nat1)}(.lcar nat1 T1 list1 n) ->
  (.lcar nat2 T2 list2 (.fun nat1 nat2 hnat n))}
{h_eq_1 : (EQ (h (.zero nat1) (.nil nat1 T1 list1))
  (.nil nat2 T2 list2))}
{h_eq_2 : {n : (.car nat1)}
  {x : T1}{l : (.lcar nat1 T1 list1 n)}
  (EQ (h (.suc nat1 n) (.cons nat1 T1 list1 n x l))
    (.cons nat2 T2 list2 (.fun nat1 nat2 hnat n)
      (hT x) (h n l))))}
(ListMor nat1 nat2 hnat T1 T2 hT list1 list2) )
...

( list : {nat : Nat}{T : Type}(List nat T) )

( list_universal_mor :
  {nat,nat' : Nat} {hnat : (NatMor nat nat')}
  {T,T' : Type} {hT : (T -> T')}
  {list' : (List nat' T')}
  (ListMor nat nat' hnat T T' hT (list nat T) list') )

( list_universal_mor_unique :
  {nat,nat' : Nat} {hnat : (NatMor nat nat')}
  {T,T' : Type} {hT : (T -> T')}
  {list' : (List nat' T')}
  {h1,h2 : (ListMor nat nat' hnat T T' hT (list nat T) list')}
  (EQ h1 h2) )

```

An interesting aspect worth pointing out is that the axioms `h_eq_1` and `h_eq_2` of `MkListMor` can only be typed, because of the computational equations for morphisms between `Nat` algebras, which have been specified earlier. Hence, this is another good example which relies on the openness of the computational system of our type theory.

Note also how the initial object `list` becomes a universal object `(list nat T)` by virtue of parameterization. In other words, the initiality constraint is generalized to a freeness constraint, stating that `(list nat T)` is free over `nat` and `T`. Indeed, by means of the universal morphism we can extend `list` to a free functor in a unique way:

```

( list_mor =
  [nat,nat' : Nat] [hnat : (NatMor nat nat')]
  [T,T' : Type] [hT : (T -> T')]
  (list_universal_mor nat nat' hnat T T' hT (list nat' T'))

```

```

: {nat,nat' : Nat} {hnat : (NatMor nat nat')}
  {T,T' : Type} {hT : (T -> T')}
  (ListMor nat nat' hnate T T' hT (list nat T) (list nat' T')) )

```

In a similar way as we derived the elimination principle from the initiality of our specification of natural numbers, we could in this case derive the elimination principle and hence the induction principle for parameterized lists.

Until now we have discussed how initiality and freeness constraints can be expressed in OCC. A formalization of category theory in OCC is beyond the scope of this thesis. Such a formalization would provide a generic toolbox and the examples above would become particular instances of the general definitions of initiality and freeness.

The dual of initial and universal objects are final and couniversal objects respectively. As explained earlier, these notions form the basis for a general notion of behavioral or coinductive specifications. Below we illustrate the formalization of finality constraints using our earlier specification of streams over a fixed data type.

```

( T : Type )

( Stream : Type )

( MkStream : {scar : Type}
  {head : T}
  {tail : scar -> scar}
  Stream )

( .scar : Stream -> Type )
( .head : {stream : Stream} (.scar stream) -> T )
( .tail : {stream : Stream} (.scar stream) -> (.scar stream) )
...

( StreamMor : {stream1, stream2 : Stream} Type )
( MkStreamMor : {stream1, stream2 : Stream}
  {h : (.scar stream1) -> (.scar stream2)}
  {hax1 : {s : (.scar stream1)}
    (EQ (.head stream2 (h s)) (.head stream1 s))}
  {hax2 : {s : (.scar stream1)}
    (EQ (.tail stream2 (h s)) (h (.tail stream1 s)))}
  (StreamMor stream1 stream2) )
...

```

```
( stream : Stream )

( stream_final_mor : {stream' : Stream} (StreamMor stream stream') )

( stream_final_mor_unique : {stream' : Stream}
  {h1,h2 : (StreamMor stream stream')}}(EQ h1 h2) )
```

In complete analogy with the previous example, we can make the parameterization explicit to obtain the more general specification of parameterized streams as couniversal objects. The coelimination principle and the coinduction principle stated earlier can then be derived from this specification.

In contrast to the situation with inductive specifications, we are not aware of a coelimination principle which contains the coinduction principle as a special case. Indeed, this may be taken as an indication that the categorical formulation involving universal and couniversal objects provides a more uniform level of axiomatization for inductive and coinductive concepts, from which we can derive a variety of ad hoc (co)elimination and (co)induction principles for practical use.

Enriched Categories

Another interesting application of categories, that illustrates the features of OCC, is their use as transition categories to formalize the semantics of concurrent systems. As explained in Chapter 2, enriched transition categories are used in the semantics of rewriting logic, but they can also be used to give a direct semantics for other system models such as Petri nets, as they have been studied in Chapter 3.

Subsequently, we show how enriched categories, i.e. categories with additional algebraic structure, can be treated along the same lines as categories in the previous section. To pick a concrete example we use strict symmetric monoidal categories, which have already been used in Chapter 3 to provide a semantics of place/transition nets and more generally monoidal rewrite specifications. The example once again demonstrates the advantages of a type theory based on an open computational system, since typechecking essentially relies on the equational properties of the objects of the category which are markings in this case.

```
( Marking : Type )

( m_empty : Marking )
( m_par : Marking -> Marking -> Marking )

( m_par_comm : || {m,m' : Marking}
  (EQ (m_par m m') (m_par m' m)) )
```

```

( m_par_assoc : || {m,m',m'' : Marking}
  (EQ (m_par m (m_par m' m'')) (m_par (m_par m m') m'')) )

( m_par_right_id : || {m : Marking}
  (EQ (m_par m m_empty) m) )

( m_par_left_id : || {m : Marking}
  (EQ (m_par m_empty m) m) )

( Proc : Marking -> Marking -> Type )

( p_empty : {A | Marking} (Proc A A) )

( p_par : {A,B,C,D | Marking}
  (Proc A B) -> (Proc C D) -> (Proc (m_par A C) (m_par B D)) )

( p_seq : {A,B,C | Marking}
  (Proc A B) -> (Proc B C) -> (Proc A C) )

( p_par_comm : {A,B,C,D : Marking}
  {f : (Proc A B)}{g : (Proc C D)}
  (EQ (p_par f g) (p_par g f)) )

( p_par_assoc : {A,B,C,D,E,F : Marking}
  {f : (Proc A B)}{g : (Proc C D)}{h : (Proc E F)}
  (EQ (p_par f (p_par g h)) (p_par (p_par f g) h)) )

( p_par_right_id : {A, B : Marking}{f : (Proc A B)}
  (EQ (p_par f (p_empty | m_empty)) f) )

( p_par_left_id : {A, B : Marking}{f : (Proc A B)}
  (EQ (p_par (p_empty | m_empty) f) f) )

( p_empty_ax_l : {A,B : Marking}{f : (Proc A B)}
  (EQ (p_seq p_empty f) f) )

( p_empty_ax_r : {A,B : Marking}{f : (Proc A B)}
  (EQ (p_seq f p_empty) f) )

```

```

( assoc_ax : {A,B,C,D : Marking}
  {f : (Proc A B)}{g : (Proc B C)}{h : (Proc C D)}
  (EQ (p_seq (p_seq f g) h) (p_seq f (p_seq g h))) )

( func_ax_p_empty : {A,B : Marking}
  (EQ (p_par (p_empty | A) (p_empty | B)) (p_empty | (m_par A B))) )

( func_ax_seq : {A,B,C,A',B',C' : Marking}
  {f : (Proc A B)}{g : (Proc B C)}
  {f' : (Proc A' B')}{g' : (Proc B' C')}
  (EQ (p_par (p_seq f g) (p_seq f' g'))
      (p_seq (p_par f f') (p_par g g'))) )

```

The arrows of this category are (possibly nonatomic) proofs in the sense of rewriting logic. It is easy to see that this OCC specification extended by the transition rules of a given place/transition net (see example below) and an initiality constraint, is logically equivalent to the rewrite specification introduced in Chapter 3 as a rewrite semantics, if we take into account the fact that the single *sort* of processes is replaced by more specific *types* of processes with origin and destination as explicit parameters. Using this equivalence and the results of Chapter 3, we can conclude that we have indeed axiomatized the category of Best-Devillers processes for a given net.

To be more concrete, consider a place/transition net with three places `p11`, `p12`, and `p13`, sequentially connected by two transitions `tr12` and `tr23`. Then we can specify the category of Best-Devillers processes of this net if we extend the previous loose specification by

```

( p11 : Marking )
( p12 : Marking )
( p13 : Marking )

( tr12 : (Proc p11 p12) )
( tr23 : (Proc p12 p13) )

```

together with an initiality constraint for the entire specification.

An example of a particular process constructed by goal-oriented refinement is given next.

```

( pr = ? : (Proc (m_par p11 p12) (m_par p13 p13)) )

{ tr12 : ( Proc p11 p12 ) }
{ tr23 : ( Proc p12 p13 ) }

```

```

-----
?799 : ( Proc ( m_par p11 p12 ) ( m_par p13 p13 ) )
1 new goal

( Inst ( p_par | p11 | p13 | p12 | p13 ? ? ) )

{ tr12 : ( Proc p11 p12 ) }
{ tr23 : ( Proc p12 p13 ) }
-----
?840 : ( Proc p12 p13 )

{ tr12 : ( Proc p11 p12 ) }
{ tr23 : ( Proc p12 p13 ) }
-----
?839 : ( Proc p11 p13 )
2 new goals

( Inst ( p_seq tr12 tr23 ) )
1 goal solved (no new goals)

( Inst tr23 )
all goals solved

{ pr = ( p_par ( p_seq tr12 tr23 ) tr23 )
      : ( Proc ( m_par p11 p12 ) ( m_par p13 p13 ) ) }

```

To sum up, we have exploited here the ability of OCC to formalize enriched categories in a very direct way. A considerable improvement w.r.t. the logically equivalent representation in MEL of Chapter 3 is the strongly typed nature of the OCC representation. There is no need to introduce auxiliary sorts/types, and the fact that an OCC term denotes a rewrite proof, indeed a Best-Devillers process in this case, is ensured by type checking alone.

8.2.8 Executable Rewrite Specifications

In several earlier examples we have exploited the fact that the underlying membership equational logic of rewriting logic is also a sublogic of OCC, and in fact is generalized by its embedding into OCC in several directions such as the possibility of using full Horn clause logic and extensions allowing us to write higher-order equational specifications with dependent types. In this section we will show, by means of several examples, how this relationship can be extended to rewriting logic, making rewriting logic a sublogic of OCC via a simple embedding, which is

entirely analogous to the embedding into MEL of Section 2.4.5. We furthermore show how the embedded rewrite specifications can be actually executed in OCC. Assuming a suitable specification of natural numbers `nat`, we consider the following rewrite specification:

```
sort st .

op loop : nat -> st .
op stop : -> st .

var i : nat .

cr1 [a1] : loop(i) => loop(minus(i,1)) if lt(0,i) .

r1 [a2] : loop(0) => stop .
```

It can be represented by the following OCC specification:

```
( st : Type )
( act : Type )

( a1 : act ) ( a2 : act )

( loop : nat -> st )
( stop : st )

( rule : st -> st -> act -> Prop )

( rule_a1 : !! {i : nat} (nat_lt 0 i) ->
  (rule (loop i) (loop (nat_minus i 1)) a1) )

( rule_a2 : !!
  (rule (loop 0) stop a2) )
```

As usual, the `!!` syntax marks operationally relevant parts of the specification. It introduces `rule` as a rewrite predicate with two rewrite axioms `rule_a1` and `rule_a2`, the first axiom being a conditional one.

To execute the specification using the default strategy, which repeatedly performs one-step sequential rewrites rules in an unspecified order until no further rules can be applied, we write

```
( Rew rule (loop 3) )
stop
```

Rewriting logic has rules to generate reflexive, transitive, and context closure of the rewrite relation induced by the rules. In this example, however, the state space st is of a rather simple nature, in particular there is no parallel composition operator and hence there are no corresponding context rules. So it remains to specify the reflexive and transitive closure. Not surprisingly this leads to a category which can be specified following the approach of Section 8.2.7, giving rise to the following parameterized type **Proc** of sequential processes, which we require in accordance with the semantics of rewriting logic to be freely generated by its constructors and equations.

```
( Proc : st -> st -> Type )

( p_rule : {A,B | st}{e | act}
  (rule A B e) -> (Proc A B) )

( p_empty : {A | st}
  (Proc A A) )

( p_seq : {A,B,C | st}
  (Proc A B) -> (Proc B C) -> (Proc A C) )

( p_empty_ax_l : {A,B : st}{f : (Proc A B)}
  (EQ (p_seq p_empty f) f) )

( p_empty_ax_r : {A,B : st}{f : (Proc A B)}
  (EQ (p_seq f p_empty) f) )

( assoc_ax : {A,B,C,D : st}
  {f : (Proc A B)}{g : (Proc B C)}{h : (Proc C D)}
  (EQ (p_seq (p_seq f g) h) (p_seq f (p_seq g h))) )
```

This type of processes can be used to specify a rewrite predicate **trans*** that is the reflexive and transitive closure of **rule**.

```
( trans* : {A,B : st}(Proc A B) -> Prop )

( t*_rule : !! {A,B : st}{e : act}{p : (rule A B e)}
  (trans* A B (p_rule p)) )

( t*_empty : !! {A : st}
  (trans* A A (p_empty | A)) )
```

```
( t*_seq : !! {A,B,C : st}{p : (Proc A B)}{q : (Proc B C)}
  (trans* A B p) -> (trans* B C q) -> (trans* A C (p_seq p q)) )
```

Since according to the semantics of rewriting logic, each rewrite specification has an implicit freeness constraint, stating that it is free over its data subspecification, the specification of `nat` in this case, we should add a corresponding constraint for the OCC specification. For better structuring, however, it would be preferable to decompose this single constraint into separate freeness constraints for `st`, `act`, `rule`, `Proc`, and `trans*`.

Monoidal Rewrite Specifications

In a similar way we can use OCC to express the rewriting semantics of the place/transition net example introduced as the banker's problem in Chapter 3. In contrast to the previous specification we have a nontrivial state space, which is equipped with a monoidal structure. We first recall the specification in rewriting logic:

```
sort Marking .

op empty : -> Marking .
op __ : Marking Marking -> Marking [assoc comm id: empty] .

ops BANK CREDIT-1 CREDIT-2 CLAIM-1 CLAIM-2 : -> Marking .

r1 [GRANT-1] : BANK CLAIM-1 => CREDIT-1 .

r1 [RETURN-1] : CREDIT-1 CREDIT-1 CREDIT-1 =>
  BANK BANK BANK CLAIM-1 CLAIM-1 CLAIM-1 .

r1 [GRANT-2] : BANK CLAIM-2 => CREDIT-2 .

r1 [RETURN-2] : CREDIT-2 CREDIT-2 =>
  BANK BANK CLAIM-2 CLAIM-2 .
```

This rewrite specification can be represented in OCC as follows:

```
( Marking : Type )
( m_empty : Marking )
( m_par : Marking -> Marking -> Marking )
```

```

( m_par_comm : || {m,m' : Marking}
  (EQ (m_par m m') (m_par m' m)) )

( m_par_assoc : || {m,m',m'' : Marking}
  (EQ (m_par m (m_par m' m'')) (m_par (m_par m m') m'')) )

( m_par_right_id : || {m : Marking}
  (EQ (m_par m m_empty) m) )

( m_par_left_id : || {m : Marking}
  (EQ (m_par m_empty m) m) )

( BANK : Marking )
( CREDIT-1 : Marking ) ( CREDIT-2 : Marking )
( CLAIM-1 : Marking ) ( CLAIM-2 : Marking )

( act : Type )
( GRANT-1 : act ) ( RETURN-1 : act )
( GRANT-2 : act ) ( RETURN-2 : act )

( rule : Marking -> Marking -> act -> Prop )

( rule_grant_1 : !!
  ( rule (m_par BANK CLAIM-1)
    CREDIT-1
    GRANT-1 ) )

( rule_return_1 : !!
  ( rule (m_par CREDIT-1 (m_par CREDIT-1 CREDIT-1))
    (m_par BANK (m_par BANK (m_par BANK
      (m_par CLAIM-1 (m_par CLAIM-1 CLAIM-1))))))
    RETURN-1 ) )

( rule_grant_2 : !!
  ( rule (m_par BANK CLAIM-2)
    CREDIT-2
    GRANT-2 ) )

( rule_return_2 : !!
  ( rule (m_par CREDIT-2 (m_par CREDIT-2 CREDIT-2))
    (m_par BANK (m_par BANK (m_par BANK
      (m_par CLAIM-2 (m_par CLAIM-2 CLAIM-2))))))
    RETURN-2 ) )

```

Since the operational semantics of rewriting logic allows rewriting in arbitrary contexts, and since OCC intentionally avoids any built-in rules for rewrite predicates, we have to explicitly specify suitable context rules for each parallel composition operator. Here, we only have one commutative operator `m_par`, so that the single rule `t_par` below suffices.

```
( trans : Marking -> Marking -> act -> Prop )

( t_rule : !! {m,m' : Marking}{e : act}
  (rule m m' e) -> (trans m m' e) )

( t_par : !! {m,m',m'' : Marking}{e : act}
  (trans m' m'' e) -> (trans (m_par m m') (m_par m m'') e) )
```

A sample execution using the default strategy is the following:

```
(Rew trans (m_par BANK (m_par BANK
  (m_par (m_par CLAIM-1 (m_par CLAIM-1 CLAIM-1))
    (m_par (m_par CLAIM-2 CLAIM-2))))))

( m_par ( m_par ( m_par CREDIT-1 CREDIT-1 ) CLAIM-1 )
  ( m_par ( m_par CLAIM-2 CLAIM-2 ) ) )
```

The default strategy of OCC corresponds to the default strategy of Maude, which repeatedly performs one-step sequential rewriting at the top of a term. User-definable strategies as they are available in Maude are not supported in the present version of the OCC prototype. Such strategies would be needed in practice to deal with nondeterministic and potentially nonterminating systems. Other features of Maude, such as state space exploration and temporal logic model checking, could also be lifted to the OCC level, thanks to the reflective architecture of the OCC prototype.

To accommodate the reflexivity, transitivity, and the context rules of rewriting logic, we follow the approach to enriched categories of Section 8.2.7 giving rise to the following parameterized type of concurrent processes `Proc`.

```
( Proc : Marking -> Marking -> Type )

( p_rule : {A,B : Marking}{e : act}
  (rule A B e) -> (Proc A B) )

( p_empty : {A : Marking}
  (Proc A A) )
```

```

( p_par : {A,B,C,D : Marking}
  (Proc A B) -> (Proc C D) -> (Proc (m_par A C) (m_par B D)) )

( p_seq : {A,B,C : Marking}
  (Proc A B) -> (Proc B C) -> (Proc A C) )
...

( trans* : {A,B : Marking}(Proc A B) -> Prop )

( t*_rule : !! {A,B : Marking}{e : act}{p : (rule A B e)}
  (trans* A B (p_rule p)) )

( t*_empty : !! {A : Marking}
  (trans* A A (p_empty | A)) )

( t*_par : !! {A,B,C,D : Marking}{p : (Proc A B)}{q : (Proc C D)}
  (trans* A B p) -> (trans* C D q) ->
  (trans* (m_par A C) (m_par B D) (p_par p q)) )

( t*_seq : !! {A,B,C : Marking}{p : (Proc A B)}{q : (Proc B C)}
  (trans* A B p) -> (trans* B C q) -> (trans* A C (p_seq p q)) )

```

We refer to our specification of enriched categories in Section 8.2.7 for the omitted equations. To properly reflect the model-theoretic semantics of rewriting logic we should finally add freeness constraints for `st` and `act` (which reduce to initiality constraints, since there are no parameters), and freeness constraints for `rule`, `trans`, and `Proc`.

In spite of the generality offered by RWL and OCC, it is worth pointing out that under the rewrite strategies currently used by the OCC prototype and in Maude the increased generality of `trans*` w.r.t. `trans` is not exploited for two reasons: first, rewrite proofs are presently not computed by the system, and second, all rewrites applied by the strategy are on step-sequential rewrites that are one-step sequential rewrites that are already covered by `trans*`. On the other hand, our strongly typed representation of general rewrites, would be useful for more general rewrite strategies and for reasoning about executions in the logic of OCC.

Colored Net Specifications

The benefits of regarding rewriting logic as a sublogic of OCC are best demonstrated by a nontrivial application. To this end, we reconsider the rewrite specification which emerged from the case study in Chapter 3. In this case study,

the echo algorithm was taken as a typical example of a distributed network algorithm, which can naturally be modeled as an algebraic net specification and, via the given rewriting semantics, can be translated into a rewrite specification. In this section we exploit the capabilities of OCC to obtain an equivalent strongly typed specification. This task is nontrivial, since the state space consists of a single untyped soup of objects, represented by the kind `Marking`, and has to satisfy nonstandard structural equations.

Reusing our earlier specification of finite multisets we first specify a parameterized type of markings. The type `(Marking V C)` denotes all markings over a type `V` of places that are typed according to `C`. Markings have three constructors `m_empty`, `m_place`, and `m_union`. As in Chapter 3 we impose the structural equations for a commutative monoid, together with place linearity equations as computational equations. We should add a freeness constraint for `(Marking V C)`, to express that we are concerned with a parameterized equational inductive definition.

```
( Marking : {V : Type}{C : V -> Type} Type )

( m_empty : {V | Type}{C | V -> Type}
  (Marking V C) )

( m_place : {V | Type}{C | V -> Type}{v : V}
  (fms (C v)) -> (Marking V C) )

( m_union : {V | Type}{C | V -> Type}
  (Marking V C) -> (Marking V C) -> (Marking V C) )

( m_union_comm : || {V : Type}{C : V -> Type}
  {m,m' : (Marking V C)}
  (EQ (m_union m m') (m_union m' m)) )

( m_union_assoc : || {V : Type}{C : V -> Type}
  {m,m',m'' : (Marking V C)}
  (EQ (m_union m (m_union m' m'')) (m_union (m_union m m') m'')) )

( m_union_id : !! {V : Type}{C : V -> Type}{m : (Marking V C)}
  (EQ (m_union m m_empty) m) )

( m_place_linearity_eq_0 : !! {V : Type}{C : V -> Type}{v : V}
  (EQ (m_place (fms_empty | (C v))) (m_empty | V | C)) )
```

```

( m_place_linearity_eq_1 : !! {V : Type}{C : V -> Type}
  {v : V}{b,b' : (fms (C v))}
  (Not (fms_is_empty b)) -> (Not (fms_is_empty b')) ->
  (EQ (m_place v (fms_union b b'))
    (m_union (m_place v b) (m_place v b')))) )

( m_single = [V | Type][C | V -> Type][v : V][x : (C v)]
  (m_place v (fms_single x)) )

```

We furthermore equationally specify a projection function, which in a strongly typed fashion allows us to extract the contexts of a particular place for a given marking.

```

( m_proj : {V | Type}{C | V -> Type}
  (Marking V C) -> {v : V}(fms (C v)) )

( m_proj_eq_1 : !! {V : Type}{C : V -> Type}{v : V}
  (EQ (m_proj (m_empty | V | C) v) (fms_empty | (C v)))) )

( m_proj_eq_2 : !! {V : Type}{C : V -> Type}
  {v,v' : V}{b : (fms (C v'))} (!! (EQ v v')) ->
  (EQ (m_proj (m_place v b) v') b) )

( m_proj_eq_3 : !! {V : Type}{C : V -> Type}
  {v,v' : V}{b : (fms (C v))} (Not (EQ v v')) ->
  (EQ (m_proj (m_place v b) v') (fms_empty | (C v')))) )

( m_proj_eq_4 : !! {V : Type}{C : V -> Type}
  {m,m' : (Marking V C)}{v : V}
  (EQ (m_proj (m_union m m') v)
    (fms_union (m_proj m v) (m_proj m' v)))) )

```

To simplify the presentation we have slightly deviated (operationally but not logically) from the rewrite specification in Chapter 3 by specifying `m_union_id` as a computational rather than a structural equation. The use of a structural equation is equally possible, but we would have to add conditions in the specification of `m_proj` to ensure termination, similar to the conditions used in the specification of finite multisets.

Now we translate the membership equational specification underlying the algebraic net specification given in Chapter 3 into an equivalent but strongly typed OCC specification:

```

( id : Type )

```

```

( Pair = (prod id id) )

( network : (fms Pair) )

( outgoing : (fms Pair) -> id -> (fms Pair) )

( outgoing_eq_1 : !! {x' : id}
  (EQ (outgoing (fms_empty | Pair) x') (fms_empty | Pair)) )

( outgoing_eq_2 : !! {x,y,x' : id}{g : (fms Pair)}
  (EQ x x') ->
  (EQ (outgoing (fms_union (fms_single (pair x y)) g) x')
    (fms_union (fms_single (pair y x)) (outgoing g x')))) )

( outgoing_eq_3 : !! {x,y,x' : id}{g : (fms Pair)}
  (Not (EQ x x')) ->
  (EQ (outgoing (fms_union (fms_single (pair x y)) g) x')
    (outgoing g x')) )

( incoming : (fms Pair) -> id -> (fms Pair) )

( incoming_eq_1 : !! {y' : id}
  (EQ (incoming (fms_empty | Pair) y') (fms_empty | Pair)) )

( incoming_eq_2 : !! {x,y,y' : id}{g : (fms Pair)}
  (EQ y y') ->
  (EQ (incoming (fms_union (fms_single (pair x y)) g) y')
    (fms_union (fms_single (pair y x)) (incoming g y')))) )

( incoming_eq_3 : !! {x,y,y' : id}{g : (fms Pair)}
  (Not (EQ y y')) ->
  (EQ (incoming (fms_union (fms_single (pair x y)) g) y')
    (incoming g y')) )

```

Our next step is the translation of the part of the rewrite specification that has been obtained from the original net specification of the echo algorithm. To this end, we need to define the type of states as `(Marking p1 p1_color)` for suitable choices of `p1` and `p1_color`, which are introduced below. We should add constraints stating that both `p1` and `color_name` are freely generated by their respective constructors.

```

( p1 : Type )

```

```

( QUIET : pl ) ( WAITING : pl ) ( TERMINATED : pl )
( UNINFORMED : pl ) ( PENDING : pl ) ( ACCEPTED : pl )
( MESSAGES : pl )

( not_eq_QUIET_WAITING : ?? (Not (EQ QUIET WAITING)) )
( not_eq_QUIET_TERMINATED : ?? (Not (EQ QUIET TERMINATED)) )
( not_eq_QUIET_UNINFORMED : ?? (Not (EQ QUIET UNINFORMED)) )
...

( color_name : Type )

( cid : color_name )
( cpair : color_name )

( not_eq_cid_cpair : ?? (Not (EQ cid cpair)) )

( color : color_name -> Type )

( color_name_eq_1 : !! (EQ (color cid) id) )
( color_name_eq_2 : !! (EQ (color cpair) Pair) )

( type : color_name -> Type )

( type_eq_1 : !! (EQ (type cid) (fms id)) )
( type_eq_2 : !! (EQ (type cpair) (fms Pair)) )

( pl_color_name : pl -> color_name )

( pl_color_name_eq_1 : !! (EQ (pl_color_name QUIET) cid) )
( pl_color_name_eq_2 : !! (EQ (pl_color_name WAITING) cid) )
( pl_color_name_eq_3 : !! (EQ (pl_color_name TERMINATED) cid) )
( pl_color_name_eq_4 : !! (EQ (pl_color_name UNINFORMED) cid) )
( pl_color_name_eq_5 : !! (EQ (pl_color_name PENDING) cpair) )
( pl_color_name_eq_6 : !! (EQ (pl_color_name ACCEPTED) cid) )
( pl_color_name_eq_7 : !! (EQ (pl_color_name MESSAGES) cpair) )

( pl_color = [v : pl] (color (pl_color_name v)) )
( pl_type = [v : pl] (fms (pl_color v)) )

( st = (Marking pl pl_color) )

( m_single_QUIET = [x : id] (m_single QUIET x) )

```

```

( m_single_WAITING = [x : id] (m_single WAITING x) )
( m_single_TERMINATED = [x : id] (m_single TERMINATED x) )
( m_single_UNINFORMED = [x : id] (m_single UNINFORMED x) )
( m_single_PENDING = [x,y : id] (m_single PENDING (pair x y)) )
( m_single_MESSAGES = [x,y : id] (m_single MESSAGES (pair x y)) )
( m_single_ACCEPTED = [x : id] (m_single ACCEPTED x) )

( m_MESSAGES = [s : (fms Pair)] (m_place MESSAGES s) )

```

The atomic actions of the rewrite specification are specified by the following type act.

```

( act : Type )

( ISEND : {x : id} act )
( IRECEIVE : {x : id}{m : st} act )
( SEND : {x,y : id} act )
( RECEIVE : {x,y : id}{m : st} act )

```

Finally, we translate each rule of the rewrite specification:

```

( rule : st -> st -> act -> Prop )

( rule_ISEND : !! {x : id}
  (rule (m_single_QUIET x)
    (m_union (m_single_WAITING x)
      (m_MESSAGES (outgoing network x)))
    (ISEND x)) )

( rule_IRECEIVE : !! {x : id}{m : st}
  (EQ m (m_MESSAGES (incoming network x))) ->
  (rule (m_union (m_single_WAITING x) m)
    (m_single_TERMINATED x)
    (IRECEIVE x m)) )

( rule_SEND : !! {x,y : id}
  (rule (m_union (m_single_UNINFORMED x) (m_single_MESSAGES x y))
    (m_union (m_single_PENDING x y)
      (m_MESSAGES (fms_rm (outgoing network x)
        (pair y x))))
    (SEND x y)) )

```



```
(fms_union (sym c i)
(fms_union (sym c a)
           (sym a b))))))))) )
```

Finally, we can execute the specification by

```
( Rew trans
  (m_union (m_single_QUIET i)
(m_union (m_single_UNINFORMED a)
(m_union (m_single_UNINFORMED b)
(m_union (m_single_UNINFORMED c)
(m_union (m_single_UNINFORMED d)
          (m_single_UNINFORMED e)))))) )

( m_union ( m_union ( m_union ( m_union ( m_union
  ( m_place TERMINATED ( fms_single i ) )
  ( m_place ACCEPTED ( fms_single e ) ) ) )
  ( m_place ACCEPTED ( fms_single a ) ) ) )
  ( m_place ACCEPTED ( fms_single b ) ) ) )
  ( m_place ACCEPTED ( fms_single c ) ) ) )
  ( m_place ACCEPTED ( fms_single d ) ) ) )
```

On top of this specification, we can again specify an enriched category of rewrite proofs using the approach of Section 8.2.7. This is entirely analogous to the place/transition net case of the previous subsection and has therefore been omitted here.

To conclude this section we would like to point out that, in contrast to the examples given earlier in this section, the representation of colored net specifications in OCC is not just a direct translation of the rewrite semantics of the echo algorithm given as an algebraic net specification in Chapter 3, but it instead uses the distinctive features of OCC, namely parameterized and dependent types to obtain a strongly typed representation. In this way, we can represent arbitrary colored net specifications over OCC, which generalize colored net specifications over MEL.

8.3 Examples in Interactive Theorem Proving

In the previous section we have exploited the internal logic of OCC for specification purposes. As in other logical type theories which allow a propositions-as-types interpretation, interactive theorem proving becomes equivalent to typed programming, in the generalized sense that well-typed programs are built from a

collection of hypothetical program components. Since the assumed components may be specified axiomatically and may not even be implementable, i.e. definable inside the type theory, the notion of a program is relative to such hypothetical components, and hence more general than the usual notion of a program in a typical programming language.

8.3.1 Goal-Oriented Refinement

As in other type theories such as LEGO and COQ, programs and proofs can be constructed by goal-oriented refinement. To represent incomplete programs/proofs, OCC simply uses metavariables, which can either be automatically solved by the system or can be manually instantiated by the user. It is noteworthy that these metavariables are precisely the same metavariables that have been used to abbreviate specifications in the previous section, but in the context of interactive theorem proving user interaction is required to solve typical goals.

To give a first idea of interactive theorem proving in OCC, we prove the trivial theorem that the previously introduced logical operator `And` is symmetric. Starting out from a goal with an unknown proof, written as `?`, we successively refine the incomplete proof using `(Inst M)` which is equivalent to `(?n := M)` where `?n` is the metavariable on the top of the stack of current goals, which is displayed in reverse order.

```
(A : Prop)
(B : Prop)

( and_is_symmetric = ? : (And A B) -> (And B A) )

{ A : Prop }
{ B : Prop }
-----
?387 : ( ( ( And A ) B ) -> ( ( And B ) A ) )
1 new goal

(Inst [H : ?] ?)

{ A : Prop }
{ B : Prop }
{ H : ( ( And A ) B ) }
-----
?422 : ( ( And B ) A )
1 new goal
```

```

(Inst (And_intro ? ? ? ?))

{ A : Prop }
{ B : Prop }
{ H : ( ( And A ) B ) }
-----
?446 : A

{ A : Prop }
{ B : Prop }
{ H : ( ( And A ) B ) }
-----
?445 : B
2 new goals

( Inst (And_elim_r ? ? H) )
1 goal solved (no new goals)

( Inst (And_elim_l ? ? H) )
all goals solved

```

After completing the proof the context contains the full proof term together with its type, i.e. the proved theorem:

```

{ A : Prop }
{ B : Prop }
{ C : Prop }
{ and_symmetry = [ H : ( And A B ) ]
                  ( And_intro B A
                    ( And_elim_r A B H )
                    ( And_elim_l A B H ) )
                  : ( ( And A B ) -> ( And B A ) ) }

```

As mentioned earlier, the use of goal-oriented refinement is by no means restricted to the refinement of propositions, i.e. to the construction of proofs, but is applicable to arbitrary types. For instance, a goal-oriented construction of the factorial function introduced earlier can be conducted as follows.

```

( nat_fact = ? : nat -> nat )
( Inst (nat_rec ? ?) )
( Inst 1 )
( Inst [n : nat] [nat_factn : nat] ? )
( Inst (nat_mult (suc n) nat_factn) )

```

Indeed, `nat_fact` is a program defined relative to the hypothetical components `nat_rec`, `nat_mult`, `1`, and `suc`. The function `nat_mult` has previously been axiomatized by an executable specification, but it could also be implemented, i.e. could be defined as a program, relative to `nat_rec`, `0`, and `nat_plus`.

8.3.2 Reasoning with Equality

We show the use of Leibnitz equality in a simple example, in which we prove disequality of two elements of an inductive type. The example simultaneously shows how the elimination principle for an inductive type ensures that constructors can be distinguished (“no confusion” in the terminology of algebraic specifications, cf. Chapter 2). The general idea is to use the elimination principle to map elements of the inductive type in question to elements of a different type, which can be distinguished. In our case we use `Prop` as a target type together with its distinguishable elements `True` and `False`.

```
( bool_true_not_eq_false = ? : (Not (EQ true false)) )

-----
?3787 : ( Not ( EQ true false ) )
1 new goal

( Inst (Not_intro ? ?) )

-----
?3813 : ( ( EQ true false ) -> False )
1 new goal

( Inst [H : ?] ?)

{ H : ( ( EQ true ) false ) }
-----
?3838 : False
1 new goal

( Inst [H' = (eq_imp_leq ? ? ? H) : ?] ?)

{ H : ( EQ true false ) }
{ H' = ( eq_imp_leq bool true false H )
  : ( { P : ( bool -> Prop ) }
    ( ( P true ) -> ( P false ) ) ) }
}
```

```

-----
?3857 : False
1 new goal

( Inst [G = (H' bool_to_prop) : ?] ? )

{ H : ( EQ true false ) }
{ H' = ( ( eq_imp_leq bool true false H )
        : ( { P : ( bool -> Prop ) }
            ( ( P true ) -> ( P false ) ) ) ) }
{ G = ( ( H' bool_to_prop )
        : ( ( bool_to_prop true ) -> ( bool_to_prop false ) ) ) }
-----
?3890 : False
1 new goal

( Inst (G ?) )
all goals solved

```

Explicitly performing each single step of a complex equality proof in the style above can be cumbersome and has been recognized as one of the main drawbacks of existing extensions of the calculus of constructions. This problem is usually addressed, e.g. in LEGO and COQ, by the addition of tactics that facilitate reasoning with equality, but in our view this is not entirely satisfactory, since tactics are external to the type theory, and furthermore not taken into account in the operational semantics, on which type checking is based.

In OCC we can take advantage of the direct support for equality in the type theory itself. We have already seen that each OCC context induces an operational semantics, given by its structural and computational equations and its assertional axioms. Interactive theorem proving takes place in typically changing contexts and can exploit the operational semantics available at each stage.

Equality proofs can be conducted in two common ways. In the first place, an equality assumption can be used as a computational equation, leading to a proof by reduction, as in the following example:

```

( T : Type )

( a : T ) ( b : T ) ( c : T )

( eq_symmetry_1 = ? : (EQ a c) -> (EQ c a) )

```

```
( Inst [H : !! (EQ a c)] ? )
all goals solved
```

```
{ eq_symmetry_1 = ( ( [ H : !! ( EQ a c ) ]
  ( EPSILON ( EQ c a ) ) ) ) : ( EQ a c ) -> ( EQ a c ) ) }
```

Alternatively, and this possibility is available for arbitrary predicates, we can use an equality assumption as an operational axiom, thereby enabling a proof by exhaustive goal-oriented search. Unlike other predicates, `EQ` has built-in rules for reflexivity and symmetry. Therefore, a proof of `(EQ a c)` also counts as a proof of `(EQ c a)`, as we can see in the next example:

```
( eq_symmetry_2 = ? : (EQ a c) -> (EQ c a) )
```

```
( Inst [H : ?? (EQ a c)] ? )
all goals solved
```

```
{ eq_symmetry_2 = ( ( [ H : ?? ( EQ a c ) ]
  ( EPSILON ( EQ c a ) ) ) ) : ( EQ a c ) -> ( EQ a c ) ) }
```

In both examples the generated proof term contains a term of the form (ϵP) of type P , written as `(EPSILON P)` above. This term simply denotes an unspecified proof of P . The existence of a proof is verified operationally, i.e. by means of exhaustive goal-oriented search. Recall that goal-oriented search does not only apply assertional propositions, but can also involve reduction using computational equations (possibly modulo structural equations). In this way, OCC proofs abstract away from certain subproofs which are of purely computational nature, a feature which not only reduces the complexity of proof terms but also provides a well-defined form of automation inside the type theory rather than at the metalevel.

8.3.3 Inductive Theorem Proving

From the viewpoint of a logical type theory, inductive theorem proving is the use of primitive recursion to construct proofs from elements of an inductive type. In other words, the induction principles are a special case of elimination principles, but it is the capability to cover all elements of an inductive type rather than the capability to distinguish elements which is relevant for inductive proofs. As a first example, the induction principle `bool_ind` for the type `bool` is a special case of the elimination principle `bool_elim`, and could therefore be defined in terms of `bool_elim` rather than introducing it by the axiom below. We use it in the following example to prove a simple boolean equality.

```

( bool_ind : {P : bool -> Prop}
  (P true) ->
  (P false) ->
  {b : bool} (P b) )

( not_not_eq = ? : !! {b : bool} (EQ (not (not b)) b) )

-----
?1432 : ( !! ( { b : bool } ( EQ (not (not b)) b ) ) )

( Inst (bool_ind ([b : ?] ?) ? ?))
all goals solved

```

The proved theorem is designated as an operational theorem by virtue of the !! tag. In this way we have specified that it should be added to the context as a computational equality, thereby enriching the computational system of the type theory.

```

(b : bool)

( Red (not (not b)) )
b

```

In a similar way, we can specialize `nat_elim` to an induction principle `nat_ind` for the type `nat`. Here we use it to prove monotonicity of the predicate `nat_le`.

```

( nat_ind : {P : nat -> Prop}
  (P 0) ->
  ({i : nat} (P i) -> (P (suc i))) ->
  {n : nat} (P n) )

( nat_suc_is_monotone = ? : {n : nat} (nat_le n (suc n)) )

-----
?7585 : { n : nat } ( nat_le n ( suc n ) )
1 new goal

( Inst (nat_ind ([n : ?] ?) ? ?) )

```

```

-----
?7616 : { i : nat }
  ( ( [ n : nat ] ( nat_le n ( suc n ) ) ) i ) ->
  ( ( [ n : nat ] ( nat_le n ( suc n ) ) ) ( suc i ) )
1 new goal

( Inst ([i : nat] ?) )

{ i : nat }
-----
?7693 : ( ( [ n : nat ] ( nat_le n ( suc n ) ) ) i ) ->
  ( ( [ n : nat ] ( nat_le n ( suc n ) ) ) ( suc i ) )
1 new goal

( Inst ([H : ?? ?] ?) )
all goals solved

```

The last step of this inductive proof is noteworthy, since by forcing the premise H into the form $?? ?$ we express that it should be added to the context as an assertional equality, to support an automatic proof of the right hand side of the implication.

Another example of an inductive proof in OCC is commutativity of `nat_plus`, assuming that it has not been specified axiomatically:

```

( nat_plus : nat -> nat -> nat )

( nat_plus_eq_1 : !! {i : nat}
  (EQ (nat_plus 0 i) i) )
( nat_plus_eq_2 : !! {i : nat}
  (EQ (nat_plus i 0) i) )
( nat_plus_eq_3 : !! {i,j : nat}
  (EQ (nat_plus (suc i) j) (suc (nat_plus i j))) )
( nat_plus_eq_4 : !! {i,j : nat}
  (EQ (nat_plus i (suc j)) (suc (nat_plus i j))) )

( nat_plus_is_commutative =
  ? : || {m,n : nat} (EQ (nat_plus m n) (nat_plus n m)) )
-----
?1595 : || { m : nat } { n : nat } ( EQ ( plus m n ) ( plus n m ) )
1 new goal

```

```
( Inst (nat_ind ([n : nat] ?) ? ?) )

-----
?1652 : { i : nat }
      ( ( [ n : nat ] { n : nat }
          ( EQ ( plus n{1} n ) ( plus n n{1}) ) ) i ) ->
      ( ( [ n : nat ] ( { n : nat }
          ( EQ ( plus n{1} n ) ( plus n n{1}) ) ) ) ( suc i ) )
1 new goal

( Inst [i : ?][H : !! ?] ? )
all goals solved
```

As a side remark, this example reminds us of the fact that OCC uses the CINNI calculus to deal with bound variables. Variables are indexed names, but for convenience we always write $n\{0\}$ as n , thereby referring to the innermost binder for n , which is the usual case. Above, we have another variable called $n\{1\}$ which allows us to skip the innermost binder for n , thereby referring to the outermost binder for n in this case.

Coming back to the main issue, we observe that the proof is again partially automated, but here by forcing the premise into the form of a computational equation as expressed by the pattern `!! ?`. Furthermore, `nat_plus_is_commutative` is designated as a structural equations by `||`, and therefore enriches the computational system associated with this context.

To demonstrate that inductive proofs over parameterized datatypes can be done in a similar style with partial automation, we show associativity of the polymorphic function `append` for polymorphic lists with constructors `nil` and `cons`. Again the induction principle could be obtained from a more general elimination principle for parameterized data types, e.g. the one described in [CPM90].

```
( list_ind : {T : Type}{P : (list T) -> Prop}
  (P nil) ->
  ({h : T}{t : (list T)} (P t) -> (P (cons h t))) ->
  {l : (list T)} (P l) )

( append : {T | Type} (list T) -> (list T) -> (list T) )

( append_eq_1 : !! {T : Type}{l : (list T)}
  (EQ (append nil l) l) )
```

```

( append_eq_2 : !! {T : Type}{l : (list T)}
  {h : T}{t : (list T)}
  (EQ (append (cons h t) l) (cons h (append t l)))) )

( list_append_is_associative = ? : {T : Type}
  {l1,l2,l3 : (list T)}
  (EQ (append l1 (append l2 l3)) (append (append l1 l2) l3)) )

-----
?4827 : { T : Type }
  { l1 : ( list T ) }{ l2 : ( list T ) }{ l3 : ( list T ) }
  ( EQ ( append l1 ( append l2 l3 ) )
    ( append ( append l1 l2 ) l3 ) )
1 new goal

( Inst ([T : ?] ?) )

{ T : Type }
-----
?4955 : { l1 : ( list T ) }{ l2 : ( list T ) }{ l3 : ( list T ) }
  ( EQ ( append l1 ( append l2 l3 ) )
    ( append ( append l1 l2 ) l3 ) )
1 new goal

( Inst (list_ind ? ([l : ?] ?) ? ?) )

{ T : Type }
-----
?5042 : { h : T } { t : ( list T ) }
  ( ( [ l : ( list T ) ]
    { l2 : ( list T ) } { l3 : ( list T ) }
    ( EQ ( append l ( append l2 l3 ) )
      ( append ( append l l2 ) l3 ) ) ) t ) ->
  ( ( [ l : ( list T ) ]
    { l2 : ( list T ) } { l3 : ( list T ) }
    ( EQ ( append l ( append l2 l3 ) )
      ( append ( append l l2 ) l3 ) ) )
    ( cons h t ) ) )
1 new goal

( Inst ([h : ?][t : ?][H : !! ?] ?) )
all goals solved

```

8.3.4 Coinductive Theorem Proving

Coinductive reasoning allows us to prove equality of coinductively defined objects that cannot be distinguished in terms of their behavior. Our means to compute with coinductive objects are strictly limited to the specified modifiers and observers. We demonstrate the main idea using the behavioral specification of flags given earlier. The first step to prove an equality is to find a behavioral equality, i.e. an equivalence relation that is respected by all observers and modifiers. In the case of the flag example such a behavioral equality can be defined as follows via the observer `up?`, so that the behavioral equality is obviously respected by the observer. The following three theorems correspond to the condition that modifiers respect the behavioral equality, too.

```
( BEQ = [f,f' : flag] (EQ (up? f) (up? f')) )
```

```
( BEQ_is_behavioral_1 = ? : {f,f' : flag}
  (BEQ f f') -> (BEQ (up f) (up f')) )
all goals solved
```

```
( BEQ_is_behavioral_2 = ? : {f,f' : flag}
  (BEQ f f') -> (BEQ (down f) (down f')) )
all goals solved
```

```
( BEQ_is_behavioral_3 = ? : {f,f' : flag}
  (BEQ f f') -> (BEQ (rev f) (rev f')) )
( Inst [f,f' : flag] [H : !! ?] ? )
all goals solved
```

Since the essence of coinductive definitions is that behaviorally equivalent objects are identified, we can assume the following axiom `BEQ_implies_EQ` to prove the subsequent theorem.

```
( BEQ_implies_EQ : {f,f' : flag}
  (BEQ f f') -> (EQ f f') )
```

```
( flag_theorem = ? : {f : flag} (EQ (rev (rev f)) f) )
```

```
( Inst [f : flag] ? )
```

```
{ f : flag }
```

```
-----
?8272 : ( ( EQ ( rev ( rev f ) ) ) ) f )
1 new goal
```

```
( Inst (BEQ_implies_EQ ? ? ?) )
all goals solved
```

As we can see the verification of all theorems is essentially automatic by reduction using the computational equations in the flag specification.

What we have done above is in fact the implicit application of a general coinduction principle, which simply states that behaviorally equivalent objects are identified for any behavioral equivalence. It can be expressed in OCC as follows.

```
( flag_coind :
  {BEQ : flag -> flag -> Prop}
  ({f,f' : flag}
   (BEQ f f') -> (EQ (up? f) (up? f')))) ->
  ({f,f' : flag}
   (BEQ f f') -> (BEQ (up f) (up f')))) ->
  ({f,f' : flag}
   (BEQ f f') -> (BEQ (down f) (down f')))) ->
  ({f,f' : flag}
   (BEQ f f') -> (BEQ (rev f) (rev f')))) ->
  {f,f' : flag} (BEQ f f') -> (EQ f f') )
```

In a similar way, we can set up a coinduction principle for our behavioral specification of polymorphic streams.

```
( stream_coind : {T : Type}
  {BEQ : {T | Type} (stream T) -> (stream T) -> Prop}
  ({s,s' : (stream T)}
   (BEQ s s') -> (EQ (head s) (head s')))) ->
  ({s,s' : (stream T)}
   (BEQ s s') -> (BEQ (tail s) (tail s')))) ->
  {s,s' : (stream T)} (BEQ s s') -> (EQ s s') )
```

It can be used to prove equalities such as the one given by the following theorem.

```
( stream_theorem = ? : {T : Type}{s : (stream T)}
  (EQ (merge (odd s) (even s)) s) )
```

```
-----
?9125 : { T : Type }{ s : ( stream T ) }
      ( EQ ( merge ( odd s ) ( even s ) ) s )
1 new goal
```

```

( Inst ([BEQ = [T | Type][s1,s2 : (stream T)]
        (EQ s1 (merge (odd s2) (even s2)))] ?) )

{ BEQ = [ T | Type ][ s1 : ( stream T ) ][ s2 : ( stream T ) ]
  ( EQ s1 ( merge ( odd s2 ) ( even s2 ) ) ) }
-----
?9201 : { T : Type }{ s : ( stream T ) }
      ( EQ ( merge ( odd s ) ( even s ) ) s )
1 new goal

( Inst ([T : ?][s : ?] ?) )

{ BEQ = [ T | Type ][ s1 : ( stream T ) ][ s2 : ( stream T ) ]
  ( EQ s1 ( merge ( odd s2 ) ( even s2 ) ) ) }
{ T : Type }
{ s : ( stream T ) }
-----
?9292 : ( EQ ( merge ( odd s ) ( odd ( tail s ) ) ) s )
1 new goal

( Inst (stream_coind T BEQ ? ? ? ? ?) )

{ BEQ = [ T | Type ][ s1 : ( stream T ) ][ s2 : ( stream T ) ]
  ( EQ s1 ( merge ( odd s2 ) ( even s2 ) ) ) }
{ T : Type }
{ s : ( stream T ) }
-----
?9363 : { s : ( stream T ) }{ s' : ( stream T ) }
      ( BEQ T s s' ) -> ( BEQ T ( tail s ) ( tail s' ) )

{ BEQ = [ T | Type ][ s1 : ( stream T ) ][ s2 : ( stream T ) ]
  ( EQ s1 ( merge ( odd s2 ) ( even s2 ) ) ) }
{ T : Type }
{ s : ( stream T ) }
-----
?9362 : { s : ( stream T ) }{ s' : ( stream T ) }
      ( BEQ T s s' ) -> ( BEQ T ( head s ) ( head s' ) )
2 new goals

( Inst ([s : ?][s' : ?] ?) )

```

```

{ BEQ = [ T | Type ][ s1 : ( stream T ) ][ s2 : ( stream T ) ]
  ( EQ s1 ( merge ( odd s2 ) ( even s2 ) ) ) }
{ T : Type }
{ s : ( stream T ) }
{ s : ( stream T ) }
{ s' : ( stream T ) }
-----
?9772 : ( BEQ T s s' ) -> ( EQ ( head s ) ( head s' ) )
1 new goal

( Inst ([H : !! ?] ?) )
1 goal solved (no new goals)

( Inst ([s : ?][s' : ?] ?) )

{ BEQ = [ T : Type ][ s1 : ( stream T ) ][ s2 : ( stream T ) ]
  ( EQ s1 ( merge ( odd s2 ) ( even s2 ) ) ) }
{ T : Type }
{ s : ( stream T ) }
{ s : ( stream T ) }
{ s' : ( stream T ) }
-----
?10319 : ( BEQ T s s' ) -> ( EQ ( tail s ) ( tail s' ) )
1 new goal

( Inst ([H : !! ?] ?) )
all goals solved

```

Again we can see that the main part of the proof is done automatically by reduction using the computational equations of the behavioral specification.

8.3.5 Theorem Proving Modulo

Theorem proving modulo [DHK98] refers to the technique of proving theorems w.r.t. an underlying equality on propositions, i.e. at a higher-level of abstraction that ignores distinctions that are subsumed by the equality. In fact, we have already used theorem proving modulo an equational theory given by computational equations, namely in the inductive and coinductive proofs of the previous sections, and we have seen that a considerable degree of partial automation can be achieved in this way. In this section we illustrate theorem proving modulo in a more general form, where the underlying equational theory is given by a combination of structural and computational equations.

The capabilities of OCC to support this theorem proving style can best be demonstrated by means of a typical example, namely the verification of linear place invariants in colored net specifications. To this end we apply some concepts formally developed in Chapter 4, namely the notion of typed state space and invariant, to the case study of Chapter 3, which in Section 8.2.8 has been specified using OCC. Hence, this section simultaneously shows how OCC can be employed to reason about specifications in the embedded rewriting logic.

Our starting point is the OCC specification from Section 8.2.8, which is essentially the rewriting semantics of the echo algorithm from Chapter 3. Our first goal is to slightly modify this specification to obtain an equivalent, but suitably explicit and abstract version, which is a more convenient basis for interactive theorem proving.

Recall that the specification introduces a type `act` of actions, a type `st` of states, and the transition predicate `trans`. Since we are now interested in reasoning about the models of this specification, we need to partially make explicit the imposed freeness constraints inside the logic of OCC. Obviously, the freeness constraint for `act` justifies adding the following induction principle.

```
( act_ind : {P : act -> Prop}
  ({x : id}(P (ISEND x))) ->
  ({x : id}{m : st}(P (IRECEIVE x m))) ->
  ({x,y : id}(P (SEND x y))) ->
  ({x,y : id}{m : st}(P (RECEIVE x y m))) ->
  {e : act}(P e) )
```

Aiming at a simple and generic inductive specification of our transition relation `trans`, we introduce an inductive predicate `guard` to represent the guards of net transitions, as well as functions `pre` and `post` to denote their pre- and postmarkings.

```
( guard : act -> Prop )

( guard_ISEND : !! {x : id}
  (EQ (guard (ISEND x)) True) )

( guard_IRECEIVE : !! {x : id}{m : st}
  (EQ (guard (IRECEIVE x m))
    (EQ m (m_MESSAGES (incoming network x)))) )

( guard_SEND : !! {x,y : id}
  (EQ (guard (SEND x y)) True) )
```

```

( guard_RECEIVE : !! {x,y : id}{m : st}
  (EQ (guard (RECEIVE x y m))
    (EQ m (m_MESSAGES (fms_rm (incoming network x) (pair x y)))))) )

( pre : act -> st )
( post : act -> st )

( pre_ISEND : !! {x : id}
  (EQ (pre (ISEND x)) (m_single_QUIET x)) )

( post_ISEND : !! {x : id}
  (EQ (post (ISEND x))
    (m_union (m_single_WAITING x)
      (m_MESSAGES (outgoing network x)))) )

( pre_IRECEIVE : !! {x : id} {m : st}
  (EQ (pre (IRECEIVE x m))
    (m_union (m_single_WAITING x) m)) )

( post_IRECEIVE : !! {x : id} {m : st}
  (EQ (post (IRECEIVE x m))
    (m_single_TERMINATED x)) )

( pre_SEND : !! {x,y : id}
  (EQ (pre (SEND x y))
    (m_union (m_single_UNINFORMED x)
      (m_single_MESSAGES x y)))) )

( post_SEND : !! {x,y : id}
  (EQ (post (SEND x y))
    (m_union (m_single_PENDING x y)
      (m_MESSAGES (fms_rm (outgoing network x)
        (pair y x)))))) )

( pre_RECEIVE : !! {x,y : id}{m : st}
  (EQ (pre (RECEIVE x y m))
    (m_union (m_single_PENDING x y) m)) )

( post_RECEIVE : !! {x,y : id}{m : st}
  (EQ (post (RECEIVE x y m))
    (m_union (m_single_ACCEPTED x)
      (m_single_MESSAGES y x)))) )

```

Now we are prepared to give a concise inductive specification of the transition relation `trans`. It is generated by the atomic transitions as expressed by `t_rule` and by the context closure rule `t_union`. The reader can easily verify that the transition relation `trans` is equivalent to the rewrite predicate `trans` of the former specification.²⁶ We have only introduced the notation `pre/post` for left/right hand side of transition rules and `guard` for rule conditions to enhance readability of proofs. Again, to allow inductive reasoning we add a corresponding induction principle as justified by the freeness constraint for `trans`.

```
( trans : act -> st -> st -> Prop )

( t_rule : {e : act}
  (guard e) -> (trans e (pre e) (post e)) )

( t_union : {e : act}{m,m',m'' : st}
  (trans e m' m'') -> (trans e (m_union m m') (m_union m m'')) )

( trans_ind : {e : act}{P : st -> st -> Prop}
  ((guard e) -> (P (pre e) (post e))) ->
  ({m,m',m'' : st}
   (P m' m'') -> (P (m_union m m') (m_union m m'')))) ->
  {m',m'' : st}(trans e m' m'') -> (P m' m'') )

( trans_ind' : {e : act}{P : st -> st -> Prop}
  ({m : st}(guard e) ->
   (P (m_union m (pre e)) (m_union m (post e)))) ->
  {m',m'' : st}(trans e m' m'') -> (P m' m'') )
```

The last axiom `trans_ind'` is an optimized induction principle that can be proved using `trans_ind`. It is more convenient, since it has fewer premises. We have omitted the proof for the sake of brevity.

Since we wish to reason about the echo algorithm independently of the concrete network topology, we only need a number of minimal assumptions about the initial marking, which are formulated below. The former executable specification with a concrete network topology is a particular instance of this more abstract specification.

```
( initiator : id )
( uninformed_agents : (fms id) )
```

²⁶There is an inessential change in the order of arguments for `trans`, which enables us later to reuse some notions from Chapter 4.

```

( uninformed_agents_not_empty :
  ?? (Not (fms_is_empty uninformed_agents)) )

( initial_marking =
  (m_union (m_single_QUIET initiator)
    (m_place | pl | pl_color UNINFORMED uninformed_agents)) )

( initial_condition = [m : st] (EQ m initial_marking) )

```

On top of this specification of our system model we add the OCC translations of the modules `sets_pred` and `safety` from the temporal library of Chapter 4. Since the translations are straightforward, we omit them at this point and we refer to Chapter 4 for the definitions of `view`, `hoare`, `stable` and `ind_invariant`, which are used in the following.

As a means to prove assertions of the form $(\text{hoare } e \ p \ q)$ we use the following reasoning principle, which is a direct consequence of the induction principle `trans_ind'`.

```

( pn_rule = ? : {p,q : s_prop}{e : act}
  ({m : st}(guard e) ->
    (p (m_union m (pre e))) -> (q (m_union m (post e))))->
  (hoare e p q) )

( Inst ([p : st -> Prop] [q : st -> Prop] ?) )
( Inst ([e : ?] [H : ?] ?) )
( Inst (trans_ind' e ([m' : st] [m'' : st] ((p m') -> (q m'')))) ?) )
( Inst H )
all goals solved

```

The linear invariant we wish to prove is given by the following predicate. It states that, by taking the multiset union of the contents of the places `QUIET`, `WAITING`, and `TERMINATED`, we obtain a singleton multiset which contains only the initiator.

```

( inv = [m : st] (EQ (fms_single initiator)
  (fms_union (m_proj m QUIET)
    (fms_union (m_proj m WAITING)
      (m_proj m TERMINATED)))) )

```

We begin with the main part, namely the proof of stability. Since there is no need for partitioning the system into components, we use the trivial full view, which contains all actions. The definition of stability is unfolded to a universal quantification over all actions, which is proved by induction using `act_ind`.

```

( full_view = [e : act] True )

( inv_stable = ? : (stable full_view inv) )

-----
?23493 : { e : act } ( set_in ( [ x : act ] True ) e ) ->
  ( hoare e
    ( [ m : st ] ( EQ ( fms_single initiator )
      ( fms_union ( fms_union ( m_proj m QUIET )
        ( m_proj m WAITING ) )
        ( m_proj m TERMINATED ) ) ) ) )
    ( [ m : st ] ( EQ ( fms_single initiator )
      ( fms_union ( fms_union ( m_proj m QUIET )
        ( m_proj m WAITING ) )
        ( m_proj m TERMINATED ) ) ) ) ) ) )

( Inst (act_ind ? ? ? ? ?) )

-----
?28894 : { x : id }{ y : id }{ m : st }
  ( ( [ e : act ]
    ( set_in ( [ x : act ] True ) e ) ->
    ( ( hoare e
      ( [ m : st ] ( EQ ( fms_single initiator )
        ( fms_union ( m_proj m QUIET )
        ( fms_union ( m_proj m WAITING )
          ( m_proj m TERMINATED ) ) ) ) ) )
      ( [ m : st ] ( EQ ( fms_single initiator )
        ( fms_union ( m_proj m QUIET )
        ( fms_union ( m_proj m WAITING )
          ( m_proj m TERMINATED ) ) ) ) ) ) ) )
    ( RECEIVE x y m ) )

-----
?28893 : { x : id }{ y : id }
  ( ( [ e : act ]
    ( set_in ( [ x : act ] True ) e ) ->
    ( ( hoare e
      ( [ m : st ] ( EQ ( fms_single initiator )
        ( fms_union ( m_proj m QUIET )
        ( fms_union ( m_proj m WAITING )
          ( m_proj m TERMINATED ) ) ) ) ) )
      ( [ m : st ] ( EQ ( fms_single initiator )

```

```

      ( fms_union ( m_proj m QUIET )
        ( fms_union ( m_proj m WAITING )
          ( m_proj m TERMINATED ) ) ) ) ) ) )
    ( SEND x y ) )

```

```

-----
?28892 : { x : id }{ m : st }
  ( ( [ e : act ]
    ( set_in ( [ x : act ] True ) e ) ->
    ( ( hoare e
      ( [ m : st ] ( EQ ( fms_single initiator )
        ( fms_union ( m_proj m QUIET )
          ( fms_union ( m_proj m WAITING )
            ( m_proj m TERMINATED ) ) ) ) ) ) )
      ( [ m : st ] ( EQ ( fms_single initiator )
        ( fms_union ( m_proj m QUIET )
          ( fms_union ( m_proj m WAITING )
            ( m_proj m TERMINATED ) ) ) ) ) ) ) ) )
    ( IRECEIVE x m ) )

```

```

-----
?28891 : { x : id }
  ( ( [ e : act ]
    ( set_in ( [ x : act ] True ) e ) ->
    ( ( hoare e
      ( [ m : st ] ( EQ ( fms_single initiator )
        ( fms_union ( m_proj m QUIET )
          ( fms_union ( m_proj m WAITING )
            ( m_proj m TERMINATED ) ) ) ) ) ) )
      ( [ m : st ] ( EQ ( fms_single initiator )
        ( fms_union ( m_proj m QUIET )
          ( fms_union ( m_proj m WAITING )
            ( m_proj m TERMINATED ) ) ) ) ) ) ) )
    ( ISEND x ) )

```

4 new goals

The four resulting subgoals are similar, which is why below we have given only the most interesting case of IRECEIVE. As we can see the proof is practically automatic, thanks to the fact that we are reasoning *modulo* the structural and computational equations of finite multisets and markings (which especially include the place linearity equations).

--- Case ISEND ---

...

1 goal solved (no new goals)

--- Case IRECEIVE ---

(Inst ([x : id][mrk' : st][D : ?] ?))

{ x : id }

{ mrk' : st }

{ D : (set_in ([x : act] True) (IRECEIVE x mrk')) }

?73262 : { s : st } { s' : st }

((trans (IRECEIVE x mrk') s s') ->

((([m : st] (EQ (fms_single initiator)

(fms_union (fms_union (m_proj m QUIET)

(m_proj m WAITING))

(m_proj m TERMINATED)))) s) ->

(([m : st] (EQ (fms_single initiator)

(fms_union (fms_union (m_proj m QUIET)

(m_proj m WAITING))

(m_proj m TERMINATED)))) s')))

1 new goal

(Inst (pn_rule (IRECEIVE x mrk') ?))

{ x : id }

{ mrk' : st }

{ D : (set_in ([x : act] True) (IRECEIVE x mrk')) }

?78669 : { m : st } (guard (IRECEIVE x mrk')) ->

((([m : st] (EQ (fms_single initiator)

(fms_union (fms_union (m_proj m QUIET)

(m_proj m WAITING))

(m_proj m TERMINATED))))

(m_union m (pre (IRECEIVE x mrk')))) ->

(([m : st] (EQ (fms_single initiator)

(fms_union (fms_union (m_proj m QUIET)

(m_proj m WAITING))

(m_proj m TERMINATED))))

(m_union m (post (IRECEIVE x mrk')))))

1 new goal

```
(Inst ([mrk : st] [G : !! ?] [H : !! ?] ?))
```

```
1 goal solved (no new goals)
```

```
--- Case SEND ---
```

```
...
```

```
1 goal solved (no new goals)
```

```
--- Case RECEIVE ---
```

```
...
```

```
all goals solved
```

Given the proof of stability it remains to check the invariant property `inv` under the initial condition, which is expressed by one of the subgoal ?124607 below.

```
( inv_invariant = ? : (ind_invariant full_view
                        initial_condition inv) )
```

```
...
```

```
( Inst (And_intro ? ? ? ?) )
```

```
-----
```

```
?124608 : { e : act } ( set_in ( [ x : act ] True ) e ) ->
  ( ( ( hoare e )
    ( [ m : st ] ( EQ ( fms_single initiator )
      ( fms_union ( fms_union ( m_proj m QUIET )
        ( m_proj m WAITING )
        ( m_proj m TERMINATED ) ) ) ) ) )
    ( [ m : st ] ( EQ ( fms_single initiator )
      ( fms_union ( fms_union ( m_proj m QUIET )
        ( m_proj m WAITING )
        ( m_proj m TERMINATED ) ) ) ) ) ) )
```

```
-----
```

```
?124607 : { s : st }
  ( ( [ s : st ]
    ( ( ( [ m : st ] ( EQ m
      ( m_union
        ( m_place QUIET ( fms_single initiator ) )
        ( m_place UNINFORMED uninformed_agents ) ) ) ) ) s ) ->
```

```

      ( ( [ m : st ] ( EQ ( fms_single initiator )
        ( fms_union ( fms_union
          ( m_proj m QUIET )
          ( m_proj m WAITING ) )
          ( m_proj m TERMINATED ) ) ) ) s ) ) )
s )
2 new goals

( Inst ([s : st][H : !! ?] ?) )
1 goals solved (no new goals)
( Inst inv_stable )
all goals solved

```

In summary we can say that the idea of theorem proving modulo leads to considerable improvements regarding partial automation of proofs, even in the form that we employ in OCC, which is not limited to the first-order case, but on the other hand based on reduction modulo structural equations rather than on more powerful narrowing, which is used in [DHK98]. Moreover, we believe that reasoning about concurrent systems represented in a natural multiset-rewriting style is an important application, that we have only touched upon here in a very specific setting, but seems to have much more potential in the verification of concurrent systems that is worth to investigate in the future.

8.4 Final Remarks

We have explored a combination of CC with a flexible ECC-style universe hierarchy and a computational system based on conditional rewriting modulo equations as it can be found in membership equational logic or, more generally, in rewriting logic. The resulting system, that we refer to as OCC, for the open calculus of constructions, is more expressive than ECC, because typechecking can take into account logical knowledge, which could be given by axioms or theorems that are available in a given context. The resulting system is also more expressive than membership equational logic or rewriting logic, since instead of kinds we have expressive higher-order types, and instead of a fragment of first-order logic we can utilize the higher-order logic provided by the propositions-as-types interpretation. The classical set-theoretic semantics that we have developed in this chapter is given independently of the formal system, which implies that logically OCC specifications can be understood independently and in particular without reference to the operational semantics. The semantics is in our view very intuitive, and it shows that the use of dependent types as a primitive concept, rather than having a logic together with a more or less restricted type system, is an absolutely satisfactory way to deal with classical mathematics.

As it is often the case with an increase in expressiveness, there is a price to be paid. The computational system of CC is context-independent and enjoys good operational properties such as confluence and strong normalization. The drawback is that only a restricted class of functional programs can be executed, and executable specifications cannot be expressed. In OCC, however, the computational system is context-dependent, and we have not imposed any restrictions, except that it should be based on a form of conditional equational rewriting and goal-oriented search that is covered by the rules of the formal system. As a consequence it is possible to write OCC specifications that do not enjoy general confluence or normalization properties, with corresponding consequences for type inference and type checking, and, just as in membership equational logic and in rewriting logic, it is the responsibility of the user to write computationally useful specifications, i.e. specification with good computational properties for the relevant application scenarios. Since type inference and type checking is based on computation, it is furthermore in the hands of the user to specify a computational system which is sufficiently complete for the purposes of the application.

In addition to its relationship to CC, it is instructive to compare our approach with Martin-Löf's type theory or better with Nuprl, which takes Martin-Löf's original ideas even further. The main difference is that both Nuprl and the related version of Martin-Löf's type theory are polymorphic, in the sense that the underlying language is untyped and typically semantically different types can be assigned to the same term. This is in contrast to OCC, which is a monomorphic type theory in the sense that terms are equipped with explicit type information and well-typed terms are equipped with a semantically unique type. A difference related to polymorphism is that functions are set-theoretic in OCC, but in Martin-Löf's polymorphic type theory and in Nuprl they are characterized by their computational behavior (this observational approach is especially visible in Nuprl's contravariant subtyping rules). We have already pointed out that Martin-Löf has also considered monomorphic type theories, and that Martin-Löf's types can be specified to reside in the predicative universes of OCC. Especially noteworthy is Martin-Löf's logical framework [PSN90], which is similar to the intended use of OCC, but neither Martin-Löf's type theories nor Nuprl provide a means to extend or specify the computational system inside the theory. Another point is that all universes in Martin-Löf's type theory and Nuprl are strictly predicative, whereas OCC admits impredicative universes. If the distinguished OCC universe `Prop` is designated as an impredicative universe, then it contains an impredicative higher-order logic by virtue of the propositions-as-types interpretation, which can be directly used to express all common concepts of classical mathematics. On the other hand, the use of impredicative universes with the intention to obtain a classical impredicative higher-order logic is more a matter of convenience rather than a necessity as the HOL/Nuprl connection clearly demonstrates. Therefore, we have explicitly admitted instances of OCC without impredicative universes in

our treatment. Since we have furthermore not excluded the possibility that `Prop` is neither predicative nor impredicative, we can still enjoy the convenience of a classical propositional universe.

`Nuprl` is an open-ended type theory with a rich collection of types, whereas `OCC` is rather minimalistic, in the sense that it is based on the single concept of dependent function types. The openness of `OCC` should be clearly distinguished from the openendedness of `Nuprl`, which refers to the possibility to add new type constructors and corresponding inference rules [Tsu01]. This is possible, because the semantics of `Nuprl` is given for an underlying untyped programming language in a generic way, and hence can be used to justify the introduction of new type constructors to classify elements that are already semantically meaningful. In `OCC` openness refers to openness of the computational system, which can be freely specified by the user within the bounds provided by the logic, but without extending the formal system itself. The addition of entirely new type constructors would require an extension of the underlying language, most likely requiring a considerable modification of the associated semantics, and hence seems to be more difficult than in the generic approach of `Nuprl`. On the flip side, the semantics of `OCC` is a classical set-theoretic one, which is intuitively simpler than the constructive semantics of `Nuprl` [All87].²⁷ The reason is that `OCC` has only a few primitive concepts to justify, but especially because equality in `OCC` is implicitly polymorphic and is interpreted uniformly as set-theoretic equality, a point that is important to use equations as computation rules in arbitrary contexts, whereas in `Nuprl` equality is explicitly polymorphic and its semantics is type-dependent in an essential way.²⁸

Indeed, our rationale was to keep `OCC` as simple as possible with the intention that, rather than to anticipate possible extensions of the formal system, it could be used more as a framework to internally develop new concepts such as contravariant subtyping, quotients, inductive and coinductive types, parameterization, etc. inside the type theory. As indicated by some of our examples, a systematic approach for such internal extensions could be based on a development of computational category theory inside `OCC`. Indeed, we think of `OCC` as a type theory that provides a minimal starting point for such an endeavor, which could be conducted in a similar way as the development [HS98] conducted in `COQ` by Huet and Saïbi, but it could benefit from the particular computational features of `OCC` as we have seen from some of the examples discussed in this chapter. Also the elegant representation of certain computational concepts

²⁷It is interesting to note that one of the motivations of [How97] was to provide a simpler semantics for a variant of `Nuprl`, which eliminates the need for partial equivalence relations, but our impression is that the resulting semantics would be still difficult to understand for a nonexpert.

²⁸As shown in [How98b], it is still possible to support efficient equational rewriting in `Nuprl` at the tactic level, if terms are equipped with certain type annotations.

of category theory [RB88] using a typed functional programming language such as ML could be carried out in OCC and could benefit from the use of dependent types and computational equations.

In spite of several differences, there is an interesting feature of Nuprl that would be very desirable in the context of OCC, namely direct computation rules [CAB⁺86], which allow untyped computation, and have already been admitted in the semantics of Martin-Löf's type theory (cf. [Tsu01]). By using an untyped notion of equality, OCC already supports a limited form of untyped computation, which can be semantically justified. On the other hand, the rule for β -reduction **RedBeta** in OCC, and all the other rules **Str**, **As**, **Red**, and **Rew** have dynamic type checking premises. Using, however, the simplified set-theoretic semantics presented at the end of Section 8.1.2 the omission of the premise of **RedBeta** can be easily justified for purely predivisive instances of OCC. This idea could be generalized to our equational/rewriting setting (i.e. to the rules **Str**, **As**, **Red**, and **Rew**) to cover computation without generating dynamic type checking conditions, provided that the types of quantified variables are suitably general and certain mild variable restrictions are satisfied, similar to those imposed for executability of membership equational and rewriting logic specifications in Chapter 2. Unfortunately, such direct computation rules cannot be justified by our classical set-theoretic semantics for the general case of OCC (for instance, our soundness proof of **RedBeta** makes essential use of the typing premise). Whether a rich hybrid set-theoretic/computational semantics in the spirit of [HS94, How97] (in these references dynamic type checking is performed at the semantic level) could be given to impredicative universes remains unclear, however, because his interpretation of types is restricted to purely set-theoretic objects.²⁹ On the other hand, it should also be mentioned that there are also some arguments in favor of dynamic type checking (at the level of the formal system rather than at the semantic level). First of all, the variable restriction mentioned above excludes examples (such as the operational extensionality axiom of Section 8.2.6), where variables could be instantiated by means of type inference. It is furthermore too strong for computational rewrite axioms if quantified variables are not instantiated by matching but by more general rewrite strategies, where arbitrary terms can be supplied by the metalevel and dynamic typechecking is needed to check their well-typedness. Second, dynamic type checking would be an important feature in an extension of OCC by a richer notion of (covariant) subtyping that is not confined to universe subtyping, because the applicability of assertional axioms, computational equations, or computational rewrite axioms could depend

²⁹An interesting question in this context is if disregarding computational aspects an instance of OCC with a standard predicative and cumulative universe hierarchy could be equipped with an abstract logical semantics via an embedding into Nuprl, but it seems that the answer is negative, already because of the inherently untyped nature of OCC's equality judgements, which is in contrast to Nuprl's type-dependent equality. A similar difficulty arises if we want to relate OCC and Martin-Löf's type theory (extended by fully cumulative universe subtyping).

on types in an essential way.

Type checking in Martin-Löf's polymorphic, extensional type theory and in Nuprl is a matter of interactive theorem proving and in the case of the Nuprl system is supported by tactics external to the type theory. A very powerful characteristic of both type theories is that there are virtually no restrictions on how logical equalities can be used to prove typing judgements. In other words, type equality is logical rather than computational. Instead, OCC preserves the, in our view essential, feature of CC and other PTSs (a feature that can already be found in Martin-Löf's decidable type theory [ML74]), namely that type checking should be based on the operational semantics of the type theory itself. In other words, it is based on a notion of computation, which in the case of OCC is the general notation of conditional rewriting modulo equations. Since any logical equality can be designated as operational (in different ways) we do not lose expressiveness, but type checking becomes a process that is conducted using the same operational semantics of the type theory that is used to execute programs and specifications. Since proof checking is a particular case of type checking, it follows that also proof checking in OCC is based on its own operational semantics. This is different from Nuprl, where a sufficiently rich notion of proof and the associated process of proof checking are explained using tactics in the metalanguage and hence are external to the formal system.

In summary, we consider OCC as a promising research direction in the integration of concepts from equational logic, rewriting logic, and different type theories. Regarding its expressiveness, our approach subsumes membership equational logic, the corresponding version of rewriting logic, the calculus of constructions, and a monomorphic Martin-Löf-style type theory. On the other hand, we have seen that OCC is very different from existing logics and type theories in terms of its semantics and its formal system. In essence, OCC is based on the interaction between just two key concepts, which are dependent types and an operational semantics based on conditional rewriting modulo equations. As suggested by our examples, OCC could be used as a core formalism for the development of a unified language for programming, specification, and interactive theorem proving, but we feel that the contribution of this chapter should be regarded as the beginning rather than the end of such an endeavour. Hence, we conclude this section by pointing out a number of issues that we think would be important to continue our line of research.

On the theoretical side, we have indicated that, due to its generality, a meta-theoretic study of operational properties of OCC is a difficult enterprise and is left as an issue for future work. Apart from a better understanding of the computational system of OCC, such a study could be valuable from an implementation viewpoint. Especially important seems to be the investigation of useful sufficient conditions under which dynamic type checking can be eliminated or optimized,

since this would impact the efficiency of possible implementations considerably.³⁰ Also on the model-theoretic side, further work is worthwhile. The study of alternative semantic models for OCC, would independently contribute to a better understanding from a different perspective and could furthermore justify future extensions of the formal system. We have already mentioned that concepts such as implicitly dependent types and direct computation rules would be natural extensions if they could be justified by semantic means, and that, furthermore, a study of the predicative instances of OCC would be of independent interest. Also the integration of universe polymorphism, e.g. by means of type universes, would be an important issue that needs a careful semantic justification. In addition, the definition of a constructive set-theoretic semantics, e.g. models based on ω -sets similar to those in [Luo94], models based on combinatory algebras similar to [SG96], or even less abstract term models, should be possible, because the formal system of OCC is not classically biased. Still the constructions known from CC or ECC cannot be directly used, because models of OCC should have a uniform interpretation for the syntax and should especially not rely on metatheoretic properties of the formal system, which seems to indicate that a higher degree of intensionality is necessary in the interpretation. Since OCC allows different hierarchies of universes and the universes can be semantically separated to a certain degree, it seems also possible to construct hybrid models, for instance, combining a domain-theoretic semantics and a classical set-theoretic one (e.g. to reason about domains in a classical logic), another direction that might be interesting to explore.³¹ Last but not least, it would be worth investigating if suitable instances of OCC could be interpreted in untyped languages such as those proposed in [Fef75], [Bee88], and [Gru92], which combine computation and set-theoretic concepts.³²

On the practical side, we think that more experience is needed with the use of OCC. By means of a number of examples we have touched upon many possible applications. A consequent next step would be to conduct larger case studies, again in different application domains ranging from the formalization of mathematical concepts to computer science applications such as verification of concurrent systems and metatheoretic reasoning about formalisms. The feedback from such case studies would constitute a valuable input for the future development and possible extensions. We have already pointed out a few extensions of the formal system that would still fit well into the minimalistic approach of OCC. To more systematically support other concepts, such as contravariant subtyping, algebras

³⁰Since a similar form of dynamic type checking is needed in Nuprl (although at the tactic level), a type annotation scheme has been developed in [How98b] to increase the efficiency of, for instance, tactics for equational rewriting.

³¹Since every type would be inhabited in domain-theoretic universes, we cannot use them for logical purposes, however.

³²It should be mentioned that [Gru92] is based entirely on a notion of a map and the relation to set theory is less explicit.

and modules as objects, inductive/coinductive definitions, and parameterization, our examples suggest that a development of computational category theory, which would itself be a larger case study in formalized mathematics, could provide a basis to develop such concepts internally. In addition, there are a number of conceivable metalevel extensions, often referred to as syntactic sugar, because they do not impact the underlying formal system, that could considerably improve the usability of OCC, schemes for certain forms of (co)inductive definitions and implicit coercions as they are available in the COQ proof assistant being particular examples. Yet another issue of practical relevance in the context of larger applications is support for formal interoperability, especially with classical logic theorem provers such as HOL and PVS. Since OCC is equipped with a classical set-theoretic semantics and an open computational system, we do not expect major theoretical difficulties regarding this issue.

A different but important aspect that we have not treated in this chapter is the specification of rewrite strategies, for which several different approaches are conceivable, one of them being the generalization of the reflective approach of Maude [CM96], which would amount to a reflection of the entire language of OCC and would allow us to specify strategies themselves in OCC at the metalevel. Another possibility, which is less flexible but does not rely on reflection, would be to complement OCC specifications with strategy specifications in a fixed strategy language similar to the approach of ELAN [BKK⁺98, BCD⁺98]. As indicated by the reflective architecture of the OCC prototype itself, reflection has many other applications and hence constitutes a feature of independent interest, which can be realized in a variety of flavors and to various degrees. For instance, instead of a full reflection of the entire language of OCC, one might consider partial reflection of the first-order computational equational/rewriting logic sublanguage (e.g. similar to Maude) or partial reflection of a higher-order functional sublanguage (e.g. similar to applicative LISP). Of practical interest would be the question how the strong typing capabilities of OCC can be used to ensure that the representations are themselves strongly typed. Whether a practically useful strongly typed reflection of the full language of OCC in the sense of [LO93] can be obtained is another interesting question that remains beyond the scope of this thesis.

8.5 Acknowledgements

An important motivation for this chapter is derived from my occupation with higher-order logics and type theories, such as those implemented in the proof assistants HOL [GM93b], IMPS [FGT93], PVS [ORS92], LEGO [Pol94], COQ [BBC⁺99], and Nuprl [CAB⁺86], and their practical use to formalize concepts in mathematics and computer science. This has happened in the context of my lectures on type theory at the University of Hamburg in 1997, in subsequent projects

with students, and furthermore in the scope of the European Community project MATCH (CHRX-CT94-0452), which was also the context for the COQ development presented in Chapter 4. Since all this took place during my appointment as a research assistant at the University of Hamburg, I would first of all like to thank Rüdiger Valk for giving me the opportunity to pursue my interests and providing support in many conceivable ways. A particular influencing experience for me was the attendance of the Marktoberdorf Summer School on Logic and Computation in 1997, where I among several lectures on type theory learned about rewriting logic from José Meseguer. Not surprisingly, another major source of inspiration for this chapter is derived from my subsequent occupation with equational logic and rewriting logic, which mainly took place during several fellowships in the rewriting group at SRI International between 1998 and 2001. I am indebted to José Meseguer, who not only invited me to come to SRI, but also in numerous discussions influenced the work contained in the present chapter in a very positive and constructive way. My first visit at SRI which specifically contributed to the research presented in this chapter was sponsored by a DAAD grant in the scope of HSP-III, which is hereby gratefully acknowledged. The environment at SRI was a perfect setting for the subject of this chapter, especially the presentation of early ideas on OCC during the SRI logic lunch and many subsequent discussions with members of the PVS group, especially Pat Lincoln, Sam Owre, Harald Reuss, John Rusby, and Natarajan Shankar, have been a valuable experience. During the SRI fellowship I also had the opportunity to visit Robert Constable and his group at Cornell from whom I learned much about the philosophy behind Nuprl. I appreciate his invitation for a presentation at the Nuprl seminar in 1999, and I well remember very stimulating discussions with Stuart Allen about Nuprl and its semantics and with other members of the Nuprl group including Mark Bickford, Christoph Kreitz, and Pavel Naumov. Furthermore, I especially remember a very helpful discussion with Doug Howe at Lucent/Bell Labs about the HOL/Nuprl connection the ideas behind his hybrid set-theoretic/computational semantics, and I am very grateful for his invitation. Also the presentation of some preliminary ideas on OCC concerned with formal interoperability at the DARPA formal methods meeting in 1999 and the feedback of the other participants have been very encouraging. I furthermore appreciate a discussion with Claude Kirchner on theorem proving modulo pointing me to his work on the subject, and remember a very informative meeting with Jean-Pierre Jouannaud, which was interesting for me, because OCC tries to realize his early idea of a combination of the calculus of constructions and membership equational logic. Moreover, I benefitted from several enjoyable work meetings with Helmut Schwichtenberg on type theory and inductive definitions, when he was on sabbatical at Stanford. I also enjoyed talking with Randy Pollack at TPHOLs'2001 in Edinburgh about the difficulties of programming with dependent types in present type theories, and I would like to thank him for his examples. From the members of the Maude team, I would like to send special thanks to Steven Eker, because he was always willing to help

when I suggested special features for a more efficient support of the OCC prototype, which became one of the larger applications of Maude. Back in Hamburg, I had a very fruitful discussion with Thorsten Altenkirch when he was visiting our department. We especially discussed semantic aspects of OCC, and I would like to thank him for his knowledgeable comments. Last but not least, I would like to thank Carolyn Talcott and José Meseguer for their extensive feedback and for many detailed suggestions which improved the present chapter considerably.

Chapter 9

Conclusions and Future Work

This thesis is an attempt to contribute to the long-term goal of a unified language for programming, specification, and interactive theorem proving. It reflects our view that such an enterprise should be first and foremost based on the practical experience gained in a preferably wide range of application domains. Instead of developing an entirely new approach from scratch we have selected two state-of-the-art approaches as starting points, for which a substantial body of theoretical and practical experience is already available.

First of all, we have chosen *rewriting logic* (RWL), more precisely a version of rewriting logic that is based on a *membership equational sublogic* (MEL). Rewriting logic is the foundation of the Maude language and generalizes the traditional equational programming and specification paradigm to address the executable specification of concurrent and distributed systems. We found the approach of rewriting logic very attractive, because it is not tailored to specific applications and actually can be used as a semantic framework for a wide range of approaches to programming and system specification, ranging from simple automata to potentially complex concurrent object systems. Apart from its use as a semantic framework for programming and specification languages, another aspect of rewriting logic is that of its logical framework applications, in which formal systems can be explained using a simple first-order approach with typical benefits such as executability of the specified formalisms.

The second starting point is *type theory*, more precisely the *calculus of constructions* (CC) extended by a universe hierarchy in the style of Martin-Löf's type theory. This is a logical type theory in the broader class of pure type systems, which is the core of the type theories used in the higher-order logic proof assistants LEGO and COQ. The attractive feature of logical type theories is the unified view of programming and interactive higher-order logic theorem proving, which is made possible by the dependent function types available in these theories, being a consequent generalization of the standard function types available in typed functional programming languages. In other words, instead of having to deal with two primitive concepts, a logic and an underlying type theory as in higher-order theorem provers like HOL, we have dependent types as the only primitive concept, since higher-order logic can be defined in the type theory by means of a propositions-as-types interpretation. This leads not only to a certain economy that is beneficial for theory and implementation, but it makes evident that typed functional programming and proof development are the same task and do not need to be artificially separated.

Without ignoring the important impact of mathematical elegance and inspiration, we believe that theory should ultimately be driven by applications. In the following we focus on how the design of the *open calculus of constructions* (OCC), that is our approach to unify the lines of research mentioned above, was influenced by several applications of rewriting logic and type theory studied in Chapters 3 – 7 of this thesis. To this end, we briefly summarize the specific contributions

of each the these application chapters to the research in its corresponding domain, simultaneously pointing out future work that would be of interest in the respective context. In addition, we take a look at each of these chapters from the viewpoint of their impact on the design and applications of OCC, the calculus which constitutes the main contribution of this thesis to which we come back in the remainder of this chapter.

Chapter 3

Rewriting Logic as a Semantic Framework: Representing High-Level Petri Nets

We begin with Chapter 3, where we studied the use of rewriting logic as a semantic framework for concurrency. To this end, we have treated different classes of Petri nets, which are one of the most popular models for concurrent and distributed systems. Continuing the line of research initiated by Meseguer and Montanari under the motto “Petri nets are monoids”, we have developed a rewriting semantics for a very general class of algebraic net specifications, namely colored net specifications over membership equational logic, and, by establishing functoriality of the semantics and a natural isomorphism with the Best-Devillers process semantics, we have given a semantic justification for our representation. We think that our categorical treatment is quite elegant, because it first develops the relevant concepts for place/transition nets and as a second step generalizes the results to the high-level case. We have also argued in favor of using the collective token philosophy as a mathematical basis for the semantics of concurrent systems, the justification being its elegant generic nature as witnessed by rewriting logic and the fact that more refined semantic models, the individual token semantics being just one example, can always be obtained by explicit enrichment, provided that the formalism is powerful enough to express this enrichment, a condition that we think is usually satisfied in the languages used in practice and especially in colored net specifications.

An immediate practical implication of our results is that we can exploit the executability of rewriting logic to execute Petri net models. Using a case study of a distributed network algorithm we have shown how (high-level) Petri nets can be efficiently executed using a typical rewriting engine such as Maude, which supports rewriting modulo associativity, commutativity and identity laws. We have also shown that the equational/rewriting logic approach suggests an abstract notion of execution that is by no means confined to the initial model, an important point for applications in formal verification using (interactive) theorem proving.

The rewriting logic approach is uniform in the sense that from a logical point of view the gap between the data and the state space representation disappears, and that operationally the same notion of conditional rewriting that is used to represent Petri net transitions is employed at the level of net inscriptions. In

this chapter we have furthermore indicated how a rewriting semantics could be defined for many other Petri net classes, ranging from Petri nets based on other inscription languages to Petri nets with a state space structure that is more complicated than those explicitly discussed in this chapter. The reason for the close connection between rewriting logic and Petri nets is in our view that rewriting logic generalizes the notion of Petri nets without giving up the two key principles of this formalism, which are locality and monotonicity. Apart from its contribution to a conceptual unification in the quite diversified field of Petri nets, the rewriting logic approach suggests interesting new Petri net classes, active token nets being one example that we specifically mentioned. In summary, this chapter opens up an entire research direction that could be explored guided by the unifying role of rewriting logic.

The results of this chapter are of immediate use also in the context of OCC, which can serve in a similar way as a semantic framework, since it generalizes rewriting logic and its underlying membership equational logic. The representation in OCC also answers the question about the representation of colored Petri nets based on typed higher-order functional/equational programming languages, and a question of how to represent colored net specifications based on a language, like higher-order logic, that is more expressive than membership equational logic. Indeed, the main case study of Chapter 3 is revisited in Chapter 8, where we give an equivalent strongly typed representation, which relies on dependent types in an essential way, and hence suggests that OCC is a suitable framework to represent colored net specification based on a strongly typed language, which can be OCC itself or a language that is expressed using OCC as a semantic framework.

Another main issue brought up already in Chapter 3 was how the same formal representation that can be used for net execution, in fact we have already touched upon the more general idea of abstract net execution, could also be used as a basis for interactive theorem proving, i.e. to prove properties about the system represented. In Chapter 8 we have explained how this can indeed be done, based on the strongly typed representation in OCC. Hence, the OCC approach gives further evidence to the thesis of Chapter 3 that the conventional notion of execution, abstract execution, and interactive theorem proving are just different points on a spectrum that can be beneficially integrated. In the future, it would be important to gain more experience with colored net specifications based on OCC and their formal verification in the same formalism. To this end, it would be desirable to exploit the unifying idea of rewriting logic also at the verification level, that is, to develop sufficiently general techniques that could also be used for Petri net extensions or even for more general rewrite specifications.

Finally, we should emphasize once again that Petri nets are a typical representative of an entire class of models based on the multiset-representation of distributed states, and that the potential to use rewriting logic and OCC for other classes of Petri nets and similar models clearly exists. The class of concurrent

object systems used in the rewriting logic approach to concurrent object-oriented programming, for instance, is such a model, which is also based on a multiset-representation and has much in common with our representation of colored Petri nets. In view of the growing interest in object-oriented extensions of Petri Nets it would be worth to further explore this connection.

Chapter 4

Metalogical Reasoning in the Calculus of Inductive Constructions: A Formalized Generalization of UNITY

A formal framework for proving temporal properties about systems has been presented in Chapter 4, where we used the *calculus of inductive constructions* (CIC) and its implementation in the COQ proof assistant to conduct a rigorously formal development of a UNITY-style temporal logic library, which goes beyond the standard UNITY approach in several aspects. Instead of dealing with the execution of system models as in Chapter 3, we exploited here the strengths of a logical type theory with dependent types and its higher-order logic in interactive theorem proving.

On the theoretical side, one aspect of our generalization is the use of labeled transition systems as a common semantic basis, which makes the development applicable not only to UNITY-style programs, but also to Petri nets, rewriting logic specifications, and many other formalisms that can be equipped with a transition system semantics. Another improvement is that we generalize the unconditional notion of UNITY-fairness to a notion of weak group fairness, which allows us to specify weak fairness for groups of transitions, a feature that is needed for instance in the verification of colored net specifications and, more generally, in rewrite specifications, where transitions and rules are conditional and semantically correspond to groups of transitions in the labeled transition system. An interesting point of all our generalizations is that we preserve the core philosophy of the UNITY approach, namely that the temporal logic is inductively defined over the underlying transition system, a feature which makes the UNITY temporal logic unique and very different from standard approaches to temporal logic.

Apart from proving most proof rules known from UNITY in our general setting, we have proved a number of additional rules. These include, for instance, rules to deal with relative assertions, which address the difficulties caused by the use of a strongly typed logic that have already pointed out by Lamport. It is noteworthy that the proof rules for relative assertions make essential use of dependent types. The additional rules also include a new rule for compositional reasoning with LEADS TO assertions, which naturally extends the collection of composition rules for the other operators. The rule is intended for tightly coupled systems and can be regarded as a formalization of the concept of interference-free proof

schemes which goes back to Owicki and Gries. The rule makes essential use of the propositions-as-types interpretation, and its formal metatheoretic proof, which is conducted by induction over temporal logic proofs, becomes quite elegant thanks to the proofs-as-objects interpretation. As far as we know, this use of the propositions-as-types interpretation for compositional reasoning is a new application that has not been considered so far, and it would certainly be interesting to further explore its potential.

An important issue for a temporal logic library, that has not been addressed in this chapter, is the support for rely/guarantee-style compositional reasoning about loosely coupled systems. We have mentioned several approaches that have been studied in the literature, and we pointed out that their integration and formalization in CIC would constitute an interesting and challenging extension of our development. Another topic left for future work is concerned with the extension of our development by a stronger notion of fairness. The question is if, similar to our notion of weak group fairness, a notion of strong group fairness can be added without giving up the elegance and the inductive nature of the original UNITY approach. Yet another issue that has not been addressed in this thesis is the question of semantic completeness of the temporal logic and its formalization in CIC. Further directions for future work could be concerned with the extension of the temporal logic library with more specific support for classes of system models. Interesting examples in our view are colored net specifications, restricted classes of rewrite logic specifications, and, furthermore, Misra's new Seuss methodology, an object-oriented discipline of multiprogramming, which can be regarded as a successor of the UNITY methodology and actually employs the (New) UNITY temporal logic that was the starting point of our generalization.

Technically, our development is based on the use of CIC as a higher-order logical framework and simultaneously as a metalogic, i.e. in this case for inductive reasoning about the temporal logic represented. This approach enabled us to derive all temporal logic proof rules as theorems in the metalogic, and thus provides a very high degree of confidence in the correctness of the library. The capabilities of CIC and its implementation in COQ for inductive reasoning made the formal development a manageable task. Although inductive definitions are not a primitive concept in OCC, the open computational system allows us to view inductive definitions as specifications of a restricted form, so that the development of this chapter could also be conducted in OCC without major modifications. However, as we pointed out in Chapter 8, the use of OCC leads to possible improvements. For instance, the somewhat cumbersome treatment of abstract state spaces in CIC is caused by the restrictive notion of computational equality at the type level, being just one example where OCC allows more natural formulations.

Another more fundamental issue that we learned from this development, and we think is worth pointing out here, is the apparent tradeoff between modularity and conservativity in those type theories where type checking is based on com-

putation. In fact, our impression is that true modularity, which is based on the principle of information hiding by distinguishing between an abstract specification and the concrete implementations of a module, is incompatible with a closed computational system as it is used for instance in CIC. In earlier versions of our development, we have encountered situations where client modules cannot be typed without exhibiting a computationally useful implementation of the modules they are based on. We have interpreted this experience as a key argument in favor of a computationally open type theory such as OCC. However, we should mention that an alternative solution would be to give up the idea that type checking should be based on computation, and to employ a system like Martin-Löf's type theory or Nuprl, where type checking becomes a matter of theorem proving rather than a matter of computation alone.

Finally, we should mention another benefit that could be expected from the use of OCC in the context of this application. The added value of having such a temporal logic library available in OCC, and this has been shown by a high-level Petri net example in Chapter 8, is that the same OCC specification can be used as a basis for symbolic system execution and for interactive theorem proving. What is also interesting is that the increased computational capabilities of OCC lead to a substantial automation of certain subproofs, so that abstract execution and interactive theorem proving work hand in hand. For instance, the idea of theorem proving modulo structural and computational equations allows us to partially automate reasoning with multisets, which we exploited in Chapter 8 by conducting an interactive proof of a linear invariant in our running example of a colored net specification.

Chapter 5

First-Order Representations of Higher-Order Formalisms: A Calculus of Names and Substitutions

In Chapter 5 we switched to a very different topic that is related to OCC at two different levels. The contribution of this chapter is the development of a general approach to the representation of higher-order languages, more precisely languages with binding constructs, in equational first-order frameworks like membership equational logic and rewriting logic. Our approach is based on a new generic calculus of explicit substitutions that we called CINNI, which refers to *Calculus of Indexed Names and Named Indices*, alluding to its unconventional term representation which has originally been introduced in a similar form by Klaus Berking in the context of the λ -calculus. The notation is a unifying generalization of both the standard named notation and the well-known notation based on de Bruijn indices. The CINNI calculus is given by a number of equations that can be equipped with the standard operational interpretation based on equational rewriting, and it is generic in the sense that it can be systematically

instantiated to the syntax of different object languages. A remarkable point, which closely connects our work with earlier work on explicit substitutions, is that by instantiating our calculus to λ -terms and by restricting the set of available names to a single one, we arrive at Lescanne's calculus $\lambda\nu$ which is based on de Bruijn's representation. Indeed, both our calculus and our treatment can be regarded as a consequent generalization of Lescanne's earlier work. Regarding metatheoretical results in this chapter, we have investigated the equational and operational properties of CINNI, namely confluence and strong normalization. Furthermore, we have considered the composition of CINNI with other equations or rules, which could for instance capture the dynamic semantics of the language being represented, and we have studied coherence properties and proved preservation of confluence under a quite liberal well-formedness condition.

On the other hand, we have left the issue of preservation of strong normalization as an important subject for future work with the conjecture that it is not difficult to generalize the proof of Pierre Lescanne under a condition that is slightly stronger than our well-formed condition for confluence. Regarding further topics for future work we have explained how CINNI opens a new direction in a three-dimensional research space, namely the direction which introduces names in first-order substitution calculi. A systematic comparison with other approaches in this space would be important and could lead to further generalizations of the CINNI approach. Especially interesting seems to be the question whether CINNI can be equipped with an operationally well-behaved notion of substitution composition.

As a typical application, CINNI enables the use of membership equational logic or rewriting logic as a semantic or logical framework for the specification of higher-order languages, which for instance could be programming languages or logics, and hence allows us to use a rewriting engine such as Maude to build execution environments for such languages. We have applied our approach to the representation of the untyped λ -calculus, Abadi and Cardelli's object calculus, also called the ζ -calculus, and Milner's π -calculus for communicating and mobile systems. As a real-world application of our approach we have included a specification of an active network programming language in Appendix B. This application of rewriting logic as an executable semantic framework is a demonstration of the advantages of being able to represent both the semantics of a programming language and a notion of concurrent systems uniformly in a single specification language. Regarding the use of CINNI, it shows how explicit substitutions can completely eliminate the need for environments in the specification of an operational semantics, a point which leads to a very elegant and concise specification in this case.

Not surprisingly the main role of CINNI in this thesis was its application in the formalization of type theories, which was the subject of Chapter 6. Furthermore, the calculus is the basis for Chapter 8. Indeed, the OCC prototype has been writ-

ten in Maude and uses CINNI to implement the formal system and especially the operational semantics. However, the use of CINNI has by no means been confined to the implementation level. Indeed, we have seen that both the formal system and the set-theoretic semantics of OCC are based on the CINNI calculus. The advantage of CINNI is not only its operational first-order nature, but it also shows that the often informal assumption and treatment of α -equality can be completely avoided without major inconvenience. For instance, the proof of soundness of OCC was carried out in Chapter 8 without considerable overhead by using the equational properties of CINNI. Furthermore, the use of explicit substitutions in the OCC semantics, a technical novelty of our approach, is again very convenient, because it eliminates the need for environments typical of standard approaches to denotational semantics.

The second level at which the CINNI calculus appears in the context of OCC is its use to represent object languages in OCC itself, leading to a representation that is more concrete than, but of similar generality as, the standard higher-order abstract syntax approach. Furthermore, as a first-order approach the semantic difficulties of the higher-order abstract syntax approach are avoided, which are mainly caused by the use of the semantics-sensitive concept of function spaces to represent syntax of object languages. As in many other examples, the capabilities of OCC as an equational programming and specification language, which go beyond pure functional programming, are of key importance to obtain the benefits of executability for such representations based on explicit substitutions.

Chapter 6

Rewriting Logic as a Logical Framework: Representing Pure Type Systems

Chapter 6 is concerned with the use of rewriting logic as a logical framework and can from a more general perspective be classified as an attempt to further explore and advance the general logics methodology, which is based on an abstract logical metatheory, namely the theory of general logics, and a concrete logical framework, namely rewriting logic. The specific contribution of this chapter is to be the first systematic study of rewriting logic as logical framework for type theories and higher-order logics.

To this end, we have shown how an important class of pure type systems can be expressed using membership equational logic and rewriting logic at various levels of abstraction, ranging from the usual high-level textbook presentation modulo α -conversion to an operationally useful representation with concrete names. To accomplish the later task, we started with a suitable instance of the CINNI calculus, which we developed in the previous chapter, and introduced the class of uniform pure type systems, a more concrete variant of pure type systems which takes names seriously and solves the problem of α -closure that had been pre-

viously pointed out by Pollack. It was interesting for us that the admission of contexts with multiple declarations of the same name, which is the basis of our approach, was also one of the possibilities that Pollack considered in his work, but he also found that the approach does not scale up to type theories with dependent types, the only reason with hindsight being that he used the traditional notation with its unsatisfactory aspect of accidental hiding. So we can say that our work can be seen in this regard as a possible continuation of his earlier ideas.

The operational representation of uniform pure type systems in rewriting logic is based on the idea of representing inference rules directly as rewrite rules, so that goal-oriented deduction becomes a simple matter of rewriting modulo suitable equations. Apart from the executability, which can be exploited using a rewriting engine such as Maude, another advantage is that the algorithmic nature of the goal-oriented inference system receives a clear formal status, namely that of a transition system, and can itself become a subject of formal analysis. On the practical side, the immediate applicability of the results of this chapter has been demonstrated in Appendix A, where we used the Maude rewriting engine and our operational representation of uniform pure type systems in rewriting logic to obtain an executable prototype of the calculus of constructions with universes.

Technically, our approach again uses the notion of entailments systems from general logics to express the relationship between four formal systems that represent pure type systems at different levels of abstraction. Instead of using maps between sentences we have introduced the more general notion of correpondence, which allows us to associate to a sentence of the source system several sentences in the target system, leading to a more appropriate treatment when we have to bridge different abstraction levels in the syntax. The overall result of our treatment is a chain of sound correspondences between the high-level textbook specification and the operational implementation, which by composition proves correctness of the implementation. Furthermore, the rewriting logic approach to view inference systems as transition systems enabled us to prove further important operational confluence and termination properties at the implementation level. Since the main goal of this chapter is to explore the advantages of rewriting logic as a logical framework, we have focussed on the important subclass of pure types systems that are decidable, normalizing, functional, full, and without top sorts, pointing out that with slight modifications the corresponding semi-full systems with or without top sorts can be treated.

We should emphasise that our work is heavily based on and inspired by earlier work, especially by Pollack, van Benthem Jutting, and McKinna, who have conducted a rigorous formalization of the metatheory of pure type systems using the LEGO proof assistant. Although we have focused on representational aspects in this chapter, we think that with OCC we have created a suitable formalism to conduct metatheoretic proofs about formal systems represented in rewriting logic, but we still consider formal metatheory as one of the most challenging

applications, due to its overly syntactic and hence very technical nature. Other possibilities for future work that arise from this chapter include the treatment of additional classes of pure type systems and extensions but also the application of our approach to other type theories or typed higher-order logics. For instance, richer systems in the line of Martin-Löf's type theory and Nuprl, belong to the very different class of polymorphic type theories, in which type checking becomes as complicated as theorem proving and the use of metalevel rewrite strategies would be essential to guide the type checking process.

From the viewpoint of OCC, our occupation with pure types systems in this chapter has influenced its development in essential ways. At the theoretical level OCC employs the CINNI approach to deal with names explicitly in a way similar to its use in uniform pure type systems. At the practical level, the use of rewriting logic as a logical framework enabled us to develop an experimental prototype of OCC in Maude. The main purpose of this prototype was to study the pragmatics of OCC in the context of this thesis, but it appears to us that the rewriting-based approach of this chapter has more potential and could play a major role in an actual implementations of OCC and other type theories or logics, which have traditionally been mainly implemented using functional programming languages such as ML.

Compared with the use of type theory in Chapter 4 as a higher-order logical framework and as a metalogic to reason about the represented formalism, Chapter 6 focuses on the first-order representation of formalisms and the associated benefits of such representations for executability. Since OCC subsumes both approaches, the added value of its use for logical framework applications like this would be that both the executable representation of formal systems and inductive reasoning about the representation in higher-order logic can take place in a single language with all the benefits mentioned earlier, such as the availability of expressive types and the integration of symbolic execution and interactive theorem proving.

Chapter 7

Classical Logic via Type Theory:

A Proof-Theoretic Approach to the HOL/Nuprl Connection

Chapter 7, the last of the five chapters that we refer to as application chapters in this thesis, was again concerned with an application of type theory, but one with a quite general objective, namely the use of type theory as a classical logic to achieve formal interoperability with classical theorem provers. The chapter specifically reexamined the approach of the HOL/Nuprl connection developed by Doug Howe, which he justified using a classical hybrid set-theoretic/computational semantics for Nuprl. Howe's construction of such a semantics for a polymorphic extensional type theory was a remarkable achievement, but the resulting semantics is rather

complicated and not easy to use. On the other hand, it serves its purpose very well, namely to provide a semantic justification for classical reasoning in a type theory like Nuprl and hence for formal interoperability with theorem provers such as HOL.

The contribution of this chapter, which can be regarded as a continuation of Howe's work, is an alternative proof-theoretic justification of the HOL/Nuprl connection, which is not very specific to Nuprl, but more generally applies to Martin-Löf's type theory, and could furthermore be easily adapted to the calculus of constructions with universes that is a subsystem of OCC. To this end, we have presented the HOL/Nuprl connection in the strongly typed categorical setting of Meseguer's general logics as a combination of a map between the entailment systems of HOL and Nuprl and a theory interpretation inside the Nuprl type theory. Inspired by Howe's implementation of the HOL/Nuprl connection in the functional programming language ML, we have developed a formal executable specification of the HOL/Nuprl connection in Maude using membership equational logic as an executable metalogic. In addition, our proof-theoretic approach has led to the development of an HOL/Nuprl proof translator, which has been implemented by Naumov on top of the Nuprl system and constitutes a new application of our results that goes beyond Howe's original HOL/Nuprl connection. The development of this proof translator was inspired by the constructive nature of our soundness proof and the observation that the algorithmic content of our informal proof can be in most cases quite naturally expressed in the metalanguage of Nuprl, which is Classic ML, using just a few Nuprl tactics.

Regarding our treatment, we should point out that we were mainly interested in the level of entailment systems, with the intention of abstracting from those features of Nuprl that are not relevant in this context. Our treatment was also independent of Howe's semantics, except for the fact that it justified the consistency of the classical variant of Nuprl. General logics, however, offer the possibility of integrating both the entailment systems and the model-theoretic side, where the later is captured by the well-known notion of institution proposed by Goguen and Burstall. It seems to us that an integrated treatment of the HOL/Nuprl connection in this sense would be possible, and would constitute a consequent continuation of our work. On the practical side, it would be most interesting to further develop and complete Naumov's proof translator. The main questions in this context are how large proofs can be handled efficiently, and how the structure of formal developments can be preserved.

Concerning the design of OCC, that we compared with Nuprl in Chapter 8, a major design decision was to preserve the monomorphic character of the calculus of constructions, since this allows us to define the intuitive classical set-theoretic semantics of OCC that we presented. Under the classical semantics, propositions in OCC have the expected two-valued interpretations, and HOL can be trivially embedded into OCC. On the other hand, the formal system of OCC

is not classically biased, and we do not want to exclude, but rather encourage, the development of alternative semantic models for OCC. For instance, a semantics based on a richer structure of the propositional universe is conceivable and, as explained in Chapter 8, the idea of the HOL/Nuprl connection can then be applied to obtain an embedding of HOL in OCC. This simply gives rise to an impredicative classical logic hosted by the boolean data type without destroying the possible intensionality of the propositions-as-types logic of OCC, which resides in the propositional universes, and can therefore be considered a more conservative approach to classical reasoning with OCC than the strong assumption of classical axioms for the propositional universe. Last but not least, we can learn from the HOL/Nuprl connection that already the purely predicative instances of OCC are absolutely satisfactory for classical mathematics, a possibility that we think is worth exploiting in the future, because of the straightforward set-theoretic semantics that can be used in this case.

From a slightly more general perspective, the theme of Chapter 7 was the use of the methodology of general logics in the development of logic translators, and we studied Howe's HOL/Nuprl connection as a particular instance of this approach. However, the use of membership equational logic as an executable metalogic in our Maude case study was not completely satisfactory, because its simple algebraic type system is not expressive enough to adequately reflect our strongly typed categorical treatment of the HOL/Nuprl connection, which could be very adequately expressed using the combination of equational logic and dependent types of OCC. Although we have not revisited this application specifically in this thesis, we have given some pieces of evidence in Chapter 8 supporting our view of OCC as a possible framework for computational category theory, which could not only provide an appropriate foundation to formalize concepts of general logics, but would also have other interesting applications to which we will come back below.

Chapter 8

Towards a Unified Language: The Open Calculus of Constructions

After highlighting the main streams of inspiration for our research in Chapter 8 from the viewpoint of various applications of rewriting logic and type theory, we would like to conclude by summarizing the main contributions of the last major chapter and by giving an outlook on possible future work that we think would help to continue our line of research.

The main contribution of Chapter 8 was the study of a formalism that combines the features of membership equational logic, rewriting logic, Martin-Löf's type theory, and the calculus of constructions to overcome some of the limitations of each individual formalism. We have tried to keep the resulting formalism

minimal, and especially have avoided the addition of any distractive features, to focus on the interaction between two primitive concepts, namely dependent types and an operational semantics based on conditional rewriting modulo equations.

The resulting system, the *open calculus of constructions* (OCC), is parameterized by a cumulative universe hierarchy which can contain impredicative and predicative universes. Our first step was the development of a classical set-theoretic semantics, which in contrast to standard approaches does not rely on the formal system and is in particular defined independently of the operational semantics of the type theory. An important aspect of this semantics, which should be classified as a proof-irrelevance semantics (for impredicative universes), is that universe subtyping is directly interpreted as set-theoretic inclusion and without reference to the formal system, a feature that to our knowledge is not present in any of the existing approaches. In our opinion, we have constructed a true set-theoretic semantics of utmost simplicity, which shows that the type theoretic approach to treat logic as a derived concept is absolutely appropriate to deal with classical mathematics. The semantics has also been developed with the intention of enabling formal interoperability with existing theorem provers such as HOL, PVS, LEGO, and COQ. For us these are important issues, since OCC is intended as an open formalism, in the sense that the user should not be required to justify all axioms or theorems inside the formal system itself. After introducing the semantics of OCC we presented its formal system, which simultaneously defines judgements for type checking, type inference, and the judgements of a typed operational semantics based on conditional rewriting modulo equations. The formal system of OCC should be regarded as a framework that admits implementations with various degrees of generality, and we have informally motivated the rules by explaining how they would be realized in a typical implementation. The theoretical part of this chapter concludes with a proof of soundness and consistency of the formal system w.r.t. the semantics that we defined earlier.

One lesson suggested by our research is that the combination of two very different paradigms can lead to synergistic effects which are practically highly beneficial on the one hand but are theoretically very challenging on the other. In the case of the calculus of constructions and rewriting logic with its equational sublogic it turned out that, loosely speaking, the combination is much more powerful than the sum of its parts. First of all, the computational expressiveness of the typed λ -calculus is considerably enhanced by the computational versatility of equational logic and rewriting logic, which conversely benefits from the higher-order features of the typed λ -calculus. In contrast to the calculus of constructions, which is based on a fixed notion of computation, the computational system of OCC is open in the sense that the operational semantics depends on the context and can hence be specified by the user. Regarding specifications viewed as particular contexts, this corresponds to the approach of membership equational logic and rewriting logic, where each specification is equipped with an individual operational semantics.

Our main guiding principle, which we consider as a key characteristic of the calculus of constructions and some versions of Martin-Löf's type theory, is that type checking with dependent types should be ultimately based on a notion of computation. Therefore type checking in OCC immediately benefits from the open computational system and the increased computational expressiveness. In fact, we more generally allow a circular interdependency between type checking and the operational semantics, in the sense that new operationally meaningful propositions can be introduced in the context only because of type checking capabilities which rely on previously introduced operational propositions. In practice, we found it often to be the case that new concepts can be elegantly introduced from existing concepts using this form of bootstrapping. Another main argument in favor of an open computational system is that true modularity, based on executable specifications and potentially hidden implementations, cannot be achieved in a closed computational system, provided that we do not abandon the principle that type checking should be based on computation.

Still speaking about the combination of two very different paradigms, another synergy is exhibited by the integration of type theory and rewriting logic, as opposed to equational logic, because this allows us to leave the realm of pure functional and even equational programming, making the capabilities of type theory available for instance in the specification of concurrent and distributed systems. Using a formalization of enriched transition categories in OCC, we have obtained a very direct and strongly typed representation of rewrite theories as OCC specifications, which exhibit a very elegant correspondence between rewrite judgements and typing judgements. Although we have used different classes of Petri nets to illustrate the advantages, the benefits of a strong typing discipline carry over to many other formalisms that can be specified using rewriting logic as a semantic or logical framework, including for instance Meseguer's class of concurrent object systems which combines the features of concurrency and object-oriented programming.

On the practical side, we used the techniques of Chapters 5 and 6 in the development of an experimental prototype of OCC in Maude. A detailed discussion of this prototype, although interesting due to its unconventional reflective architecture, was beyond the scope of this thesis. Instead, we have used the OCC prototype to develop various examples and case studies, that we have included in this thesis to explain the pragmatic aspects of OCC, which cannot be conveyed by just defining a formal system and its semantics. In tutorial form we have in the first major section covered the use of OCC in programming and specification and then in the second major section have discussed interactive theorem proving as a special case of functional programming, which is typically conducted using a style of goal-oriented refinement.

Our first subject was the use of executable equational specifications with examples illustrating the use of structural, computational and assertional equality,

definition by pattern matching modulo structural equations, the representation of membership equational specifications, the employment of various generalizations, including higher-order equational specifications, and furthermore the treatment of equationally defined datatypes.

We moreover revisited the power of dependent types and universes, covering type operators, explicit polymorphism, and the use of dependent types for parameterization in general. The interaction between dependent types and the operational capabilities of OCC has been nicely illustrated by expressing concepts known from the programming languages Haskell and Gofer, like type classes and inheritance, multiple parameter classes and type relations, and we have suggested OCC as a framework for exploring further generalizations of these ideas. We next touched upon type equality, type casting, and further discussed the enhanced typing capabilities in the context of operational propositions. In addition, by using an example involving heterogeneous data structures, we have further illustrated the equational style of programming with dependent types in OCC, which leads to quite natural multi-level descriptions, and hence addresses a difficulty with the use of dependent types in programming observed by Pollack.

In many of the previous examples we have employed a higher-order generalization of the algebraic specification style, which under the initial semantics defines elements of a type in terms of their constructors. In many applications it is more appropriate to define elements of a type in terms of observers, which leads to the dual style of behavioral specification which, as the name indicates, focuses on behavior rather than on structure. Specifically, we discussed how behavioral specifications, with particular emphasis on executability, can be expressed in OCC following the final coalgebra and hidden algebra approach. The equational specification and computation with infinite data structures is one of the applications discussed, but the broad significance of final coalgebras in computer science, e.g. as witnessed by the notion of bisimulation in concurrency theory, suggests that both the algebraic and the coalgebraic specification styles should be supported on an equal footing. Indeed, thanks to its equational nature, OCC does not seem to be biased towards any of these styles. We also included a section on another subject, namely the representation of object languages in OCC, since this is important for its application as a semantic and logical framework. In this context we have covered a variation of higher-order abstract syntax compatible with a classical semantics and the more concrete alternative of first-order representations via the CINNI calculus developed in Chapter 5.

In the subsequent section on elimination and coelimination principles we have shown how the inductive and coinductive nature of types can be expressed without having inductive and coinductive types as primitive concepts. This approach is not new and has already been adopted in the implementation of UTT in LEGO, but the use of OCC suggests two possible improvements that are worth pointing out. First, the computational aspect of elimination and coelimination principles

can be expressed in OCC itself, rather than being based on ad hoc extensions of the type theory, which is ECC in the case of LEGO. Second and more importantly, a more general class of equational inductive definitions, which is not subsumed by the standard schemes, finite multisets being a typical example, can be handled in a satisfactory way. In the next section we gave a minimal specification of higher-order logic in OCC, we treated Leibnitz equality, functional extensionality, and other classical axioms such as logical extensionality and proof irrelevance, which can be justified via the OCC semantics but are not enforced by its rather weak formal system. We also explained the representation of weak and strong subset/existential types and the possibilities of information hiding, which can be controlled by the user by specifying suitable computational equations for extracting pieces of visible information. Finally, we covered the alternative approach to classical logic in OCC suggested by the HOL/Nuprl connection.

The next section deals with the representation of algebras and categories in OCC. We have reviewed how algebras, categories are a special case, can be treated as first-class citizens based on the well-known idea of strong Σ -types, which are primitive in some type theories such as ECC/UTT or Nuprl, are a special case of inductive types in CIC, and can be equationally specified in OCC. We moreover covered the categorical notions of initial algebras and final coalgebras, which provide a more abstract and more general treatment of inductive and coinductive definitions than the ad hoc axiomatization discussed earlier. The fact that the standard treatment is too limited and especially does not cover equational inductive definitions which are very common in MEL, motivated our choice not to equip OCC with a primitive notion of inductive and coinductive types. Instead, we favored using the categorical approach, from which the standard elimination and coelimination principles can be derived, as we have demonstrated in a number of examples. Another use of categories that we discussed in this section are enriched transition categories, which are for instance used in the semantics of Petri nets and, more generally, in the semantics of rewriting logic. Using an example of an enriched transition category generated by a place/transition net, we have illustrated the expressiveness of the type system by specifying the class of Best/Devillers processes as a dependent type in OCC, which is a clear improvement relative to the weakly typed MEL approach of Chapter 3. In both applications of categories that we discussed in this section, the internalization of inductive/coinductive definitions and the interpretation as enriched transition categories, it turned out that the computational openness of OCC is the key to a successful formalization.

What has been covered in the previous sections is concerned with the interaction between type theory and equational logic. The additional benefits of the rewriting logic features of OCC have been first exploited in the subsequent section on the representation of executable rewrite specifications. As examples we have especially treated automata and place/transition nets as a case of monoidal rewrite

specifications, but we have also discussed how rewrite specifications with a more complex state space specification can be treated using a representation of colored net specifications in OCC as an example. In contrast to rewriting logic, OCC admits multiple rewrite predicates and does not have any built-in rewrite axioms such as reflexivity, transitivity, or compatibility. Only those properties that are needed for the application at hand should be explicitly specified by the user, to keep the resulting induction principles manageable. Furthermore, what counts as a rewrite proof in OCC, and hence as an action in the specified transition systems, is again entirely up to the user. In our example of a place/transition net, represented in OCC via its rewriting semantics, we have chosen as actions exactly the standard proof terms of rewriting logic, that is we have employed the enriched transition category introduced earlier.

This concluded our tour of the programming and specification capabilities of OCC. All these capabilities can be beneficially employed in interactive theorem proving in OCC, which we have addressed in the second major section. Topics that we touched upon include a review of the standard goal-oriented refinement approach to the construction of proofs and functional programs, reasoning with equality in OCC, inductive and coinductive theorem proving with partial automation thanks to the operational semantics of OCC, and specifically we have illustrated theorem proving modulo computational and structural equations by proving a linear invariant in our running example of a colored net specification expressed using a fragment of the UNITY-style temporal logic of Chapter 4.

In summary, the last part of Chapter 8 gives a broad overview of the pragmatics of OCC, simultaneously providing a proof-of-concept for our approach. We have included many subjects that are well-known in the practice of type theory or equational/rewriting logic and have presented them in tutorial form to make the material accessible to readers which are not familiar with both fields. This also helped us to clearly point out where OCC leads to improvements, or even to new possibilities. We are aware of the fact that, due to the wealth of applications, most subjects could not be treated in detail and many of them deserve more attention.

An interesting experience for us was that very different and apparently not closely related needs from a variety of applications could all be reduced to the two primitive concepts present in OCC, namely dependent types and conditional rewriting modulo. However, we should emphasize that OCC is by no means intended as an exhaustive answer to the question about what a good unified language for programming, specification, and interactive theorem proving could actually be. Instead, we regard it as the exploration of one promising direction in a space of many possibilities, which at this point focuses on a few basic features of such a language but ignores many other ingredients that are practically important.

In the remainder of this chapter we wish to point out a number of topics that we think are relevant for continuing this line of research. Again we stay focused on the central theme of a unified language for programming, specification and interactive theorem proving rather than getting into the more specific potential for future research that we already discussed in the final remarks of each the previous chapters.

We have mentioned earlier that there is a price to be paid for the increase in expressiveness in OCC relative to that of the formalisms it originated from. Indeed, the generality of the approach makes the study of operational properties of OCC, an issue that has not been addressed in this thesis, a difficult enterprise. It is not only that the notions of confluence and normalization do not make sense without reference to a context, but since the relevant context varies with the position inside a term, we are concerned with a dynamics that is more complex than what is traditionally considered. A possible approach to a metatheoretic study is to consider only certain admissible contexts and certain admissible terms in such contexts, and to investigate operational properties relative to such assumptions. A better metatheoretic understanding would also be useful in view of the dynamic type checking conditions that are needed in the general case. In practice most of these conditions seem to be superfluous if the relevant contexts are known to satisfy certain minimal properties. An exact study of this issue would be important to obtain an efficient implementation that uses static analysis to minimize the generation of type checking conditions.

Also we think that additional work is needed on the model-theoretic side. Our classical set-theoretic semantics is suitable in the context of this thesis and in many applications which are concerned with classical mathematical reasoning, but on the other hand it is obviously much more abstract than the formal system of OCC itself. This suggests that there is some potential for alternative semantics, which could be either classical or constructive. Following Martin-Löf's view, which is also and especially the philosophy behind the Nuprl type theory, we consider semantics as more fundamental than the formal system, which is overly concrete due to its syntactic nature. In other words, it would be desirable to make more explicit the essential aspects of the formal system at the semantic level (possibly using an intermediate formal system with more liberal typing judgements). OCC is not intended as an open type theory like Nuprl, which encourages the introduction of entirely new types and inference rules, but it is mainly designed with the intention of exploiting the power of dependent types to develop other concepts internally (similar to the motivation behind Martin-Löf's logical framework). Nevertheless we find that there are a few potential extensions which seem to fit particularly well the minimalistic approach of OCC, namely implicitly dependent types as a natural counterpart to explicitly dependent types, universe types as a semantic approach to universe polymorphism, and direct computation rules. A generalization of Nuprl's direct computation rules to the richer computa-

tional system of OCC would provide a semantic justification for the elimination of dynamic type checking conditions under suitable restrictions. We leave the integration of these ideas for future work. In view of potential difficulties with impredicative universes, we think that the standard classical semantics for the predicative instances of OCC might be a good starting point. We have already indicated that the study of the predicative instances of OCC would be of independent interest, because it still remains a fully satisfactory formalism for doing classical mathematics by adopting the approach of the HOL/Nuprl connection, ideally in form of a classical propositional universe which is neither predicative nor impredicative, a possibility admitted by our notion of OCC signatures.

Concerning extensions of OCC one has to carefully distinguish between actual extensions of the formal system and certain extensions which are realized at the metalevel and often referred to as syntactic sugar. For instance, in the OCC prototype, a form of implicit dependent types and corresponding implicit λ -abstractions are available, but they have been explained as syntactic sugar for their explicit counterparts following the approach of LEGO, which as we explained is semantically not as satisfactory as having them as primitives. Another possible extension of OCC that we have discussed is universe polymorphism, that can be realized at the metalevel, but it would become less ad hoc if it could be integrated into the semantics and into the formal system, e.g. by means of universe types, so that it appears as a special case of ordinary polymorphism. In view of applications involving categorical concepts, such an extension is another important issue in the future. Other uses of metalevel extensions, seem to be completely satisfactory, such as schemes for certain classes of inductive and coinductive definitions as in LEGO, generalized records and strong Σ -types being special cases, but such schemes have not been implemented in the OCC prototype so far. Further metalevel extensions that OCC could benefit from are overloading, as it is for instance possible in Maude, and implicit coercions as a general approach to subtyping. An excellent implementation of implicit coercions is, for instance, available in COQ and has led to considerable improvements in readability in complex developments. Further issues that are important in practice are modularity and information hiding. Thinking of a module as a specification which may be satisfied by many possible implementations, a notion of modules-as-objects could be treated very similar to the algebras-as-objects approach, but again additional syntactic sugar would be useful to support this style, which otherwise is tedious to maintain. These are just some of the desirable practical features that are missing in the OCC prototype, which we think should be addressed in an actual implementation to make it usable for large-scale developments.

Another very different topic, that we have left as an important subject for future work, is to explore the relationship between standard approaches to higher-order rewriting and the approach adopted by OCC, which applies standard conditional rewriting to possibly higher-order terms. As we emphasized at various places,

OCC uses an underlying instance of the CINNI calculus to avoid the assumption of structural α -equality. Still we found that, thanks to the possible use of universally quantified conditions, the usual examples, which are often used to motivate the need for higher-order rewriting, can be expressed in OCC. A careful study of the expressiveness of our approach, and also a comparison with existing first-order approaches to higher order rewriting mentioned in Chapter 5, would be highly interesting, and could be conducted independently of the type-theoretic features of OCC.

Yet another direction for future work is concerned with the categorical approach to inductive and coinductive definitions suggested by the initial algebra and final coalgebra paradigms. We have explained in various examples how specifications can be structured along these notions, how parameterization is expressed, and how elimination and coelimination principles together with their computational behavior become derived notions. However, in order to pursue the approach in a more systematic way it seems that a development of a form of computational category theory in OCC would be the right starting point. From a logical viewpoint such a formalization could be similar to that conducted by Huet and Saïbi using COQ, but an important objective would be to exploit the computational features of OCC, so that the applications mentioned above become possible. A development of computational category theory would also constitute a larger case study in the formalization of mathematical concepts, and the experience gained would be a valuable input for possible improvements and further development of OCC. We have already conveyed our view that the general categorical approach to inductive and coinductive types would be more satisfactory than the use of the standard schemes. A key question, however, is if it can be realized, possibly using a modest amount of syntactic sugar, in such a way that it is convenient enough in practice.

Last but not least, although various small examples have been developed in the context of this thesis, we think that larger case studies would be an important next step on the way to develop a mature system based on our research. A development of computational category theory, that has already been mentioned, would constitute an interesting application in the formalization of mathematics. Another very challenging real-world application would be to conduct formal proofs about our executable specification of an active network programming language that is based on the CINNI approach to the representation of object languages and employs a multiset representation of the distributed network state. It seems to us that OCC has the right features to support and partially automate proofs that we so far conducted using conventional mathematical means. Of course, for applications of this complexity, it would be very desirable to have formal interoperability with popular theorem provers such as HOL or PVS, a feature which we think does not pose any theoretical difficulties, since OCC is designed as an open formalism, which does not enforce either logical or computational conservativity.

Appendix

Appendix A

Pure Type Systems in Rewriting Logic

A.1 The Executable Specification in Maude

In the following we give the full Maude specification of the rewriting-based version of optimized uniform PTSs from Chapter 6.

Recall that ROUPTS is a system module, which is parameterized over functional PTS signatures without top sorts as specified by the theory FPTSSIG.

```
fth FPTSSIG is
sort Sorts .
op Axioms : Sorts -> Sorts .
op Rules : Sorts Sorts -> Sorts .
endfth

mod ROUPTS[S :: FPTSSIG] is

protecting QID .
protecting NAT .

sorts Var Trm .
op _{ } : Qid Nat -> Var [ prec 0 gather (E &) ] .
subsort Var < Trm .
subsort Sorts < Trm .
op __ : Trm Trm -> Trm [ prec 21 gather (E e) ] .
op [_: ]_ : Qid Trm Trm -> Trm [ prec 30 gather (& & E) ] .
op {:_: }_ : Qid Trm Trm -> Trm [ prec 30 gather (& & E) ] .

vars s s1 s2 s3 : Sorts .
vars X Y Z : Qid .
```

```

vars A B M N O P Q R T A' B' M' N' T' T1 T2 T3 U U' : Trm .

sort Subst .

op [_:=_] : Qid Trm -> Subst [ prec 30 gather (& &) ] .
op [shift_] : Qid -> Subst [ prec 30 gather (&) ] .
op [lift__] : Qid Subst -> Subst [ prec 30 gather (& &) ] .
op __ : Subst Trm -> Trm [ prec 30 gather (E E) ] .

var S : Subst .
vars n m : Nat .

eq ([X := M] (X{0})) = M .
eq ([X := M] (X{suc(m)})) = (X{m}) .
ceq ([X := M] (Y{n})) = (Y{n}) if X /= Y .

eq ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X /= Y .

eq ([lift X S] (X{0})) = (X{0}) .
eq ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X /= Y .

eq (S s) = s .
eq (S (M N)) = ((S M) (S N)) .
eq S ([X : A] M) = [X : (S A)] ([lift X S] M) .
eq S ({X : A} M) = {X : (S A)} ([lift X S] M) .

sort Context .

op emptyContext : -> Context .
op _:_ : Qid Trm -> Context .
op _,- : Context Context -> Context [assoc id: emptyContext] .

var G G1 G2 : Context .

sort Judgement .

op _||- : Context -> Judgement [ gather (&) ] .
op _|_- : Context Trm Trm -> Judgement [ gather (& & &) ] .
op _||- : Context Trm Trm -> Judgement [ gather (& & &) ] .

op _=_ : Trm Trm -> Judgement .
op _<->_ : Trm Trm -> Judgement .
op _Sort : Trm -> Judgement .

```

```

op ‘(,_,_’)Rule : Trm Trm Trm -> Judgement .
op _|_>:_ : Context Trm Trm -> Judgement [ gather (& & &) ] .
op _|_‘(->:_’)‘(->:_’)->:_ : Context Trm Trm Trm Trm ->
    Judgement .

var J : Judgement .

sort JudgementList .

op emptyJudgementList : -> JudgementList .
subsort Judgement < JudgementList .
op __ : JudgementList JudgementList -> JudgementList
    [assoc id: emptyJudgementList] .

var JS : JudgementList .

sort Configuration .

op {{_}} : JudgementList -> Configuration [ gather (&) ] .

sort MetaVar .
subsort MetaVar < Trm .
op ? : Qid -> MetaVar .
var NEW NEW1 NEW2 : Qid .
var MV MV' : MetaVar .

op <_:=>_ : MetaVar Trm Trm -> Trm
    [prec 59 gather (& & &)] .
op <_:=>_ : MetaVar Trm Subst -> Subst
    [prec 59 gather (& & &)] .
op <_:=>_ : MetaVar Trm Context -> Context
    [prec 59 gather (& & &)] .
op <_:=>_ : MetaVar Trm Judgement -> Judgement
    [prec 59 gather (& & &)] .
op <_:=>_ : MetaVar Trm JudgementList -> JudgementList
    [prec 59 gather (& & &)] .

eq < MV := Q > s = s .
eq < MV := Q > X{m} = X{m} .
eq < MV := Q > (M N) = ((< MV := Q > M) (< MV := Q > N)) .
eq < MV := Q > ([X : T] M) = [X : (< MV := Q > T)] (< MV := Q > M) .
eq < MV := Q > ({X : T} M) = {X : (< MV := Q > T)} (< MV := Q > M) .

```

```

eq < MV := Q > MV = Q .
ceq < MV := Q > MV' = MV' if MV /= MV' .

eq < MV := Q > emptyContext = emptyContext .
eq < MV := Q > [X : T] = [X : (< MV := Q > T)] .
ceq < MV := Q > (G1 G2) = (< MV := Q > G1) (< MV := Q > G2)
  if G1 /= emptyContext /\ G2 /= emptyContext .

eq < MV := Q > [X := M] = [X := (< MV := Q > M)] .
eq < MV := Q > [shift X] = [shift X] .
eq < MV := Q > [lift X S] = [lift X (< MV := Q > S)] .
eq < MV := Q > (S M) = ((< MV := Q > S) (< MV := Q > M)) .

eq < MV := Q > (T Sort) =
  (< MV := Q > T) Sort .
eq < MV := Q > ((T1,T2,T3) Rule) =
  (< MV := Q > T1, < MV := Q > T2, < MV := Q > T3) Rule .
eq < MV := Q > (G ||-) =
  ((< MV := Q > G) ||-) .
eq < MV := Q > (G |- M : T) =
  (< MV := Q > G) |- (< MV := Q > M) : (< MV := Q > T) .
eq < MV := Q > (G ||- M : T) =
  (< MV := Q > G) ||- (< MV := Q > M) : (< MV := Q > T) .
eq < MV := Q > (G |- M ->: T) =
  (< MV := Q > G) |- (< MV := Q > M) ->: (< MV := Q > T) .
eq < MV := Q > (G |- (M ->: T1)(N ->: T2) ->: T') =
  (< MV := Q > G) |-
    ((< MV := Q > M) ->: (< MV := Q > T1))
    ((< MV := Q > N) ->: (< MV := Q > T2)) ->: (< MV := Q > T') .
eq < MV := Q > (M = N) =
  (< MV := Q > M) = (< MV := Q > N) .
eq < MV := Q > (M <-> N) =
  (< MV := Q > M) <-> (< MV := Q > N) .

eq < MV := Q > emptyJudgementList = emptyJudgementList .
ceq < MV := Q > (J JS) =
  (< MV := Q > J) (< MV := Q > JS) if JS /= emptyJudgementList .

sort WhNf WhReducible .

subsort WhNf < Trm .

subsort Sorts < WhNf .
subsort Var < WhNf .

```

```

mb ([X : A] M) : WhNf .
mb ({X : A} B) : WhNf .
mb (s N) : WhNf .
mb (X{m} N) : WhNf .
cmb ((P Q) N) : WhNf if (P Q) : WhNf .
mb (({X : A} M) N) : WhNf .

subsort WhReducible < Trm .

mb (([X : A] M) N) : WhReducible .
cmb (M N) : WhReducible if M : WhReducible .

op whnf : Trm -> Trm? .

ceq whnf(M) = M if M : WhNf .
eq whnf(([X : A] M) N) = whnf([X := N] M) .
ceq whnf(M N) = whnf(whnf(M) N) if M : WhReducible .

sort Trm? .
subsort Trm < Trm? .

op lookup : Context Var -> Trm? .
eq lookup((G, (X : A)), X{0}) = [shift X] A .
eq lookup((G, (X : A)), X{suc(m)}) = [shift X] lookup(G, X{m}) .
ceq lookup((G, (X : A)), Y{m}) = lookup(G, Y{m}) if (X /= Y) .

r1 [Subst] : {{ (MV = A) JS }} => {{ < MV := A > JS }} .

r1 [Conv1] : {{ (s <-> s) JS }} => {{ JS }} .

r1 [Conv2] : {{ (X{m} <-> X{m}) JS }} => {{ JS }} .

crl [Conv3] : {{ ((M N) <-> (M' N')) JS }} =>
  {{ (M <-> M') (N <-> N') JS }}
  if (M N) : WhNf /\ (M' N') : WhNf .

r1 [Conv4] : {{ ({X : A} T <-> {Y : A'} T') JS }} =>
  {{ (A <-> A')
    ([X := Y{0}] [shift Y] T <-> T') JS }} .

r1 [Conv5] : {{ ([X : A] M <-> [Y : A'] M') JS }} =>
  {{ (A <-> A')
    ([X := Y{0}] [shift Y] M <-> M') JS }} .

```

```

crl [Conv6] : {{ (M <-> N) JS }} =>
              {{ (whnf(M) <-> N) JS }}
              if M : WhReducible .

crl [Conv7] : {{ (M <-> N) JS }} =>
              {{ (M <-> whnf(N)) JS }}
              if N : WhReducible .

rl [Sort] : {{ (s Sort) JS }} => {{ JS }} .

rl [Rule] : {{ ((s1,s2,MV) Rule) JS }} =>
            {{ (MV = Rules(s1,s2)) JS }} .

rl [Ax] : {{ (G |- s ->: MV) JS }} =>
          {{ (MV = Axioms(s)) JS }} .

crl [Lookup] : {{ (G |- X{m} ->: MV) JS }} =>
               {{ (MV = lookup(G,X{m})) JS }}
               if lookup(G,X{m}) : Trm .

rl [Pi] : {{ (G |- {X : A} B ->: MV) JS }} =>
          {{ (G |- A ->: ?(NEW1)) (? (NEW1) Sort)
            (G, (X : A) |- B ->: ?(NEW2))
            ((?(NEW1), ?(NEW2), MV) Rule) JS }} .

rl [Lda] : {{ (G |- [X : A] M ->: MV) JS }} =>
           {{ (G |- A ->: ?(NEW1)) (? (NEW1) Sort)
             (G, (X : A) |- M ->: ?(NEW2))
             (MV = {X : A} ?(NEW2)) JS }} .

rl [App1] : {{ (G |- (M N) ->: MV) JS }} =>
            {{ (G |- M ->: ?(NEW1)) (G |- N ->: ?(NEW2))
              (G |- (M ->: ?(NEW1))(N ->: ?(NEW2)) ->: MV) JS }} .

rl [App2] : {{ (G |- (M ->: {X : A} B)(N ->: A') ->: MV) JS }} =>
            {{ (A <-> A') (MV = [X := N] B) JS }} .

crl [Norm1] : {{ (T Sort) JS }} =>
              {{ (whnf(T) Sort) JS }}
              if T : WhReducible .

crl [Norm2] : {{ ((A,B,T) Rule) JS }} =>
              {{ ((whnf(A),B,T) Rule) JS }}
              if A : WhReducible .

```

```

cr1 [Norm3] : {{ ((A,B,T) Rule) JS }} =>
              {{ ((A,whnf(B),T) Rule) JS }}
              if B : WhReducible .

cr1 [Norm4] : {{ (G |- (M ->: A)(N ->: B) ->: T) JS }} =>
              {{ (G |- (M ->: whnf(A))(N ->: B) ->: T) JS }}
              if A : WhReducible .

r1 [Aux] :    {{ (G |- M : A) JS }} =>
              {{ (G |- A ->: ?(NEW1)) (? (NEW1) Sort)
                (G |- M ->: ?(NEW2)) (? (NEW2) <-> A) JS }} .

r1 [Ctxt1] :  {{ (emptyContext ||-) JS }} => {{ JS }} .

r1 [Ctxt2] :  {{ (G,(X : A) ||-) JS }} =>
              {{ (G ||-) (G |- A ->: ?(NEW)) (? (NEW) Sort) JS }} .

r1 [Main] :   {{ (G ||- M : A) JS }} =>
              {{ (G ||-) (G |- M : A) JS }} .

endm

```

For efficiency reasons we are using an extension of Maude that has built-in support for metavariables to execute this specification. First, it has a built-in mechanism (a rewrite strategy from an abstract point of view) to choose fresh metavariable names written as $?(NEW)$, $?(NEW1)$, $?(NEW2)$, Second, we extended the Maude engine by a built-in version of metavariable substitution $\langle_ := _ \rangle_$, which behaves precisely as specified by the corresponding equations given above.

A.2 An Application: Validation of Proofs

To illustrate the use of this module we instantiate it to the calculus of constructions with universes and recheck a proof of the “small” deduction theorem which is part of the LEGO distribution. The proof object has been generated by the LEGO system. Implicit arguments have been made explicit and definitions are translated to Maude definitions. Of course, the following is only a low-level illustration of the use of `ROUPTS[FPTS-SIG-CC+]` in the most direct way that is not intended for proof development.

We begin with the functional module `FPTS-SIG-CC+` which formalizes a concrete functional PTS signature without top sorts, namely of the calculus of constructions extended by an infinite predicative universe hierarchy (`CC+`).

```

fmod FPTS-SIG-CC+ is

protecting NAT .

sort Sorts .
op Axioms : Sorts -> Sorts .
op Rules : Sorts Sorts -> Sorts .

op Prop : -> Sorts .
op Type : Nat -> Sorts .

var m n : Nat .

eq Axioms(Prop) = Type(0) .
eq Axioms(Type(n)) = Type(suc(n)) .

eq Rules(Prop,Prop) = Prop .
eq Rules(Prop,Type(n)) = Type(n) .
eq Rules(Type(m),Prop) = Prop .
eq Rules(Type(m),Type(n)) = Type(max(m,n)) .

endfm

```

Below we use the module ROUPTS-CC+[FPTS-SIG-CC+] which is an instance of the system module ROUPTS given before for the previous PTS signature. We have inserted some rewrites (initiated by `rew`) to illustrate the execution.

```

mod DedThm is

--- We use UPTS instantiated to the calculus of constrictions with
--- universes (CC+).

protecting ROUPTS[FPTS-SIG-CC+] .

--- the usual definition of ->

var TRMA TRMB : Trm .
op _->_ : Trm Trm -> Trm [ prec 30 gather (e E) ] .
eq (TRMA -> TRMB) = {'_ : TRMA} [shift '_] TRMB .

--- the context in which all the following proofs are carried out

op CONTEXT : -> Context .
eq CONTEXT = ('o : Prop)
              ('arr : ('o{0} -> 'o{0} -> 'o{0})) .

```

```

--- first we define the entailment relation for minimal logic

op k : -> Trm .
eq k = ['A : 'o{0}]['B : 'o{0}] ('arr{0} 'A{0} ('arr{0} 'B{0} 'A{0})) .

rew {{{CONTEXT |- k : 'o{0} -> 'o{0} -> 'o{0}}}} .

rewrites: 331 in 0ms cpu (10ms real) (~ rewrites/second)
result Configuration: {{{emptyJudgementList}}}

op s : -> Trm .
eq s = ['A : 'o{0}]['B : 'o{0}]['C : 'o{0}]
      'arr{0} ('arr{0} 'A{0} ('arr{0} 'B{0} 'C{0}))
      ('arr{0} ('arr{0} 'A{0} 'B{0}) ('arr{0} 'A{0} 'C{0})) .

rew {{{CONTEXT |- s : 'o{0} -> 'o{0} -> 'o{0} -> 'o{0}}}} .

rewrites: 712 in 10ms cpu (30ms real) (71200 rewrites/second)
result Configuration: {{{emptyJudgementList}}}

--- entails is defined by an impredicative encoding of its own induction
--- principle. (entails G H X) means {G,H} |- X.

op entails : -> Trm .
eq entails =
  ( [ 'G : 'o{0} ] ( [ 'H : 'o{0} ] ( [ 'X : 'o{0} ] ( { 'P : (
    'o{0} -> Prop ) } ( ( 'P{0} 'G{0} ) -> ( ( 'P{0} 'H{0} ) -> (
    ( { 'a : 'o{0} } ( { 'b : 'o{0} } ( 'P{0} ( ( k 'a{0} ) 'b{0} ) )
    ) ) -> ( ( { 'a : 'o{0} } ( { 'b : 'o{0} } ( { 'c : 'o{0} } (
    'P{0} ( ( ( s 'a{0} ) 'b{0} ) 'c{0} ) ) ) ) -> ( ( { 'a : 'o{0}
    } ( { 'b : 'o{0} } ( ( 'P{0} ( ( 'arr{0} 'a{0} ) 'b{0} ) ) -> ( (
    'P{0} 'a{0} ) -> ( 'P{0} 'b{0} ) ) ) ) ) -> ( 'P{0} 'X{0} ) ) ) )
    ) ) ) ) ) .

rew {{{CONTEXT |- entails : 'o{0} -> 'o{0} -> 'o{0} -> Prop}}}} .

rewrites: 3968 in 70ms cpu (80ms real) (56685 rewrites/second)
result Configuration: {{{emptyJudgementList}}}

--- now we prove a couple of lemmas corresponding to the cases of the
--- inductive proof

op lemma-identity : -> Trm .
eq lemma-identity =

```

```
( [ 'G : 'o{0} ] ( [ 'H : 'o{0} ] ( [ 'A : 'o{0} ] ( [ 'P : (
'o{0} -> Prop ) ] ( [ '_ : ( 'P{0} 'G{0} ) ] ( [ '_ : ( 'P{0}
'H{0} ) ] ( [ 'K : ( { 'a : 'o{0} } ( { 'b : 'o{0} } ( 'P{0} ( ( k
'a{0} ) 'b{0} ) ) ) ) ] ( [ 'S : ( { 'a : 'o{0} } ( { 'b : 'o{0} }
( { 'c : 'o{0} } ( 'P{0} ( ( ( s 'a{0} ) 'b{0} ) 'c{0} ) ) ) ) ]
( [ 'MP : ( { 'a : 'o{0} } ( { 'b : 'o{0} } ( ( 'P{0} ( ( 'arr{0}
'a{0} ) 'b{0} ) ) -> ( ( 'P{0} 'a{0} ) -> ( 'P{0} 'b{0} ) ) ) ) )
] ( ( ( ( 'MP{0} ( ( 'arr{0} 'A{0} ) ( ( 'arr{0} 'A{0} ) 'A{0} ) )
) ( ( 'arr{0} 'A{0} ) 'A{0} ) ) ( ( ( ( 'MP{0} ( ( 'arr{0} 'A{0} )
( ( 'arr{0} ( ( 'arr{0} 'A{0} ) 'A{0} ) ) 'A{0} ) ) ) ( ( 'arr{0}
( ( 'arr{0} 'A{0} ) ( ( 'arr{0} 'A{0} ) 'A{0} ) ) ) ( ( 'arr{0}
'A{0} ) 'A{0} ) ) ) ( ( ( 'S{0} 'A{0} ) ( ( 'arr{0} 'A{0} ) 'A{0}
) ) 'A{0} ) ) ( ( 'K{0} 'A{0} ) ( ( 'arr{0} 'A{0} ) 'A{0} ) ) ) )
( ( 'K{0} 'A{0} ) 'A{0} ) ) ) ) ) ) ) ) ) ) ) ) .
```

```
rew {{CONTEXT |- lemma-identity : {'G : 'o{0}}{'H : 'o{0}}{'A :
'o{0}}entails 'G{0} 'H{0} ('arr{0} 'A{0} 'A{0})}} .
```

```
rewrites: 30413 in 510ms cpu (520ms real) (59633 rewrites/second)
result Configuration: {{emptyJudgementList}}
```

```
op lemma-forget : -> Trm .
eq lemma-forget =
```

```
( [ 'G : 'o{0} ] ( [ 'H : 'o{0} ] ( [ 'A : 'o{0} ] ( [ 'M : 'o{0}
] ( [ 'm : ( ( ( entails 'G{0} ) 'H{0} ) 'M{0} ) ] ( [ 'P : (
'o{0} -> Prop ) ] ( [ 'g : ( 'P{0} 'G{0} ) ] ( [ 'h : ( 'P{0}
'H{0} ) ] ( [ 'K : ( { 'a : 'o{0} } ( { 'b : 'o{0} } ( 'P{0} ( ( k
'a{0} ) 'b{0} ) ) ) ) ] ( [ 'S : ( { 'a : 'o{0} } ( { 'b : 'o{0} }
( { 'c : 'o{0} } ( 'P{0} ( ( ( s 'a{0} ) 'b{0} ) 'c{0} ) ) ) ) ]
( [ 'MP : ( { 'a : 'o{0} } ( { 'b : 'o{0} } ( ( 'P{0} ( ( 'arr{0}
'a{0} ) 'b{0} ) ) -> ( ( 'P{0} 'a{0} ) -> ( 'P{0} 'b{0} ) ) ) ) )
] ( ( ( ( 'MP{0} 'M{0} ) ( ( 'arr{0} 'A{0} ) 'M{0} ) ) ( ( 'K{0}
'M{0} ) 'A{0} ) ) ( ( ( ( ( 'm{0} 'P{0} ) 'g{0} ) 'h{0} ) 'K{0}
'S{0} ) 'MP{0} ) ) ) ) ) ) ) ) ) ) ) .
```

```
rew {{CONTEXT |- lemma-forget : {'G : 'o{0}}{'H : 'o{0}}{'A :
'o{0}}{'M : 'o{0}}entails 'G{0} 'H{0} 'M{0} -> entails 'G{0} 'H{0}
('arr{0} 'A{0} 'M{0})}} .
```

```
rewrites: 78519 in 1230ms cpu (1240ms real) (63836 rewrites/second)
result Configuration: {{emptyJudgementList}}
```

```
op lemma-apply : -> Trm .
eq lemma-apply =
```

```
( [ 'G : 'o{0} ] ( [ 'H : 'o{0} ] ( [ 'A : 'o{0} ] ( [ 'M : 'o{0}
```



```

( { 'b'11 : 'o'0 } ( 'P'0 ( ( k 'a'10'0 ) 'b'11'0 ) ) ) ) ] (
[ '_ : ( { 'a'11 : 'o'0 } ( { 'b'12 : 'o'0 } ( { 'c : 'o'0 } (
'P'0 ( ( ( s 'a'11'0 ) 'b'12'0 ) 'c'0 ) ) ) ) ) ] ( [ '_ : ( {
'a'12 : 'o'0 } ( { 'b'13 : 'o'0 } ( ( 'P'0 ( ( 'arr'0 'a'12'0
) 'b'13'0 ) ) -> ( ( 'P'0 'a'12'0 ) -> ( 'P'0 'b'13'0 ) ) ) )
] ( ( 'K'0 'a'0 ) 'b'0 ) ) ) ) ) ) ) ) ( [ 'a : 'o'0 ]
( [ 'b : 'o'0 ] ( [ 'c : 'o'0 ] ( ( ( ( ( lemma-forget 'G'0 )
'G'0 ) 'A'0 ) ( ( ( s 'a'0 ) 'b'0 ) 'c'0 ) ) ( [ 'P : ( 'o'0
-> Prop ) ] ( [ '_ : ( 'P'0 'G'0 ) ] ( [ '_ : ( 'P'0 'G'0
) ] ( [ '_ : ( { 'a'11 : 'o'0 } ( { 'b'12 : 'o'0 } ( 'P'0 ( ( k
'a'11'0 ) 'b'12'0 ) ) ) ) ] ( [ 'S : ( { 'a'12 : 'o'0 } ( {
'b'13 : 'o'0 } ( { 'c'14 : 'o'0 } ( 'P'0 ( ( ( s 'a'12'0 )
'b'13'0 ) 'c'14'0 ) ) ) ) ) ] ( [ '_ : ( { 'a'13 : 'o'0 } ( {
'b'14 : 'o'0 } ( ( 'P'0 ( ( 'arr'0 'a'13'0 ) 'b'14'0 ) ) -> (
'P'0 'a'13'0 ) -> ( 'P'0 'b'14'0 ) ) ) ) ) ) ] ( ( ( 'S'0
'a'0 ) 'b'0 ) 'c'0 ) ) ) ) ) ) ) ) ) ) ) ( ( ( lemma-apply
'G'0 ) 'G'0 ) 'A'0 ) ) ) ) ) .

rew {{CONTEXT |- dedThm : {'G : 'o'0}}{'A : 'o'0}}{'M :
'o'0}}entails 'G'0 'A'0 'M'0 -> entails 'G'0 'G'0 ('arr'0
'A'0 'M'0)}} .

rewrites: 506023 in 15750ms cpu (16540ms real) (32128 rewrites/second)
result Configuration: {{emptyJudgementList}}

endm

```

Appendix B

Specifying a Programming Language for Active Networks

In this appendix we illustrate the use of the CINNI calculus of names and substitutions, that we developed in Chapter 5, in the context of an application in the area of active networks. *Active networks* (see [TSS⁺97] for a survey of active network research) are networks with nodes that do not operate according to a fixed scheme (e.g. as conventional routers) but are instead fully programmable and provide *execution environments* for programs that can be received from other other nodes via the network. An active network does not necessarily have a static topology, so that also unreliable, wireless and hybrid networks are subsumed by this notion. One may think of active networks as a generalization of conventional networks and as a step toward greater flexibility: Packets, which are *interpreted* by routers in conventional networks following rigid schemes, become programs, which are *executed* in active networks in a universal fashion. One of the central issues of active network research is the development of programming paradigms which provide a suitable level of abstraction to develop active network applications. A particular programming language providing such a new paradigm is *PLAN*, the *Packet Language for Active Networks* [HKM⁺98b, HKM⁺98a, MHN99, HK99, KHMG99], which has been implemented as part of PLANet, an *active internetwork architecture* [HMA⁺99]. PLAN is an imperative functional language similar to ML, but has a number of unique additional features, such as remote function execution, resource management and anytime typechecking. Remote function execution, means that functions can be applied to arguments so that the execution does not take place locally but in the execution environment of a different network node. To this end, the function call is treated as a so-called chunk, i.e. as a piece of data, which is transmitted to the destination node by means of a packet. Resource management refers to a mechanism which keeps track of computational resources and ensures that all PLAN programs are terminating.

B.1 The Executable Specification in Maude

In the following, we give an executable formal specification of PLAN. The specification is a boiled-down version of [ST01] which has been developed in a collaborative effort together with Carolyn Talcott (Stanford University), Jose Meseguer (SRI) and the PLAN group, in particular Carl A. Gunter and Pankaj Kakkar (University of Pennsylvania) in the scope of the DARPA Active Network Program [DAR] and the ONR Coordinated Research in Adaptive Network Centric Computing [ONR]. In accordance with a research objective which has been first set in [WMG00] our work provides a formal semantics for PLAN. Indeed our specification fully captures the intent of the specifications [Kak99] and [KGA00], but has the benefit of being both rigorously formal and executable. As shown in [ST01, MÖST02, ST02a] our specification can be used as an active network model with execution environments for PLAN programs. More generally, a formal executable specification can be used at very different levels [GDMT00] ranging from formal modeling and execution via model checking and narrowing analysis to formal verification. For instance, a form of abstract execution has been used in [ST02a] as one step in the verification of a concrete PLAN program.

By exploiting the capabilities of rewriting logic, the Maude specification enhances the original PLAN specification in several aspects. For instance, the Maude specification directly reflects the concurrent nature of the network, meaning that each node makes only use of local information about the network topology and that the possibility of concurrent execution in different nodes is directly reflected in the specification.

In fact, a particularly interesting aspect of our problem is the presence of *multi-level concurrency*: (1) A network configuration is modeled as a multiset containing nodes and packets. (2) With each node we associate a multiset of processes local to the node, which serve as execution environments for PLAN programs and can themselves execute concurrently within the node. (3) Each process encapsulates the local state of the execution environment together with an abstract reduction machine for the PLAN language. More specifically, there are three kinds of rewrite rules used in the specification, corresponding to the following actions: (1) A process in a node can emit a packet containing a PLAN program for remote execution; (2) A process can call a service function which has access to the local process state and possibly modifies it, i.e. causes a side effect; (3) The abstract machine, which implements the functional part of the PLAN language, can change its state.

To specify the abstract machine we use a general approach suitable for functional languages with side-effects which is inspired by [FF86, HMST95, MT01]. The main idea is that the reduction state of the abstract machine is a pair, consisting of a reduction context (i.e. an expression with a hole, i.e. a metavariable) and the expression to be reduced in this context. Furthermore, the specification uses the

CINNI calculus to specify the binding structure of the language and to formalize environments as explicit substitutions. A particular benefit of this approach is that environments can always be eliminated, resulting in a specification that does not have to deal with them explicitly at various places. Especially the delicate issue of binding and environment handling in the context of recursive remote function calls turns out to have a very elegant solution. In summary, the use of the abstract reduction machine in connection with CINNI has greatly increased the clarity and reduced the complexity of the specification w.r.t. earlier versions [Ste00b, Tal00, Tal01], bringing us closer to the goal of a specification which is useful for application programmers, language designers/implementors and formal methods researchers.

Our specification uses a representation of programs based on an imperative λ -calculus and presupposes that PLAN programs are mapped to this uniform representation by a simple preprocessing stage. For instance, in PLAN programs recursive references to functions can only occur inside the chunks which are arguments to `Eval`, `OnNeighbor`, or `OnRemote`; otherwise the termination property proved in [ST02b] would be compromised. To focus on the essential aspects we have simplified the specification, by removing type annotations (which are only used for type checking), the `OnRemote` construct (which generalizes the `OnNeighbor` construct), and some other features such as the routing table information which is stored in the nodes. This simplified specification is given below, and a possible execution scenario is specified subsequently. For the full specification and a more detailed presentation we refer to [ST01].

```

mod PLAN is

protecting BOOL .
protecting MACHINE-INT .

-----
--- Network Sorts

sort Loc .    --- physical locations (i.e. hosts)
sort Addr .  --- addresses (i.e. network interfaces)

sort LocList .
subsort Loc < LocList .
op empty-ll : -> LocList .
op _',_ : LocList LocList -> LocList
        [ assoc id: empty-ll ] .

sort AddrList .
subsort Addr < AddrList .
op empty-al : -> AddrList .

```

```

op _',_ : AddrList AddrList -> AddrList
    [ assoc id: empty-al ] .

sort Connection .
op _>>_ : Addr Addr -> Connection . --- network connection

sort ConnectionList .
subsort Connection < ConnectionList .
op empty-conl : -> ConnectionList .
op _',_ : ConnectionList ConnectionList -> ConnectionList
    [ assoc id: empty-conl ] .

var l l' : Loc .
var ll : LocList .
var a a' : Addr .
var al al' : AddrList .
var con con' : Connection .
var from to : Addr .
var conl conl' : ConnectionList .

op contains : AddrList Addr -> Bool .
eq contains(empty-al,a') = false .
ceq contains((a,al),a') = true if a == a' .
ceq contains((a,al),a') = contains(al,a') if a /= a' .

op remove : AddrList Addr -> AddrList .
eq remove(empty-al,a') = empty-al .
ceq remove((a,al),a') = remove(al,a') if a == a' .
ceq remove((a,al),a') = (a,remove(al,a')) if a /= a' .

op contains : ConnectionList Connection -> Bool .
eq contains(empty-conl,con') = false .
ceq contains((con,conl),con') = true if con == con' .
ceq contains((con,conl),con') = contains(conl,con') if con /= con' .

op contains : ConnectionList Addr -> Bool .
eq contains(empty-conl,a') = false .
ceq contains(((from >> to),conl),a') = true if to == a' .
ceq contains(((from >> to),conl),a') = contains(conl,a') if to /= a' .

op devtohost : ConnectionList Addr -> Addr .
ceq devtohost(((from >> to),conl),a') = from if to == a' .
ceq devtohost(((from >> to),conl),a') = devtohost(conl,a') if to /= a' .

```

```

-----
--- Identifiers

sort Id .

sort IdList .
subsort Id < IdList .
op empty-idl : -> IdList .
op _',_ : IdList IdList -> IdList [ assoc id: empty-idl ] .

var id id' id'' : Id .
var idl idl' idl'' : IdList .

-----
--- Indexed Variables

sorts Nat .

op 0 : -> Nat .
op suc : Nat -> Nat .

sort Var .

op _{ } : Id Nat -> Var [ prec 0 gather (E &) ] .

-----
--- Constants

sort Cstr NonCstr Const .
subsort Cstr < Const .
subsort NonCstr < Const .

op Dummy : -> Cstr .
op Equal : -> NonCstr .
ops Not And Or : -> NonCstr .
ops Plus Minus Mul Div : -> NonCstr .
op Pair : -> Cstr .
ops Fst Snd : -> NonCstr .
ops Cons Nil : -> Cstr .
ops Hd Tl : -> NonCstr .
ops Length Member Append Reverse ConsFirst : -> NonCstr .
ops Foldr Foldl : -> NonCstr .
op Chunk : -> Cstr .

op Print : -> NonCstr .

```

```

ops ThisHost ThisHostIs : -> NonCstr .
ops GetRB GetSrc GetSrcDev : -> NonCstr .
ops GetNeighbors GetDevToHost : -> NonCstr .
ops OnNeighbor Eval : -> NonCstr .

op Bool_ : Bool -> Const .
op Int_ : MachineInt -> Const .
op Addr_ : Addr -> Const .

var bool bool' : Bool .
var const : Const .
var var var' var'' : Var .
var cstr : Cstr .
var ncstr : NonCstr .

-----
--- Expressions

sort Ex .
subsort Const < Ex .
subsort Var < Ex .

op __ : Ex ExList -> Ex [ prec 21 gather (E e) ] .
op _;_ : Ex Ex -> Ex [ prec 30 gather (e E) ] .
op If_Then_Else_ : Ex Ex Ex -> Ex .
op Let'[_=_']_ : IdList ExList Ex -> Ex [ prec 30 gather (& & E) ] .
op LetRec'[_=_']_ : IdList ExList Ex -> Ex [ prec 30 gather (& & E) ] .
op Lam'[_]_ : IdList Ex -> Ex [ prec 30 gather (& E) ] .

var ex ex' ex'' : Ex .
var exl exl' exl'' : ExList .

-----
--- Contexts (Expressions with Holes i.e. Metavariables)

sort Cx .
subsort Ex < Cx .

op ? : -> Cx .
op __ : Cx CxList -> Cx [ prec 21 gather (E e) ] .
op _;_ : Cx Cx -> Cx [ prec 30 gather (e E) ] .
op If_Then_Else_ : Cx Cx Cx -> Cx .
op Let'[_=_']_ : IdList CxList Cx -> Cx [ prec 30 gather (& & E) ] .
op LetRec'[_=_']_ : IdList CxList Cx -> Cx [ prec 30 gather (& & E) ] .
op Lam'[_]_ : IdList Cx -> Cx [ prec 30 gather (& E) ] .

```

```

var cx cx' cx'' cx''' : Cx .
var cxl cxl' cxl'' cxl''' : CxList .

-----
--- Lists of Contexts and Expressions

sort EmptyExList Vallist ExList CxList .

op empty-exl : -> EmptyExList .
subsort EmptyExList < ExList < CxList .
subsort EmptyExList < ExList < CxList .

op _',_ : EmptyExList EmptyExList -> EmptyExList
      [ assoc id: empty-exl ] .

subsort Ex < ExList .
op _',_ : ExList ExList -> ExList
      [ assoc id: empty-exl ] .

subsort Cx < CxList .
op _',_ : CxList CxList -> CxList
      [ assoc id: empty-exl ] .

-----
--- Values and NonValues

sort Val NonVal .
subsort Val < Ex .
subsort NonVal < Ex .

subsort Const < Val .

subsort EmptyExList < Vallist < ExList .
subsort EmptyExList < ExList .

subsort Val < Vallist .
op _',_ : Vallist Vallist -> Vallist
      [ assoc id: empty-exl ] .

var nval nval' nval'' : NonVal .
var val val' val'' : Val .
var val0 val0' val0'' : Val .
var val1 val1' val1'' : Val .
var vall vall' vall'' : Vallist .

```

```

mb (Bool bool) : Val .
mb (Int int) : Val .
mb (Lam [idl] ex) : Val .
mb (cstr vall) : Val .

mb (val (ex1,nval,ex1')) : NonVal .
mb (nval ex1) : NonVal .
mb (ncstr ex1) : NonVal .
mb ((Lam [idl] ex) ex1) : NonVal .
mb (ex ; ex) : NonVal .
mb (If ex Then ex' Else ex'') : NonVal .
mb (Let [idl = ex1] ex') : NonVal .
mb (LetRec [idl = ex1] ex') : NonVal .

-----
--- Object-Variable Substitution
--- (an instance of the CINNI substitution calculus
--- generalized to simultaneous substitutions)

sort Subst .

op [_:=_] : Id Ex -> Subst [ prec 30 gather (& &) ] .
op [_:=_] : IdList ExList -> Subst [ prec 30 gather (& &) ] .
op [shift_] : Id -> Subst [ prec 30 gather (&) ] .
op [lift__] : Id Subst -> Subst [ prec 30 gather (& &) ] .
op [lift__] : IdList Subst -> Subst [ prec 30 gather (& &) ] .
op __ : Subst Ex -> Ex [ prec 30 gather (E E) ] .
op __ : Subst ExList -> ExList [ prec 30 gather (E E) ] .

var S : Subst .
vars m : Nat .

eq ([id := ex] (id{0})) = ex .
eq ([id := ex] (id{suc(m)})) = (id{m}) .
ceq ([id := ex] (id'{m})) = (id'{m}) if id /= id' .

eq ([shift id] (id{m})) = (id{suc(m)}) .
ceq ([shift id] (id'{m})) = (id'{m}) if id /= id' .

eq ([lift id S] (id{0})) = (id{0}) .
eq ([lift id S] (id{suc(m)})) = [shift id] (S (id{m})) .
ceq ([lift id S] (id'{m})) = [shift id] (S (id'{m})) if id /= id' .

```

```

eq ([empty-idl := empty-exl] ex') = ex' .
ceq ([id,idl] := (ex,exl)] ex') =
  ([id := ex][idl := [shift id] exl] ex') if idl /= empty-idl .

eq ([lift empty-idl S] ex') = (S ex') .
ceq ([lift (id,idl) S] ex') =
  ([lift idl [lift id S]] ex') if idl /= empty-idl .

eq (S const) = const .
eq (S (ex exl')) = ((S ex) (S exl')) .
eq (S (ex ; ex')) = ((S ex) ; (S ex')) .
eq (S (If ex Then ex' Else ex'')) =
  (If (S ex) Then (S ex') Else (S ex'')) .
eq (S (Let [idl = exl] ex')) =
  (Let [idl = (S exl)] ([lift idl S] ex')) .
eq (S (LetRec [idl = exl] ex')) =
  (LetRec [idl = ([lift idl S] exl)] ([lift idl S] ex')) .
eq (S (Lam [idl] ex)) =
  (Lam [idl] ([lift idl S] ex)) .

eq (S empty-exl) = empty-exl .
ceq (S (ex , exl')) = ((S ex) , (S exl')) if exl' /= empty-exl .

```

```

--- Hole (Meta-Variable) Substitution
--- (standard textual substitution)

```

```

op <'?:= >_ : Cx Cx -> Cx [ prec 10 ] .
op <'?:= >_ : Cx CxList -> CxList [ prec 10 ] .

eq < ? := cx > ? = cx .
eq < ? := cx > const = const .
eq < ? := cx > var = var .
eq < ? := cx > (cx' cxl'') =
  ((< ? := cx > cx') (< ? := cx > cxl'')) .
eq < ? := cx > (cx' ; cx'') =
  ((< ? := cx > cx') ; (< ? := cx > cx'')) .
eq < ? := cx > (If cx' Then cx'' Else cx''') =
  (If (< ? := cx > cx')
    Then (< ? := cx > cx'') Else (< ? := cx > cx''')) .
eq < ? := cx > (Let [idl = cxl'] cx'') =
  (Let [idl = < ? := cx > cxl'] (< ? := cx > cx'')) .
eq < ? := cx > (LetRec [idl = cxl'] cx'') =
  (LetRec [idl = < ? := cx > cxl'] (< ? := cx > cx'')) .

```

```

eq < ? := cx > (Lam [idl] cx') =
  (Lam [idl] (< ? := cx > cx')) .
eq < ? := cx > empty-exl = empty-exl .
ceq < ? := cx > (cx' , cxl'') =
  ((< ? := cx > cx') , (< ? := cx > cxl'')) if cxl'' /= empty-exl .

```

```

-----
--- Reduction Machine State

```

```

sort RedState .

```

```

op RedState : Cx Ex -> RedState .

```

```

var int int' : MachineInt .

```

```

var rb : MachineInt . --- resource bound

```

```

var devs : AddrList . --- network devices of a node

```

```

var nbrs : ConnectionList . --- neighbors of a node

```

```

var dest : Addr . --- destination address of a packet

```

```

var src : Addr . --- source address of a program/packet

```

```

var ariv : Addr . --- arival device of a program

```

```

-----
--- Reduction Machine Rules

```

```

--- control (return to top)

```

```

crl RedState(cx, val) =>
  RedState(?, < ? := val > cx) if cx /= ? .

```

```

-----
--- let statements

```

```

--- control (evaluate from left to right)

```

```

rl RedState(cx, Let [idl = (vall', nval', exl')] ex'') =>
  RedState(< ? := Let [idl = (vall', ?, exl')] ex'' > cx, nval') .

```

```

--- reduction

```

```

rl RedState(cx, Let [idl = vall'] ex'') =>
  RedState(cx, [idl := vall'] ex'') .

```

```

-----
--- letrec statements

--- control (evaluate from left to right)

r1 RedState(cx, LetRec [idl = (vall', nval', exl')] ex'') =>
   RedState(< ? := LetRec [idl = (vall', ?, exl')] ex'' > cx, nval') .

--- reduction

op LetRec'[_=_']_ : IdList ExList ExList -> Ex
   [ prec 30 gather (& & E) ] .
eq (LetRec[idl = exl] empty-exl) = empty-exl .
ceq (LetRec[idl = exl] (ex',exl')) =
   ((LetRec[idl = exl] ex'),(LetRec[idl = exl] exl'))
   if exl' /= empty-exl .

r1 RedState(cx, LetRec [idl = vall'] ex'') =>
   RedState(cx, [idl := (LetRec [idl = vall'] vall')] ex'') .

-----
--- function application

--- control (evaluate arguments from left to right)

r1 RedState(cx, (nval exl')) =>
   RedState(< ? := (? exl') > cx, nval) .

r1 RedState(cx, (val (vall', nval', exl'))) =>
   RedState(< ? := (val (vall', ?, exl')) > cx, nval') .

--- reduction: beta

r1 RedState(cx, ((Lam [idl] ex) vall)) =>
   RedState(cx, [idl := vall] ex) .

--- reduction: equality

r1 RedState(cx, (Equal (val,val'))) =>
   RedState(cx, (Bool (val == val'))) .

--- reduction: boolean ops

r1 RedState(cx, (Not (Bool bool))) =>
   RedState(cx, (Bool (not bool))) .

```

```
rl RedState(cx, (And ((Bool bool),(Bool bool')))) =>
  RedState(cx, (Bool (bool and bool'))) .
```

```
rl RedState(cx, (Or ((Bool bool),(Bool bool')))) =>
  RedState(cx, (Bool (bool or bool'))) .
```

--- reduction: integer ops

```
rl RedState(cx, (Plus ((Int int),(Int int')))) =>
  RedState(cx, (Int (int + int'))) .
```

```
rl RedState(cx, (Minus ((Int int),(Int int')))) =>
  RedState(cx, (Int (int - int'))) .
```

```
rl RedState(cx, (Mul ((Int int),(Int int')))) =>
  RedState(cx, (Int (int * int'))) .
```

```
rl RedState(cx, (Div ((Int int),(Int int')))) =>
  RedState(cx, (Int (int / int'))) .
```

--- reduction: pair ops

```
rl RedState(cx, (Fst (Pair (val,val')))) =>
  RedState(cx, val) .
```

```
rl RedState(cx, (Snd (Pair (val,val')))) =>
  RedState(cx, val') .
```

--- reduction: list ops

```
rl RedState(cx, (Hd (Cons (val,val')))) =>
  RedState(cx, val) .
```

```
rl RedState(cx, (Tl (Cons (val,val')))) =>
  RedState(cx, val') .
```

```
rl RedState(cx, (Length Nil)) =>
  RedState(cx, (Int 0)) .
```

```
rl RedState(cx, (Length (Cons (val',val'')))) =>
  RedState(cx, (Plus ((Int 1),(Length val'')))) .
```

```
rl RedState(cx, (Member (val, Nil))) =>
  RedState(cx, (Bool false)) .
```

```

cr1 RedState(cx, (Member (val, (Cons (val',val''))))) =>
  RedState(cx, (Bool true))
  if val == val' .

cr1 RedState(cx, (Member (val, (Cons (val',val''))))) =>
  RedState(cx, (Member (val, val'')))
  if val /= val' .

r1 RedState(cx, (Append (Nil, val))) =>
  RedState(cx, val) .

r1 RedState(cx, (Append (Cons (val,val'),val''))) =>
  RedState(cx, (Cons (val, (Append (val',val''))))) .

r1 RedState(cx, (Reverse Nil)) =>
  RedState(cx, Nil) .

r1 RedState(cx, (Reverse (Cons (val,val')))) =>
  RedState(cx, (Append ((Reverse val'),(Cons (val,Nil))))) .

r1 RedState(cx, (Foldr (val,Nil,val''))) =>
  RedState(cx, val'') .

r1 RedState(cx, (Foldr (val,(Cons (val0',val1')),val''))) =>
  RedState(cx, (val (val0', (Foldr (val,val1',val''))))) .

r1 RedState(cx, (Foldl (val,val',Nil))) =>
  RedState(cx, val') .

r1 RedState(cx, (Foldl (val,val',(Cons (val0'',val1''))))) =>
  RedState(cx, (Foldl (val,(val (val',val0''),val1'')))) .

-----
--- argument lists

--- control (evaluate from left to right)

r1 RedState(cx, (nval , ex')) =>
  RedState(< ? := ( ? , ex')> cx, nval) .

r1 RedState(cx, (val , nval)) =>
  RedState(< ? := (val , ?)> cx, nval) .

```

```

-----
--- if-then-else statement

--- control (evaluate condition)

rl RedState(cx, (If nval Then ex' Else ex'')) =>
   RedState(< ? := (If ? Then ex' Else ex'') > cx, nval) .

--- reduction

rl RedState(cx, (If (Bool true) Then ex' Else ex'')) =>
   RedState(cx, ex') .

rl RedState(cx, (If (Bool false) Then ex' Else ex'')) =>
   RedState(cx, ex') .

-----
--- sequential evaluation

--- control (evaluate on the left)

rl RedState(cx, (nval ; ex')) =>
   RedState(< ? := (? ; ex') > cx, nval) .

--- reduction

rl RedState(cx, (val ; ex')) =>
   RedState(cx, ex') .

-----
--- processes, packets and configurations

sort Node . --- network node
op Node : Loc AddrList ConnectionList -> Node .

--- contains: physical location (Loc),
---             list of network devices (AddrList),
---             list of connections to neighbors (ConnectionList)

sort Process . --- process being executed on a node
op Process : Loc Addr Addr MachineInt RedState -> Process .

--- contains: physical location (Loc),
---             source address (Addr),
---             arrival device of program (Addr),

```

```

---          resource bound (MachineInt),
---          reduction machine state (RedState)

sort Packet . --- packet in transit between nodes
op Packet : Addr Addr MachineInt Ex ExList -> Packet .

--- contains: destination address (Addr),
---          source address (Addr),
---          resource bound (MachineInt),
---          chunk, i.e. function (Ex) with arguments (ExList)

sort Configuration .
subsort Node < Configuration .
subsort Packet < Configuration .
subsort Process < Configuration .
op empty-conf : -> Configuration .
op __ : Configuration Configuration -> Configuration
      [assoc comm id: empty-conf] .

-----
--- local execution of chunks

crl Process(l, src, ariv, rb,
           RedState(cx, (Eval (Chunk (val,vall)))))) =>
    Process(l, src, ariv, (rb - 1),
           RedState(cx, (val vall)))
    if rb > 0 .

-----
--- remote execution of chunks (emitting packet)

var dev : Addr . --- outgoing network device

crl Node(l,devs,nbrs)
    Process(l, src, ariv, rb,
           RedState(cx, (OnNeighbor ((Chunk (val,vall)),
                                     (Addr dest),(Int int),(Addr dev)))))) =>
    Node(l,devs,nbrs)
    Process(l, src, ariv, (rb - int),
           RedState(cx, Dummy))
    Packet(dest, src, (int - 1), val, vall)
    if contains(devs,dev) and contains(nbrs,(dev >> dest)) and
       (rb >= int) and (int > 0) .

```

```

-----
--- receiving packet and starting process

crl Node(1,devs,nbrs)
  Packet(dest, src, rb, val, vall) =>
  Node(1,devs,nbrs)
  Process(1, src, dest, rb, RedState(?,(val vall)))
  if contains(devs,dest) .

```

```

-----
--- service: GetRB (get resource bound)

rl Process(1, src, ariv, rb,
  RedState(cx, (GetRB empty-exl))) =>
  Process(1, src, ariv, rb,
  RedState(cx, (Int rb))) .

```

```

-----
--- service: GetSrc (get source address)

rl Process(1, src, ariv, rb,
  RedState(cx, (GetSrc empty-exl))) =>
  Process(1, src, ariv, rb,
  RedState(cx, (Addr src))) .

```

```

-----
--- service: GetSrcDev (get arrival device)

rl Process(1, src, ariv, rb,
  RedState(cx, (GetSrcDev empty-exl))) =>
  Process(1, src, ariv, rb,
  RedState(cx, (Addr ariv))) .

```

```

-----
--- service: ThisHostIs

rl Node(1,devs,nbrs)
  Process(1, src, ariv, rb,
  RedState(cx, (ThisHostIs (Addr a)))) =>
  Node(1,devs,nbrs)
  Process(1, src, ariv, rb,
  RedState(cx, (Bool (contains(devs,a)))) .

```

```

-----
--- service: ThisHost

op cast : AddrList -> Ex .
eq cast(empty-al) = Nil .
eq cast(a,al) = (Cons ((Addr a),cast(al))) .

rl Node(l,devs,nbrs)
  Process(l, src, ariv, rb,
    RedState(cx, (ThisHost empty-exl))) =>
  Node(l,devs,nbrs)
  Process(l, src, ariv, rb,
    RedState(cx, cast(devs))) .

-----
--- service: GetNeighbors

op cast : Connection -> Ex .
eq cast(a >> a') = (Pair ((Addr a'),(Addr a))) .

op castl : ConnectionList -> Ex .
eq castl(empty-conl) = Nil .
eq castl((a >> a'),conl) = (Cons (cast(a >> a'),castl(conl))) .

rl Node(l,devs,nbrs)
  Process(l, src, ariv, rb,
    RedState(cx, (GetNeighbors empty-exl))) =>
  Node(l,devs,nbrs)
  Process(l, src, ariv, rb,
    RedState(cx, castl(nbrs))) .

-----
--- service: GetDevToHost

crl Node(l,devs,nbrs)
  Process(l, src, ariv, rb,
    RedState(cx, (GetDevToHost (Addr a)))) =>
  Node(l,devs,nbrs)
  Process(l, src, ariv, rb,
    RedState(cx, (Addr devtohost(nbrs,a))))
  if contains(nbrs,a) .

endm

```

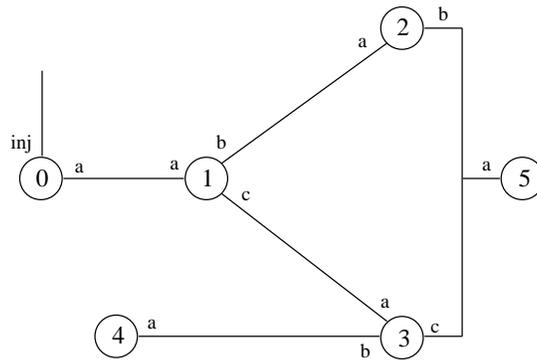


Figure B.1: Example Topology

We illustrate the use of this specification using a PLAN program, which is almost literally taken from [KHM99]. We execute this program in the network topology depicted in Fig. B.1, which consists of six nodes participating in six networks and is given by the specification below. The program is injected into node 10 via the address `inj`. The program's objective is to find all routes to a given destination, which is `4a` in the example. It makes use of a function `find`, which recursively invokes itself on all neighbors not visited so far.

```

mod EXAMPLE is

including PLAN .

--- small example topology with 6 nodes (10,...,15)

ops 10 11 12 13 14 15 : -> Loc .
ops external inj 0a 1a 1b 1c 2a 2b 3a 3b 3c 4a 5a : -> Addr .

op example-topology : -> Configuration .
eq example-topology =
  Node(10, (inj,0a), ((0a >> 1a)))
  Node(11, (1a,1b,1c), ((1a >> 0a),(1b >> 2a),(1c >> 3a)))
  Node(12, (2a,2b), ((2a >> 1b),(2b >> 3c),(2b >> 5a)))
  Node(13, (3a,3b,3c), ((3a >> 1c),(3b >> 4a),(3c >> 2b),(3c >> 5a)))
  Node(15, (5a), ((5a >> 2b),(5a >> 3c)))
  Node(14, (4a), ((4a >> 3b))) .

--- example program

ops consfirst : -> Id .
ops goback nexthop remaining route find visited : -> Id .
ops desti allneighbors newneighbors addifnew sendchild : -> Id .
ops neigh neighl dum childrb myaddrs : -> Id .

```

```

op example-program : -> Ex .
eq example-program =
  (LetRec [goback = Lam [remaining,route]
  (If (Equal (remaining{0}, Nil))
    Then (Print route{0})
    Else (Let [nexthop = (Snd (Hd remaining{0}))]
      (OnNeighbor
        ((Chunk (goback{0},((Tl remaining{0}),route{0}))),
          nexthop{0}, (GetRB empty-exl),
          (GetDevToHost nexthop{0}))))))])
  (LetRec [find = Lam [desti,route,visited]
  (Let [myaddrs =
    (ThisHost empty-exl)]
  (Let [allneighbors =
    (GetNeighbors empty-exl)]
  (Let [addifnew = Lam [neigh,neighl]
    (If (Not (Member ((Fst neigh{0}),visited{0})))
      Then (Cons (neigh{0},neighl{0}))
      Else neighl{0})])
  (Let [newneighbors =
    (Foldr (addifnew{0},allneighbors{0},Nil))]
  (Let [childrb =
    (If (Equal ((Length newneighbors{0}), (Int 0)))
      Then (GetRB empty-exl)
      Else (Div ((GetRB empty-exl), (Length newneighbors{0}))))])
  (Let [sendchild = Lam [neigh,dum]
    (Let [route =
      (Cons ((Pair ((GetSrcDev empty-exl), (Snd neigh{0}))),
        route{0}))])
    (OnNeighbor
      ((Chunk (find{0},(desti{0},route{0},
        (Append (myaddrs{0},visited{0}))))),
        (Fst neigh{0}), childrb{0}, (Snd neigh{0}))))])
  (If (ThisHostIs desti{0})
    Then (goback{0} (route{0},(Reverse route{0})))
    Else (Foldr (sendchild{0},newneighbors{0},Dummy)))))))]
  (find{0} ((Addr 4a),Nil,Nil))) .

endm

```

A sample execution of this PLAN program is given below. The final configuration computed by Maude shows that three different routes from node 10 to 14 have been found and reported back to node 10, where they are printed.

```

rew example-topology
    Process(10, external, inj, 1000,
        RedState(?, example-program)) .

rewrites: 1574214 in 16620ms cpu (16630ms real) (94718 rewrites/second)
result Configuration:
Node(10, (inj,0a), ((0a >> 1a)))
Node(11, (1a,1b,1c), ((1a >> 0a),(1b >> 2a),1c >> 3a))
Node(12, (2a,2b), ((2a >> 1b),(2b >> 3c),(2b >> 5a)))
Node(13, (3a,3b,3c), ((3a >> 1c),(3b >> 4a),(3c >> 2b),(3c >> 5a)))
Node(14, (4a), ((4a >> 3b)))
Node(15, (5a), ((5a >> 2b),(5a >> 3c)))
Process(10, external, inj, 0, RedState(?, Dummy))
Process(10, external, 0a, 119,
    RedState(?, Print (Cons (Pair (Addr inj,Addr 0a),
        Cons (Pair (Addr 1a,Addr 1b),
            Cons (Pair (Addr 2a,Addr 2b),
                Cons (Pair (Addr 3c,Addr 3b),Nil)))))))
Process(10, external, 0a, 162,
    RedState(?, Print (Cons (Pair (Addr inj,Addr 0a),
        Cons (Pair (Addr 1a,Addr 1c),
            Cons (Pair (Addr 3a,Addr 3b),Nil)))))))
Process(10, external, 0a, 241,
    RedState(?, Print (Cons (Pair (Addr inj,Addr 0a),
        Cons (Pair (Addr 1a,Addr 1b),
            Cons (Pair (Addr 2a,Addr 2b),
                Cons (Pair (Addr 5a,Addr 5a),
                    Cons (Pair (Addr 3c,Addr 3b),Nil))))))))))
Process(11, external, 1a, 1, RedState(?, Dummy))
Process(11, external, 1b, 0, RedState(?, Dummy))
...

```

For a more interesting example, which is based on our full specification [ST01] and especially employs the resident data services of PLAN, we refer to [MÖST02, ST02a], where temporal logic model checking and mathematical reasoning is used for its analysis. Beyond the analysis of individual programs, our formal specification can be used to prove very general termination properties for the entire class of PLAN programs under various extensions of the language and the active network architecture [ST02b].

Bibliography

- [ABN⁺95] F. Andersen, U. Binau, K. Nyblad, K. D. Petersen, and J. S. Pettersson. The HOL-UNITY verification system. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22–26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 795–796. Springer-Verlag, 1995.
- [AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.
- [ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
- [Acz78] P. Aczel. A general church-rosser theorem. Unpublished Manuscript, University of Manchester, July 1978.
- [AKMP96] N. A. Anisimov, K. Kishinski, A. Miloslavski, and P. A. Postupalski. Macroplaces in high level Petri nets: Application for design inbound call center. In *Proceedings of the International Conference on Information System Analysis and Synthesis (ISAS'96), Orlando, FL, USA*, pages 153–160, July 1996.
- [All87] S. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, September 1987.
- [Alt93] T. Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference, TLCA'93, Utrecht, The Netherlands, March 16–18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
- [Ani91] N. A. Anisimov. An algebra of regular macronets for formal specification of communication protocols. *Computers and Artificial Intelligence*, 10(6):541–560, 1991.

- [AO97] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Graduate Texts in Computer Science. Springer-Verlag, 1997.
- [APP94] F. Andersen, K. D. Petersen, and J. S. Pettersson. Program verification using HOL-UNITY. In J. J. Joyce and C.-J. H. Segar, editors, *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93, Vancouver, B.C., August 11-13 1993, Proceedings*, volume 780 of *Lecture Notes in Computer Science*, pages 1-15. Springer-Verlag, 1994.
- [AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181-185, 1985.
- [AS86] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1986.
- [Asp87] A. Asperti. A logic for concurrency. Unpublished Manuscript, November 1987.
- [Aug] L. Augustsson. Cayenne - a language with dependent types. Department of Computer Science, Chalmers University of Technology, <http://www.cs.chalmers.se/~augustss/cayenne/>.
- [Aug99] L. Augustsson. Cayenne - a language with dependent types. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Advanced Functional Programming, Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*, volume 1608 of *Lecture Notes in Computer Science*, pages 240-267. Springer-Verlag, 1999.
- [Bar84] H. P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. Elsevier, 1984.
- [Bar92] H. P. Barendregt. Lambda-calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*. Clarendon Press, Oxford, 1992.
- [BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed λ -programs on term algebras. *Theoretical Computer Science*, 39, 1985.
- [BBC+99] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J. C. Filliatre, E. Giménez, H. Herbelin, G. Huet, H. Lauthère, C. Muñoz, C. Murthy, C. Parent-Vigouroux,

- P. Loiseleur, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistent Reference Manual, Version 6.3.1, Coq Project. Technical report, INRIA, 1999. <http://logical.inria.fr/>.
- [BBLRD96] Z. Benaïssa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, September 1996.
- [BCC⁺86] B. Berthomieu, N. Choquet, C. Colin, B. Loyer, J. M. Martin, and A. Mauboussin. Abstract data nets: Combining Petri nets and abstract data types for high level specifications of distributed systems. In *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets, Oxford, UK, June 1986*, pages 25–48, 1986.
- [BCD⁺98] P. Borovansky, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P. E. Moreau, C. Ringeissen, and M. Vittek. ELAN v 3.3 user manual. Technical report, INRIA Lorraine & LORIA, Nancy (France), December 1998.
- [BCM88] E. Battiston, F. De Cindio, and G. Mauri. OBJSA nets: A class of high-level nets having objects as domains. In G. Rozenberg, editor, *Advances in Petri Nets 1988, covers the 8th European Workshop on Application and Theory of Petri Nets, Zaragoza, Spain, June 1987*, volume 340 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.
- [BCM99] D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. In *LFM'99: Workshop on Logical Frameworks and Metalanguages, Paris, France, September 28, 1999, Proceedings*, 1999. <http://plan9.bell-labs.com/who/felty/LFM99/>.
- [BCM00] D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. In S. Kapoor and S. Prasad, editors, *Twentieth Conference on the Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, December 13–15, 2000, Proceedings*, volume 1974 of *Lecture Notes in Computer Science*, pages 55–80. Springer-Verlag, 2000.
- [BD87] E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55:87–136, 1987.
- [BD89] L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. *Theoretical Computer Science*, 67(2/3):173–201, 1989.

- [Bee88] M. J. Beeson. Towards a computation system based on set theory. *Theoretical Computer Science*, 60:297–340, 1988.
- [Ber76] K. J. Berklings. A symmetric complement to the lambda-calculus. Interner Bericht ISF-76-7, GMD, St. Augustin, Germany, 1976.
- [Ber88] S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and other systems in Barendregt’s cube. Technical report, Carnegie Mellon University and Università di Torino, 1988.
- [BF82] K. J. Berklings and E. Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55:89–101, 1982.
- [BF88] E. Best and C. Fernández. *Nonsequential Processes – A Petri Net View*, volume 13 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.
- [BFG⁺93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993. <http://www4.informatik.tu-muenchen.de/proj/korso/papers/v10.html>.
- [BG77] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proceedings 5th International Joint Conference on Artificial Intelligence*, pages 1045–1058, 1977.
- [BG80] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. In *Proceedings Advanced Course on Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer, 1980.
- [BG90] C. Brown and D. Gurr. A categorical linear framework for Petri nets. In *Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, Pennsylvania, 4–7 June 1990, Proceedings*, pages 208–218. IEEE, June 1990.
- [Bid91] M. Bidoit. *Algebraic System Specification and Development: A Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

- [BJM97] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 1997, Proceedings*, volume 1214 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- [BJM00] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
- [BJO99] F. Blanqui, J.-P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications, 10th International Conference, RTA-99, Trento, Italy, July 2-4, 1999*, volume 1631 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [BKK⁺96] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *RWLW'96, First International Workshop on Rewriting Logic and its Applications, Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996, Proceedings*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [BKK⁺98] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, and C. Ringeissen. An overview of ELAN. In *International Workshop on Rewriting Logic and its Applications, Abbaye des Prémontrés at Pont-à-Mousson, France, September 1998, Proceedings*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. <http://www.elsevier.nl/locate/entcs/volume15.html>.
- [BKR01a] E. Bonelli, D. Kesner, and A. Ríos. A de bruijn notation for higher-order rewriting. In *Proceedings of the 11th International Conference on Rewriting Techniques and Applications (RTA'00), Norwich, UK, July 2000*, volume 2051 of *Lecture Notes in Computer Science*, pages 62–79. Springer-Verlag, 2001.
- [BKR01b] E. Bonelli, D. Kesner, and A. Ríos. From higher-order to first-order rewriting. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA'01), Utrecht, The Netherlands, June 2001*, volume 2051 of *Lecture Notes in Computer Science*, pages 47–62. Springer-Verlag, 2001.

- [Blo97] R. Bloo. *Preservation of Termination for Explicit Substitution*. PhD thesis, Eindhoven University of Technology, 1997. IPA Dissertation Series 1997-05.
- [BMMS98] R. Bruni, J. Meseguer, U. Montanari, and V. Sassone. A comparison of Petri net semantics under the collective token philosophy. In J. Hsiang and A. Ohori, editors, *Proceedings of ASIAN'98, 4th Asian Computing Science Conference*, volume 1538 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, 1998.
- [BMMS99] R. Bruni, J. Meseguer, U. Montanari, and V. Sassone. Functorial semantics for Petri nets under the individual token philosophy. In *CTCS'99, Conference on Category Theory and Computer Science Edinburgh, UK, 10–12 September 1999, Proceedings*, volume 29 of *Electronic Notes in Theoretical Computer Science*, pages 1–19. Elsevier, 1999. <http://www.elsevier.nl/locate/entcs/volume29.html>.
- [BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [Bon99] E. Bonelli. Using fields and explicit substitutions to implement objects and functions in a de Bruijn setting. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 204–219. Springer-Verlag, 1999.
- [BR96] R. Bloo and K. H. Rose. Combinatory reduction systems with explicit substitution that preserve strong normalisation. In H. Ganzinger, editor, *Rewriting Techniques and Applications, 7th International Conference, RTA '96, New Brunswick, NJ, USA, July 27–30, 1996, Proceedings*, volume 1103 of *Lecture Notes in Computer Science*, pages 169–183. Springer-Verlag, 1996.
- [Bro89] C. Brown. Relating Petri nets to formulae of linear logic. Technical Report ECS-LFCS-89-87, Laboratory of Foundations of Computer Science, University of Edinburgh, June 1989.
- [BS00] R. Bruni and V. Sassone. Algebraic models for contextual nets. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9–15, 2000, Proceedings*, volume 1853, pages 175–186. Springer-Verlag, 2000.

- [BT80] J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J.W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.
- [Bur00] R. Burstall. Christopher strachey – understanding programming languages. *Higher-Order Symbolic Computation*, 13(1/2):11–50, May 2000.
- [BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 1990.
- [CAB⁺86] R. L. Constable, S. Allen, H. Bromely, W. Cleveland, et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
- [Car86] L. Cardelli. A polymorphic lambda-calculus with Type:Type. Technical report, Digital Equipment Corporation, 1986.
- [CC99a] M. Carpentier and K. M. Chandy. Towards a compositional approach to the design and verification of distributed systems. In J. M. Wing, J. Wookcock, and J. Davies, editors, *FM'99 – Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589. Springer-Verlag, 1999.
- [CC99b] C. Coquand and T. Coquand. Structured type theory. In *LFM'99: Workshop on Logical Frameworks and Meta-languages, Paris, France, September 28, 1999, Proceedings*, 1999. <http://plan9.bell-labs.com/who/felty/LFM99/>.
- [CDE⁺99a] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, January 1999. <http://maude.csl.sri.com>.
- [CDE⁺99b] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a Formal Meta-Tool. In J. M. Wing, J. Wookcock, and J. Davies, editors, *FM'99 – Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

- [CDE⁺00a] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *The 3rd International Workshop on Rewriting Logic and its Applications, Kanazawa City Cultural Hall, Kanazawa, Japan, September 18–20, 2000, Proceedings*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297 – 318. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
- [CDE⁺00b] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A tutorial on maude. <http://maude.csl.sri.com>, March 2000.
- [CEW93] I. Claßen, H. Ehrig, and D. Wolz, editors. *Algebraic Specification Techniques and Tools for Software Development*, volume 1 of *AMAST Series in Computing*. World Scientific, 1993.
- [CH85] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL'85, European Conference on Computer Algebra, Linz, Austria, April 1–3, 1985, Proceedings Volume 1: Invited Lectures*, volume 203 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
- [CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.
- [CH93] S. Christensen and N. D. Hansen. Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. In M. A. Marsan, editor, *Application and Theory of Petri Nets, 14th International Conference, Chicago, Illinois, USA, June 21-25, 1993, Proceedings*, volume 691 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
- [CH97] B. Chetali and B. Heyd. Formal verification of concurrent programs in LP and in COQ: A comparative analysis. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, volume 1275 of *Lecture Notes in Computer Science*, pages 69–85. Springer-Verlag, 1997.
- [Cha94] K. M. Chandy. Properties of concurrent programs. *Formal Aspects of Computing*, 6(6):607–619, 1994.
- [Che95] B. Chetali. Formal verification of concurrent programs using the larch prover. In U. H. Engberg, K. G. Larsen, and A. Skou, editors,

- Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Aarhus, Denmark, 19–20 May, 1995*. BRICS, University of Aarhus, Aarhus, Denmark, 1995.
- [Che98] B. Chetali. Formal verification of concurrent programs using the Larch prover. *IEEE Transactions on Software Engineering*, 24(1):46–62, 1998.
- [CHL96] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.
- [CHO92] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proceedings of the ACM conference on LISP and Functional Programming, San Francisco, California, June 22–24, 1992*. ACM Press, 1992.
- [Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1), 1940.
- [CK97] P. Collette and E. Knapp. A foundation for modular reasoning about safety and progress properties of state-based concurrent programs. *Theoretical Computer Science*, 183:253–279, 1997.
- [Cla98] M. Clavel. *Reflection in General Logics and in Rewriting Logic, with Applications to the Maude Language*. PhD thesis, University of Navarre, 1998.
- [CM88] K. M. Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison-Wesley, 1988.
- [CM90] K. M. Chandy and J. Misra. Proofs of distributed algorithms: An exercise. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, chapter 11, pages 305–332. Addison-Wesley, Reading, Massachusetts, 1990.
- [CM96] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In *RWLW’96, First International Workshop on Rewriting Logic and its Applications, Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996, Proceedings*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [CNJU00] R. L. Constable, P. Naumov, P. Jackson, and J. Uribe. Constructively formalizing automata theory. In G. Plotkin, C. P. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robert Milner*. MIT Press, 2000.

- [CO02] A. Coja-Oghlan. Der Zusammenhang zwischen P/T-Netzen und Termersetzungslgik in kategorientheoretischer Darstellung. Fachbereichsbericht FBI-HH-B-236/02, University of Hamburg, Germany, February 2002.
- [CoF01] CoFI Task Group on Language Design. The Common Algebraic Specification Language – Summary. <http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/>, March 2001.
- [Coq85] T. Coquand. *Une theorie des constructions*. PhD thesis, Université de Paris, 1985.
- [Coq86] T. Coquand. An analysis of Girard’s paradox. In *First Annual Symposium on Logic in Computer Science, Cambridge, Massachusetts, 16–18 June 1986, Proceedings*. IEEE, 1986.
- [Coq90a] T. Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing. Academic Press, 1990.
- [Coq90b] T. Coquand. On the analogy between propositions and types. In G. Huet, editor, *Logical Foundations of Functional Programming*, The UT year of programming series. Addison-Wesley, 1990.
- [Coq91] T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
- [Coq93] T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES’93, Nijmegen, May 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 19–61. Springer-Verlag, 1993.
- [COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995*.
- [COS02] A. Coja-Oghlan and M.-O. Stehr. Revisiting the algebra of Petri net processes under the collective token philosophy. In H.-D. Burkhard, L. Czaja, G. Lindemann, A. Skowron, and P. Starke, editors, *Concurrency, Specification and Programming, CS&P’2002, Berlin, Oktober 7–9, Volume 1*, pages 77 – 88. Humboldt Universität zu Berlin, 2002.

- [Cou02] J. Courant. Explicit universes for the calculus of constructions. In V. Carreño, C. Muñoz, and S. Tashar, editors, *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, volume 2410 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2002.
- [CPM90] T. Coquand and C. Paulin-Mohring. Inductively defined types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
- [CS87] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *Second Annual Symposium on Logic in Computer Science, Ithaca, New York, 22–25 June 1987, Proceedings*, pages 183–193. IEEE, 1987.
- [Cur91] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.
- [DAR] The DARPA Active Network Program. DARPA Information Technology Office, <http://www.darpa.mil/ito/research/anets/index.html>.
- [dB72] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Proceedings Koninkl. Nederl. Akademie van Wetenschappen*, volume 75(5), pages 381–392, 1972.
- [dB80] N. G. de Bruijn. A survey of the AUTOMATH project. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 1980.
- [Den00] E. Denney. A Prototype Proof Translator from HOL to Coq. In M. Aagaard and J. Harrison, editors, *The 13th International Conference on Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 108–125, Portland, OR, August 2000. Springer-Verlag.
- [DF98] R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

- [DFH95] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications, Second International Conference, TLCA'95, Edinburgh, UK, April 10-12, 1995, Proceedings*, volume 902 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
- [DG01] R. David and B. Guillaume. A λ -calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11:169–206, 2001.
- [DGK⁺92] J. Desel, D. Gomm, E. Kindler, R. Walter, and B. Paech. Bausteine eines kompositionalen Beweiskalküls für netzmodellerte Systeme. SFB-Bericht 342/16/92 A, Technische Universität München, Institut für Informatik, 1992.
- [DH94] J. Despeyroux and A. Hirschowitz. Higher-order abstract syntax with induction in Coq. In F. Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, volume 822 of *Lecture Notes in Artificial Intelligence*, pages 159–173. Springer-Verlag, 1994.
- [DHK95] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *Tenth Annual Symposium on Logic in Computer Science, San Diego, California, 26–29 June 1995, Proceedings*, pages 366–374. IEEE, 1995.
- [DHK98] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998. <ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz>.
- [DHK99] G. Dowek, T. Hardin, and C. Kirchner. HOL-lambda-sigma: An intentional first-order expression of higher-order logic. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications, 10th International Conference, RTA'99, Trento, Italy, July 2-4, 1999, Proceedings*, volume 1631 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1999.
- [DHP91] C. Dimitrovici, U. Hummert, and L. Petrucci. Semantics, composition and net properties of algebraic high-level nets. In G. Rozenberg, editor, *Advances in Petri Nets 1991, Papers from the 11th International Conference on Applications and Theory of Petri Nets, Paris, France, June 1990*, volume 524 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

- [DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, pages 243–320. North-Holland, 1990.
- [DM79] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
- [DM99] F. Durán and J. Meseguer. Structured theories and institutions. In M. Hofmann, G. Rosolini, and D. Pavlović, editors, *CTCS'99, Conference on Category Theory and Computer Science Edinburgh, UK, 10–12 September 1999, Proceedings*, volume 29 of *Electronic Notes in Theoretical Computer Science*, pages 71–90. Elsevier, 1999. <http://www.elsevier.nl/locate/entcs/volume29.html>.
- [DMM96] P. Degano, J. Meseguer, and U. Montanari. Axiomizing the algebra of net computations and processes. *Acta Informatica*, 33:641–667, 1996.
- [DP88] N. Dershowitz and D. A. Plaisted. Equational programming. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11: The logic and acquisition of knowledge*, pages 21–56. Oxford Press, Oxford, 1988.
- [DS99] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications. Proceedings of TLCA '99, L'Aquila, Italy*, volume 1581 of *Lecture Notes in Computer Science*, pages 129 – 146. Springer-Verlag, 1999.
- [Dur99] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Malaga, 1999.
- [Dyb91] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.
- [EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
- [EP97] H. Ehrig and J. Padberg. Uniform approach to Petri nets. In C. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Potential – Theory – Cognition*, volume 1337 of *Lecture Notes in Computer Science*, pages 219–231. Springer-Verlag, 1997.

- [EPR94] H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic high-level nets: Petri nets revisited. In H. Ehrig, editor, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types Joint with the 4th COMPASS Workshop, Caldes de Malavella, Spain, October 26-30, 1992, Selected Papers*, volume 785 of *Lecture Notes in Computer Science*, pages 188–206. Springer-Verlag, 1994.
- [EW90] U. Engberg and G. Winskel. Petri nets as models of linear logic. In A. Arnold, editor, *CAAP'90, 15th Colloquium on Trees in Algebra and Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 431 of *Lecture Notes in Computer Science*, pages 147–161. Springer-Verlag, 1990.
- [Fan92] J. Fanchon. FIFO-net models for processes with asynchronous communication. In G. Rozenberg, editor, *Advances in Petri Nets 1992, The DEMON Project*, volume 609 of *Lecture Notes in Computer Science*, pages 152–178. Springer-Verlag, 1992.
- [Far99] B. Farwer. A linear logic view of object Petri nets. *Fundamenta Informaticae*, 37(3):225–246, 1999.
- [FC88] A. Finkel and A. Choquet. FIFO nets without order deadlock. *Acta Informatica*, 25(1):15–36, 1988.
- [Fef75] S. Feferman. A language and axioms for explicit mathematics. In J. N. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 87–139. Springer-Verlag, 1975.
- [Fef88] S. Feferman. Finitary inductive systems. In R. Ferro, editor, *Proceedings of Logic Colloquium '88, Padova, Italy, August 1988*, pages 191–220. North-Holland, 1988.
- [FF86] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ -calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.
- [FGT93] W. M. Farmer, J. D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.
- [FH97] A. Felty and D. J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In W. McCune, editor, *Automated Deduction – CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13–17, 1997*,

- Proceedings*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 351–365. Springer-Verlag, 1997.
- [Fie90] J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, January 1990*, pages 1–15. ACM, 1990.
- [FM82] A. Finkel and G. Memmi. FIFO nets: New model of parallel computation. In *6th GI-Conference on Theoretical Computer Science, Dortmund*, volume 145 of *Lecture Notes in Computer Science*, pages 111–121. Springer-Verlag, 1982.
- [Fra86] N. Francez. *Fairness*. Springer-Verlag, 1986.
- [Gar92] P. Gardner. *Representing Logics in Type Theory*. PhD thesis, University of Edinburgh, 1992.
- [GB86] J. A. Goguen and R. M. Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters, and parchments. In D. Pitt et al., editors, *Proceedings of the Workshop on Category Theory and Computer Programming, Guildford, 1985*, volume 240 of *Lecture Notes in Computer Science*, pages 313–333. Springer-Verlag, 1986.
- [GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.
- [GB94] J. Goguen and R. Burstall. Introducing institutions. In E. Clarke and D. Kozen, editors, *Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer-Verlag, 1994.
- [GD94] J. A. Goguen and R. Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(3):363–392, September 1994.
- [GDMT00] J. G. Denker, Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The maude experience. In *DARPA Information and Survivability Conference and Exposition (DISCEX'00)*, pages 251–265. IEEE, January 2000.
- [Gen90] H. J. Genrich. Equivalence transformation of PrT-nets. In G. Rozenberg, editor, *Advances in Petri Nets 1989, 9th European Workshop on Applications and Theory in Petri Nets, Venice, Italy, June 1988, Selected Papers*, volume 424, pages 179–208. Springer-Verlag, 1990.

- [Gen91] H. J. Genrich. Predicate/transition nets. In *High-Level Petri Nets: Theory and Practice*, pages 3–43. Springer-Verlag, 1991.
- [Geu93] H. Geuvers. *Logics and Type Systems*. PhD thesis, University of Nijmegen, 1993.
- [Geu01] H. Geuvers. Inconsistency of classical logic in type theory. <http://www.cs.kun.nl/~herman/note.ps.gz>, April 2001.
- [GGL00] H. Goguen and J. Goubault-Larrecq. Sequent combinators: a hilbert system for the lambda calculus. *Mathematical Structures in Computer Science*, 10(1):1–79, 2000.
- [GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.
- [Gim94] E. Giménez. Codifying guarded definitions with recursive schemes. In *BRA Workshop on Types for Proofs and Programs (TYPES'94)*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59, 1994.
- [Gir72] J. Y. Girard. *Interpretation fonctionnelle et elimination des coupures dans l'arithmetique d'ordre superieur*. PhD thesis, Université Paris VII, 1972.
- [Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [GL79] H. J. Genrich and K. Lautenbach. The analysis of distributed systems by means of predicate/transition-nets. In G. Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 123–146. Springer-Verlag, 1979.
- [GL81] H. J. Genrich and K. Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.
- [GL99] J. Goubault-Larrecq. Conjunctive types and SKInT. In *International Workshop on Types for Proofs and Programs (TYPES'98), Kloster Irsee, Germany, March 1998. Selected papers*, volume 1657 of *Lecture Notes in Computer Science*, pages 106–120. Springer Verlag, 1999.
- [GLT89] J. Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
- [GM85] J. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

- [GM86] J. A. Goguen and J. Meseguer. EQLOG: equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–364. Prentice-Hall, 1986.
- [GM87] J. Goguen and J. Meseguer. Models and equality for logic programming. In Ehrig H, R. A. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy, March 23-27, 1987, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Functional and Logic Programming and Specifications (CFLP)*, volume 250 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 1987.
- [GM92] J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
- [GM93a] J. A. Goguen and J. Meseguer. Order-sorted algebra solves the constructor-selector, multiple-representation, and coercion problems. *Information and Computation*, 103:114–158, 1993.
- [GM93b] M. J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [GM96a] J. A. Goguen and G. Malcom. *Algebraic Semantics of Imperative Programs*. The MIT Press, Cambridge, Massachusetts, 1996.
- [GM96b] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125, pages 173–190. Springer-Verlag, 1996.
- [GM00] J. Goguen and G. Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, August 2000. Special Issue on Algebraic Engineering edited by C. Nehaniv and M. Ito.
- [GN91] H. Geuvers and M.-J. Nederhof. A modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, April 1991.
- [Gog] J. A. Goguen. *Theorem Proving and Algebra*. MIT Press. To appear.

- [Gol90] D. M. Goldschlag. Mechanically verifying concurrent programs with the Boyer-Moore prover. *IEEE Transactions on Software Engineering*, 16(9), 1990.
- [Gol92a] D. M. Goldschlag. *Mechanically Verifying Concurrent Programs*. PhD thesis, University of Texas at Austin, 1992. Also available as Technical Report 71 from Computational Logic, Inc.
- [Gol92b] D. M. Goldschlag. Mechanically verifying safety and liveness properties of delay insensitive circuits. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July 1-4, 1991, Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 354-364. Springer-Verlag, July 1992.
- [Gol99] D. M. Goldschlag. A mechanization of Unity in PC-NQTHM-92. *Journal of Automated Reasoning*, 23(3-4):445-498, 1999.
- [GP89] R. Gerth and A. Pnueli. Rooting UNITY. In *Proceedings Fifth International Workshop on Software Specification and Design*, Pittsburgh, Pennsylvania, May 1989.
- [Gru92] K. Grue. Map theory. *Theoretical Computer Science*, 102:1-133, 1992.
- [Gui00] B. Guillaume. The λ_{se} -calculus does not preserve strong normalisation. *Journal of Functional Programming*, 10(4):321-325, July 2000.
- [GWM⁺92] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ*. Cambridge University Press, 1992.
- [Har89] T. Hardin. Confluence results for the pure strong categorical logic CCL: lambda-calculi as subsystems of CCL. *Theoretical Computer Science*, 65(3):291-342, 1989.
- [HC96] B. Heyd and P. Crégut. A modular coding of UNITY in Coq. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125. Springer-Verlag, 1996.
- [Hey71] A. Heyting. *Intuitionism - An Introduction*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1971.

- [Hey97] B. Heyd. *Application de la théorie des types et du démonstrateur COQ à la vérification de programmes parallèles*. PhD thesis, Université Henri Poincaré – Nancy 1, 1997.
- [HHJW96] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.
- [HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science, Ithaca, New York, 22–25 June 1987, Proceedings*, pages 193–204. IEEE, 1987.
- [Hic96] J. J. Hickey. Formal objects in type theory using very dependent types. In K. Bruce and G. Longo, editors, *Informal Proceedings of Third Workshop on Foundations of Object-Oriented Languages (FOOL 3)*, 1996.
- [Hir99] D. Hirschhoff. Handling substitutions explicitly in the pi-calculus. In *Proceedings of WESTAPP'99, Second International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs, Trento, Italy, July 5, 1999*, 1999.
- [HK99] M. Hicks and A. D. Keromytis. A secure PLAN. In S. Covaci, editor, *Active Networks, First International Working Conference, IWAN '99, Berlin, Germany, June 30 – July 2, 1999, Proceedings*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999. <http://www.cis.upenn.edu/~switchware/papers/iwan99.ps>. Extended version at <http://www.cis.upenn.edu/~switchware/papers/secureplan.ps>.
- [HKM⁺98a] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Network programming using PLAN. In *Proceedings of the 1998 Workshop on Internet Programming Languages (IPL'98), Part of IEEE International Conference on Computer Languages (ICCL'98), Chicago, IL, May 1998*, May 1998. <http://www.cis.upenn.edu/~switchware/papers/progplan.ps>.
- [HKM⁺98b] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland, September 1998*, pages 86–93. ACM, 1998. <http://www.cis.upenn.edu/~switchware/papers/plan.ps>.

- [HMA⁺99] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. PLANet: An active internet network. In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFO-COM Conference*, pages 1124–1133. IEEE, 1999. <http://www.cis.upenn.edu/~switchware/papers/planet.ps>.
- [HMST95] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, 1995.
- [HO80] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
- [Hof99] M. Hoffmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science, Trento, Italy, 2–5 July 1999, Proceedings*, pages 214–224. IEEE, July 19901999.
- [Hof00] K. Hoffmann. Run time modification of algebraic high level nets and algebraic higher order nets using folding and unfolding construction. In G. Hommel, editor, *Communication-Based Systems, Proceedings of the 3rd International Workshop held at the TU Berlin, Germany, 31 March – 1 April 2000*, pages 55–72. Kluwer Academic Publishers, 2000.
- [How80] W. A. Howard. The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic*. Academic Press, 1980.
- [How89] D. J. Howe. Equality in lazy computation systems. In *Fourth Annual Symposium on Logic in Computer Science, Asilomar Conference Center, Pacific Grove, California, 5-8 June 1989, Proceedings*, pages 198–203. IEEE, 1989.
- [How91] D. J. Howe. On computational open-endedness in Martin-Löf’s type theory. In *Sixth Annual IEEE Symposium on Logic in Computer Science, Amsterdam, The Netherlands, 15-18 July 1991, Proceedings*, pages 162–172. IEEE, 1991.
- [How96a] D. J. Howe. Importing mathematics from HOL into Nuprl. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs’96*,

- Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282. Springer Verlag, 1996.
- [How96b] D. J. Howe. Semantical foundations for embedding HOL in Nuprl. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101, Berlin, 1996. Springer-Verlag.
- [How97] D. J. Howe. A classical set-theoretic model of polymorphic extensional type theory. Manuscript (submitted for publication), 1997.
- [How98a] D. J. Howe. Toward sharing libraries of mathematics between theorem provers. In *Frontiers of Combining Systems, FroCoS'98, ILLC, University of Amsterdam, October 2-4, 1998, Proceedings*. Kluwer Academic Publishers, 1998.
- [How98b] D. J. Howe. A type annotation scheme for Nuprl. In J. Grundy and M. C. Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'97, Canberra, Australia, September 27 - October 1, 1998, Proceedings*, volume 1479 of *Lecture Notes in Computer Science*, pages 207–224. Springer-Verlag, 1998.
- [How99] D. J. Howe. Source Code of the HOL-Nuprl Translator (including Extensions to Nuprl), January 1999.
- [HP91] R. Harper and R. Pollack. Type checking, universe polymorphism and typical ambiguity in the calculus of constructions. *Theoretical Computer Science*, 81(1), 1991.
- [HS90] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda-Calculus*, volume 1 of *London Mathematical Society student texts*. Cambridge University Press, 1990.
- [HS94] D. J. Howe and S. Stoller. An operational approach to combining classical set theory and functional programming languages. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS'94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 36–55. Springer-Verlag, 1994.
- [HS98] G. Huet and A. Saïbi. Constructive category theory. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

- [Hue87] G. Huet. Extending the calculus of constructions with Type:Type. Unpublished Manuscript, April 1987.
- [Hue89] G. Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. World Scientific, 1989.
- [Hue94] G. Huet. Residual theory in λ -calculus: A formal development. *Journal of Functional Programming*, 4:371–394, 1994.
- [Jac94] P. B. Jackson. Exploring abstract algebra in constructive type theory. In A. Bundy, editor, *Automated Deduction – CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 – July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1994.
- [Jen81] K. Jensen. Coloured Petri nets and the invariant-method. *Theoretical Computer Science*, pages 317–336, 1981.
- [Jen92] K. Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.
- [JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [JJM97] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: Exploring the design space. In *Proceedings of the Second Haskell Workshop, Amsterdam, June 1997*, 1997.
- [JO97] J.-P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 1997.
- [Jon92a] M. P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.
- [Jon92b] M. P. Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *ESOP’92: 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
- [Jon00] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany, March 2000*, volume 1782 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

- [Jou98] J.-P. Jouannaud. Membership equational logic, calculus of inductive constructions, and rewrite logic. In *International Workshop on Rewriting Logic and its Applications, Abbaye des Prémontrés at Pont-à-Mousson, France, September 1998, Proceedings*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. <http://www.elsevier.nl/locate/entcs/volume15.html>.
- [JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
- [Kak99] P. Kakkar. The specification of PLAN. <http://www.cis.upenn.edu/~switchware/PLAN/spec/spec.ps>, 1999.
- [KBS91] B. Krieg-Brückner and D. Sanella. Structuring specifications in-the-large and in-the-small: Higher-order functions, depending types and inheritance in SPECTRAL. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8–12, 1991, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD)*, 494, pages 313–336. Springer-Verlag, 1991.
- [Kel76] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.
- [Kes00] D. Kesner. Confluence of extensional and non-extensional λ -calculi with explicit substitutions. *Theoretical Computer Science*, 238(1–2):183–220, 2000.
- [KGA00] P. Kakkar, C. A. Gunther, and M. Abadi. Reasoning About Secrecy for Active Networks. In *13th IEEE Computer Security Foundations Workshop (CSFW'00), 3 – 5 July 2000, Cambridge, England, Proceedings*, 2000. <http://www.cis.upenn.edu/~switchware/papers/csfw.ps>.
- [KHMG99] P. Kakkar, M. Hicks, J. T. Moore, and C. A. Gunter. Specifying the PLAN networking programming language. In *HOOTS'99, Higher Order Operational Techniques in Semantics Paris, France, September 30 and October 1, 1999, Proceedings*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999. <http://www.elsevier.nl/locate/entcs/volume26.html>.
- [Kin95] E. Kindler. Invariants, composition, and substitution. *Acta Informatica*, 32(4):299–312, 1995.

- [KL99] D. Kesner and P. E. M. López. Explicit substitutions for objects and functions. *The Journal of Functional and Logic Programming*, 1999. Special Issue 2.
- [Klo80] J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, Rijksuniversiteit Utrecht, June 1980. Mathematical Centre Tracts 127.
- [Klo92] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–117. Clarendon Press, Oxford, 1992.
- [KMT88] E. Kettunen, E. Montonen, and T. Tuuliniemi. Comparison of Petri net based channel models. In *Proceedings of the 12th IMACS World Conference*, volume 3, pages 479–482, 1988.
- [Kna92] E. Knapp. Derivation of concurrent programs. *Sci. Comput. Programming*, 19:1–23, 1992.
- [Kna94] E. Knapp. Soundness and completeness of UNITY logic. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science, 14th Conference, Madras, India, December 15–17, 1994, Proceedings*, volume 880 of *Lecture Notes in Computer Science*, pages 378–389. Springer-Verlag, 1994.
- [KR96] E. Kindler and W. Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, December 1996.
- [KR97] F. Kamareddine and A. Ríos. Extending a lambda-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, 1997.
- [KR00] F. Kamareddine and A. Ríos. Relating the $\lambda\sigma$ - and λs -styles of explicit substitutions. *Journal of Logic and Computation*, 10(3):349–380, June 2000.
- [Kre99] C. Kreitz. Automated fast-track reconfiguration of group communication systems. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 104–118. Springer Verlag, 1999.
- [KRVW97] E. Kindler, W. Reisig, H. Völzer, and R. Walter. Petri net based verification of distributed algorithms: An example. *Formal Aspects of Computing*, 9:409–424, 1997.

- [KV98] E. Kindler and H. Völzer. Flexibility in algebraic nets. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998, 19th International Conference, ICATPN'98, Lisbon, Portugal, June 1998, Proceedings*, volume 1420 of *Lecture Notes in Computer Science*, pages 345–384. Springer-Verlag, 1998.
- [Lak95] C. A. Lakos. From coloured Petri nets to object Petri nets. In G. De Michelis and M. Diaz, editors, *Application and Theory of Petri Nets 1995, 16th International Conference, Turin, Italy, June 26-30, 1995, Proceedings*, volume 935 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1995.
- [Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):126–143, 1977.
- [Lam90] L. Lamport. Win and sin: predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12:396–428, 1990.
- [Lam93] L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.
- [Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [LB88] B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76(2/3):278–346, 1988.
- [Les92] P. Lescanne. Termination of rewrite systems by elementary interpretations. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 1992, Proceedings*, volume 632 of *Lecture Notes in Computer Science*, pages 21 – 36. Springer-Verlag, 1992.
- [Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\nu$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January 17–21, 1994*, pages 60–69. ACM, 1994.
- [LM96] C. Laneve and U. Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 6(3):219–249, 1996.

- [LO93] K. Läufer and M. Odersky. Self-Interpretation and Reflection in a Statically Typed Language. In *OOPSLA Workshop on Reflection and Metalevel Architectures, Washington*, September 1993.
- [LP92] Z. Luo and R. Pollack. Lego proof development system: User's manual. LFCS Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.
- [LP97] L. Lamport and L. C. Paulson. Should your specification language be typed? Technical Report 147, Digital Equipment Corporation, May 1997.
- [LRD94] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions λv . Technical Report RR-2222, INRIA-Lorraine, January 1994.
- [Luo91] Z. Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90(1):107–137, 1991.
- [Luo92] Z. Luo. A unifying theory of dependent types: the schematic approach. In *Proceedings of Symposium on Logical Foundations of Computer Science (Logic at Tver 92)*, volume 620 of *Lecture Notes in Computer Science*, 1992. Also as report ECS-LFCS-92-202, Department of Computer Science, Edinburgh University.
- [Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.
- [LvRBK01] X. Liu, R. van Renesse, M. Bickford, and C. Kreitz. Proving hybrid protocols correct. In *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs'2001, Edinburgh, Scotland, UK, September 3–6, 2001, Proceedings*, Lecture Notes in Computer Science, pages 329 – 345. Springer-Verlag, 2001.
- [Mac71] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.
- [Mac86] D. MacQueen. Using dependent types to express modular structure. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 277–286, 1986.
- [Mac91] R. Mackenthun. Using temporal logic to develop correct coloured nets. Fachbereichsmitteilung FBI-HH-M-227/93, Universität Hamburg, Fachbereich Informatik, 1991.

- [Mag94] L. Magnussen. *The Implementation of ALF – A Proof Editor based on Martin-Löf’s Monomorphic Type Theory with Explicit Substitutions*. PhD thesis, University of Göteborg, Department of Computer Science, 1994.
- [Mar96] C. Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253–288, 1996. <ftp://ftp.lri.fr/LRI/articles/marche/jsc96.ps.gz>.
- [Mas99] I. A. Mason. Computing with contexts. *Journal of Higher-Order Symbolic Computation*, 12(2):171–201, September 1999.
- [Mel95] P.-A. Mellis. Typed λ -calculi with explicit substitution may not terminate. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications, Second International Conference, TLCA ’95, Edinburgh, UK, April 10-12, 1995, Proceedings*, volume 902 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
- [Mes89a] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Logic Colloquium’87, Granada, Spain, July 1987, Proceedings*, pages 275–329. North-Holland, 1989.
- [Mes89b] J. Meseguer. Relating models of polymorphism. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages, Austin, Texas, January 1989*, pages 228–241. ACM Press, 1989.
- [Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
- [Mes93] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
- [Mes96] J. Meseguer. Rewriting logic as a semantic framework for concurrency. A progress report. In U. Montanari and V. Sassone, editors, *CONCUR’96: Concurrency Theory, 7th International Conference, Pisa, Italy, August 26–29, 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer-Verlag, 1996.
- [Mes98] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop*,

- WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18 – 61. Springer-Verlag, 1998.
- [MG85] J. Meseguer and J. A. Goguen. Initiality, induction and computability. In *Algebraic Methods in Semantics*, pages 459–540. Cambridge University Press, 1985.
- [MHN99] J. T. Moore, M. Hicks, and S. M. Nettles. Chunks in PLAN: Language support for programs as packets. Technical report, Department of Computer and Information Science, University of Pennsylvania, April 1999. <http://www.cis.upenn.edu/~switchware/papers/planchunks.ps>.
- [Mil99] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, May 1999.
- [Mis90a] J. Misra. A simple proof of a simple consensus algorithm. In *Beauty is Our Business*, chapter 35, pages 312–318. Springer-Verlag, New York, 1990.
- [Mis90b] J. Misra. Soundness of the substitution axiom. Notes on UNITY, <http://www.cs.utexas.edu/users/psp/notesunity.html>, March 1990.
- [Mis94] J. Misra. New UNITY. Manuscript, <http://www.cs.utexas.edu/users/psp/newunity.html>, 1994.
- [Mis95] J. Misra. A logic for concurrent programming: Safety and progress. *Journal of Computer and Software Engineering*, 3(2):239–300, 1995.
- [Mis96] J. Misra. A discipline of multiprogramming. *ACM Computing Surveys*, 28A(4), December 1996.
- [Mis98] J. Misra. An object model for multiprogramming. In J. Rolim, editor, *Parallel and Distributed Processing, 10 IPSP/SPDP'98 Workshops Held in Conjunction with the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, Orlando, Florida, USA, March 30 - April 3, 1998, Proceedings*, volume 1388 of *Lecture Notes in Computer Science*, pages 881–889. Springer-Verlag, 1998.
- [Mis99] J. Misra. A simple, object-based view of multiprogramming. *Formal Methods in System Design*, 1999.

- [Mis01] J. Misra. *A discipline of multiprogramming: programming theory for distributed applications*, volume 18 of *Monographs in Computer Science*. Springer-Verlag, 2001.
- [ML72] P. Martin-Löf. An intuitionistic theory of types. Technical report, University of Stockholm, 1972.
- [ML74] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium 73*, pages 73–108. North-Holland, 1974.
- [ML82] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North Holland, 1982.
- [ML84] P. Martin-Löf. *Type Theory*. Bibliopolis, 1984.
- [MM90] J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, October 1990.
- [MMO95] J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, S. Margherita, Italy, May 30 - June 3, 1994, Selected Papers*, volume 906 of *Lecture Notes in Computer Science*, pages 48–80. Springer-Verlag, 1995.
- [MMS92] J. Meseguer, U. Montanari, and V. Sassone. On the semantics of Petri nets. In R. Cleaveland, editor, *CONCUR '92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992. Proceedings*, volume 630 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 1992.
- [MMS94] J. Meseguer, U. Montanari, and V. Sassone. On the model of computation of place/transition Petri nets. In *Application and Theory of Petri Nets 1994, Proceedings of the 15th International Conference, Zaragoza, Spain, June 20 - 24, 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 16–38. Springer-Verlag, 1994.
- [MMS96] J. Meseguer, U. Montanari, and V. Sassone. Process versus unfolding semantics for place/transition Petri nets. *Theoretical Computer Science*, 153(1–2):171–210, 1996.
- [MMS97] J. Meseguer, U. Montanari, and V. Sassone. Representation theorems for Petri nets. In C. Freska, M. Jantzen, and R. Valk, editors,

- Foundations of Computer Science: Potential – Theory – Cognition*, volume 1337 of *Lecture Notes in Computer Science*, pages 239–249. Springer-Verlag, 1997.
- [MOM91a] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic. *Mathematical Structures in Computer Science*, 1:69–101, 1991.
- [MOM91b] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *International Journal of Foundations of Computer Science*, 2(4):297–399, 1991.
- [MOM94] N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.
- [MOM96] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *RWLW'96, First International Workshop on Rewriting Logic and its Applications, Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996, Proceedings*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>. To appear in D. M. Gabbay, F. Guenther, (eds.), *Handbook of Philosophical Logic* (2nd edition), Kluwer Academic Publishers.
- [Mos94] Y. Moschovakis. *Notes on set theory*. Springer-Verlag, 1994.
- [MÖST02] J. Meseguer, P. C. Ölveczky, M.-O. Stehr, and C. L. Talcott. Maude as a wide-spectrum framework for formal modeling and analysis of active networks. In *DARPA Active Networks Conference and Exposition (DANCE), San Francisco, May 2002*. <http://schafercorp-ballston.com/dance2002/>.
- [MP85] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 1985*, pages 37–51, 1985.
- [MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
- [MP93] J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16–18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

- [MR95] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32:545–596, 1995.
- [MR97] P. J. McCann and G.-C. Roman. Mobile UNITY coordination constructs applied to packet forwarding for mobile hosts. In *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 338–354, Berlin, 1997. Springer-Verlag.
- [MR98] P. J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, February 1998.
- [MS99] J. Meseguer and M.-O. Stehr. Specifying logic translators and proof assistants in rewriting logic. Talk and Tool Demonstration at the DARPA Formal Methods Meeting, Cornell, Ithaca, NY, May 1999.
- [MT99] J. Meseguer and C. L. Talcott. A partial order event model for concurrent objects. In *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24–27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 415–430. Springer-Verlag, 1999.
- [MT01] I. A. Mason and C. L. Talcott. Feferman-Landin logic. In *Reflections – A symposium honoring Solomon Feferman on his 70th birthday*, Lecture Notes in Logic. Association of Symbolic Logic, 2001.
- [Muñ97] C. Muñoz. A calculus of substitutions for incomplete-proof representation in type theory. Technical Report RR-3309, Unité de recherche INRIA-Rocquencourt, November 1997.
- [Muñ98] C. Muñoz. Proof synthesis via explicit substitutions on open terms. In *Proceedings of WESTAPP'98, International Workshop on Explicit Substitutions, Theory and Applications, Tsukuba, Japan, April 1998*, 1998.
- [MV91] R. Mackenthun and R. Valk. Verifying coloured nets in UNITY-style. Fachbereichsmitteilung FBI-HH-M-222/93, Universität Hamburg, 1991.
- [NGV+94] R. P. Nederpelt, J. H. Geuvers, R. C. De Vrijer, L. S. van Benthem Jutting, and D. T. Van Daalen. *Selected Papers on Automath*. Number 133 in *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1994.
- [Nip91] T. Nipkow. Higher-order critical pairs. In *Sixth Annual IEEE Symposium on Logic in Computer Science, Amsterdam, The Netherlands, 15–18 July 1991, Proceedings*, pages 312–319. IEEE, 1991.

- [Nip01] T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.
- [NM98] G. Nadathur and D. Miller. Higher-order logic programming. In *Handbook of Logic in AI and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.
- [NSM01] P. Naumov, M.-O. Stehr, and J. Meseguer. The HOL/NuPRL proof translator — A practical approach to formal interoperability. In *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs'2001, Edinburgh, Scotland, UK, September 3–6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 329 – 345. Springer-Verlag, 2001.
- [O'D77] M. J. O'Donnell. Computing in systems described by equations. In *Fundamentals of Computation Theory, International Conference, Poznań-Kornik, Poland September 19–23, 1977, Proceedings*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.
- [O'D85] M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.
- [OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285, May 1976.
- [ONR] The ONR Coordinated Research in Adaptive Network Centric Computing (CRANCC). <http://www.cis.upenn.edu/sdrl/crancc/>.
- [ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction – CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.
- [Pac90] J. Pachl. A simple proof of a completeness result for leads-to in the UNITY logic. Technical Report RZ 2060, IBM Research Division, Zurich Research Laboratory, 1990.
- [Pac92] J. Pachl. A simple proof of a completeness result for leads-to in the UNITY logic. *Information Processing Letters*, 41, 1992.
- [Pad88] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.

- [Pag98] B. Pagano. X.R.S.: Explicit reduction systems – a first-order calculus for higher-order calculi. In C. Kirchner and H. Kirchner, editors, *Automated Deduction – CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 1998, Proceedings*, volume 1421 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1998.
- [Pau90] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing. Academic Press, 1990.
- [Pau94] L. C. Paulson. *Isabelle*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
- [Pau99] L. C. Paulson. Compositional proofs of concurrent programs. Project GR/M 75440 funded by the EPSRC (1999 – 2003), Computer Laboratory, University of Cambridge, <http://www.cl.cam.ac.uk/users/lcp/Grants/UNITY.html>, 1999.
- [Pau00] L. C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1):3–32, 2000.
- [PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN’88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, 22-24 June 1988*, SIGPLAN Notices 23(7), pages 199–208, 1988.
- [Pet62] C. A. Petri. Kommunikation mit Automaten. Schriften des IIM 2, Institut für Instrumentelle Mathematik, Bonn, 1962.
- [Pet91] J. S. Pettersson. Comments on “always-true is not invariant”: Assertion reasoning about invariance. *Information Processing Letters*, 40(5):231–233, 1991.
- [Pet96] C. A. Petri. Nets, time and space. *Theoretical Computer Science*, 153(1–2):3–48, 1996. Special volume on Petri Nets.
- [Pfe91] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
- [Pfe96] F. Pfenning. The practice of logical frameworks. In H. Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, volume 1059 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 1996.

- [Pit87] A. Pitts. Polymorphism is set-theoretic, constructively. In *Category Theory and Computer Science, Proceedings, Edinburgh, 1987*, volume 283 of *Lecture Notes in Computer Science*, pages 12–39. Springer-Verlag, 1987.
- [Pla95] D. Plaisted. Term rewriting systems. *Fundamenta Informaticae*, 24:1–207, 1995.
- [PM93] C. Paulin-Mohring. Inductive Definitions in the system Coq – Rules and Properties. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference, TLCA '93, Utrecht, The Netherlands, March 16–18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.
- [Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *Series F: Computer and System Science*, pages 123–144. Springer-Verlag, 1985.
- [Pol90] R. Pollack. Implicit syntax. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks, Antibes, May 1990*, 1990. <http://www.dcs.ed.ac.uk/home/lego/html/papers.html>.
- [Pol93] R. Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer-Verlag, 1993.
- [Pol94] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
- [Pol95] R. Pollack. A verified typechecker. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications, Edinburgh, UK, April 10–12, 1995*, volume 902 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
- [Pol01] R. Pollack. Programming with dependent types: Examples. Dagstuhl Seminar 01341: Dependent Type Theory meets Practical Programming, September 2001.

- [Pra94] I. S. W. B. Prasetya. Error in the UNITY substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.
- [PRM97] G. P. Picco, G.-C. Roman, and P. J. McCann. Expressing code mobility in mobile UNITY. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference held jointly with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 500–518, Zurich, Switzerland, 1997. Springer-Verlag.
- [PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7–10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer-Verlag, 1999.
- [PSN90] K. Petersson, J. Smith, and B. Nordstroem. *Programming in Martin-Löf’s Type Theory. An Introduction*. International Series of Monographs on Computer Science. Oxford: Clarendon Press, 1990.
- [RB88] D. E. Rydeheard and R. M. Burstall. *Computational Category Theory*. Prentice Hall, 1988.
- [Rei91] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, 1991.
- [Rei95] H. Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, pages 129–152, 1995.
- [Rei98a] C. Reinke. *Functions, Frames and Interactions — completing a λ -calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments*. PhD thesis, Christian-Albrechts-Universität Kiel, Germany, 1998. Appeared as Report 9804.
- [Rei98b] W. Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag, 1998.
- [Rey74] J. Reynolds. Towards a theory of type structure. In *Programming Symposium, Paris*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974.

- [Rey84] J. C. Reynolds. Polymorphism is not set-theoretic. In *Lecture Notes in Computer Science*, volume 173, 1984.
- [RF93] G. Reichwein and J. L. Fiadeiro. Models for the substitution axiom of UNITY logic. *Information Processing Letters*, 48(4):171–176, 1993.
- [RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [RM99] G.-C. Roman and P. J. McCann. An introduction to mobile UNITY. In *Parallel and Distributed Processing, 11 IPPS/SPDP'99 Workshops held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, San Juan, Puerto Rico, USA, April 12-16, 1999, Proceedings*, volume 1586 of *Lecture Notes in Computer Science*, pages 871–880. Springer-Verlag, 1999.
- [RMP96] G.-C. Roman, P. J. McCann, and J. Y. Plun. Assertional reasoning about pairwise transient interactions in mobile computing. In *18th International Conference on Software Engineering*, pages 155–164, 1996.
- [RMP97] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, 1997.
- [Ros96] K. H. Rose. Explicit substitution – tutorial & survey. Lecture Series LS-96-3, BRICS, Department of Computer Science, University of Aarhus, September 1996. <http://www.brics.dk/LS/96/3/BRICS-LS-96-3/BRICS-LS-96-3.html>.
- [RT99] P. Rudnicki and A. Trybulec. On equivalents of well-foundedness. *Journal of Automated Reasoning*, 23:197–234, 1999.
- [RV88] W. Reisig and J. Vautherin. An algebraic approach to high level Petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1988, covers the 8th European Workshop on Application and Theory of Petri Nets, Zaragoza, Spain, June 1987*, volume 340 of *Lecture Notes in Computer Science*, pages 51–72, 1988.
- [San91] B. A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.
- [Sev96] P. G. Severi. *Normalization in Lambda Calculus and its relation to Type Inference*. PhD thesis, Eindhoven University of Technology, 1996.

- [SG96] M. Stefanova and H. Geuvers. A simple model construction for the calculus of constructions. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
- [Sha93] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):226–262, 1993.
- [Sha98] N. Shankar. A lazy approach to compositional verification. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997, Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
- [Sho77] J. R. Shoenfield. Axioms of set theory. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 321–344. North-Holland, 1977.
- [Sib94] C. Sibertin-Blanc. Cooperative nets. In R. Valette, editor, *Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 471–490, Berlin, 1994. Springer-Verlag.
- [SLU90] S. D. Swierstra, P. J. A. Lentfert, and A. H. Uittenbogaard. Distributed incremental maximum finding in hierarchically divided graphs. Technical Report RUU-CS-90-30, Utrecht University, September 1990.
- [SM99] M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. In *LFM'99: Workshop on Logical Frameworks and Metalanguages, Paris, France, September 28, 1999, Proceedings*, 1999. <http://plan9.bell-labs.com/who/felty/LFM99/>. Extended version submitted for publication.
- [SMÖ01a] M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Representation and execution of Petri nets using rewriting logic as a uniform framework. In H. Ehrig, C. Ermel, and J. Padberg, editors, *UNIGRA'2001, Uniform Approaches to Graphical Process Specification Techniques, Genova, Italy, March 31st and April 1st, 2001, Proceedings*, volume 44 of *Electronic Notes in Theoretical Computer Science*, 2001.
- [SMÖ01b] M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Rewriting logic as a unifying framework for Petri nets. In H. Ehrig, G. Juhas, J. Padberg, and G. Rozenberg, editors, *Unifying Petri Nets, Advances in Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*, pages 250–303. Springer-Verlag, 2001.

- [SNM00] M.-O. Stehr, P. Naumov, and J. Meseguer. A proof-theoretic approach to HOL-Nuprl connection with applications to proof translation (full version). Manuscript, CSL, SRI-International, Menlo Park, CA, USA, March 2000.
- [SNM01] M.-O. Stehr, P. Naumov, and J. Meseguer. A proof-theoretic approach to HOL-Nuprl connection with applications to proof translation (extended abstract). In *WADT/CoFI'01, 15th International Workshop on Algebraic Development Techniques and General Workshop of the CoFI WG, Genova, Italy, April 1 – 3, 2001, Proceedings*, 2001. See [SNM00] for the full version.
- [SO89] A. K. Singh and R. Overbeek. Derivation of efficient parallel programs: An example from genetic sequence analysis. *International Journal of Parallel Programming*, 18:447–484, 1989.
- [SP98] C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner and H. Kirchner, editors, *Automated Deduction – CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5–10, 1998, Proceedings*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 286–300. Springer-Verlag, 1998.
- [ST91] D. Sannella and A. Tarlecki. Extended ML: past, present and future. In H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification, 7th Workshop on Abstract Data Types, Wusterhausen, Dosse, Germany, April 17–20, 1990, Proceedings*, volume 534 of *Lecture Notes in Computer Science*, pages 297–322. Springer, 1991.
- [ST01] M.-O. Stehr and C. L. Talcott. Specifying an active network programming language in rewriting logic. Manuscript, SRI International and University of Hamburg, Nov 2001.
- [ST02a] M.-O. Stehr and C. L. Talcott. PLAN in Maude: Specifying an active network programming language. In F. Gadducci and U. Montanari, editors, *The 4th International Workshop on Rewriting Logic and its Applications, Pisa, Italy, September 19–21, 2002, Proceedings*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. <http://www.elsevier.nl/locate/entcs/volume71.html>.
- [ST02b] M.-O. Stehr and C. L. Talcott. Termination of active network programs. Manuscript, SRI International and University of Hamburg, Feb 2002.

- [Sta88] M. G. Staskauskas. The formal specification and design of a distributed electronic funds transfer system. *IEEE Transactions on Computers*, 37(12):1515–1528, December 1988.
- [Sta93] M. G. Staskauskas. Formal derivation of concurrent programs: an example from industry. *IEEE Transactions on Software Engineering*, 19:503–528, 1993.
- [Stea] M.-O. Stehr. Assertion reasoning. In C. Girault and R. Valk, editors, *Petri Nets for System Engineering – A Guide to Modeling, Verification, and Applications*. Springer-Verlag. To appear.
- [Steb] M.-O. Stehr. Assertion reasoning and temporal logic for system verification. Lecture at the MATCH Advanced Summer School on System Engineering, September 14-22, 1998, Jaca, Spain. <http://www.cps.unizar.es/deps/DIIS/MATCH>.
- [Stec] M.-O. Stehr. A rewriting semantics for algebraic nets. In C. Girault and R. Valk, editors, *Petri Nets for System Engineering – A Guide to Modeling, Verification, and Applications*. Springer-Verlag. To appear.
- [Ste98] M.-O. Stehr. Embedding UNITY into the calculus of inductive constructions. Fachbereichsbericht FBI-HH-B-214/98, University of Hamburg, Germany, September 1998.
- [Ste99] M.-O. Stehr. CINNI - A New Calculus of Explicit Substitutions and its Application to Pure Type Systems. Manuscript, CSL, SRI-International, Menlo Park, CA, USA, 1999.
- [Ste00a] M.-O. Stehr. CINNI – A Generic Calculus of Explicit Substitutions and its Application to λ -, σ - and π -calculi. In K. Futatsugi, editor, *The 3rd International Workshop on Rewriting Logic and its Applications, Kanazawa City Cultural Hall, Kanazawa, Japan, September 18–20, 2000, Proceedings*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71 – 92. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
- [Ste00b] M.-O. Stehr. Concurrent object-oriented specification and analysis in Maude. In *FMOODS'2000, IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems, Stanford University, Stanford, CA, USA, September 6–8, 2000, Poster Abstracts*, 2000. <http://www.ics.uci.edu/~fmds2000/>.

- [SW83] D. Sanella and M. Wirsing. A kernel language for algebraic specification and implementation. In M. Karpinski, editor, *Proceedings 11th Colloquium on Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 1983.
- [Tal91] C. L. Talcott. Binding structures. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.
- [Tal93] C. L. Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112(1):99–143, 1993.
- [Tal00] C. L. Talcott. Specifying the PLAN language in Maude (joint work with J. Meseguer, M.-O. Stehr, C. Gunter and P. Kakkar), 2000. Talk given at University of Tokyo. Slides available at <http://www-formal.stanford.edu/clt/Talks/00sep-utokyo-talk.ps.gz>.
- [Tal01] C. L. Talcott. PLAN in Maude: A specification with multiple purposes (joint work with J. Meseguer, M.-O. Stehr, C. Gunter and P. Kakkar). In *Abstracts of the Workshop on Proofs for Mobility (PFM'2001), Genova, April 7, 2001*, 2001. <http://www.pfm2001.ens.fr>.
- [Tar38] A. Tarski. Über Unerreichbare Kardinalzahlen. *Fundamenta Mathematicae*, 30:176–183, 1938.
- [Tar39] A. Tarski. On well-ordered subsets of any set. *Fundamenta Mathematicae*, 32:176–183, 1939.
- [TBG91] A. Tarlecki, R. M. Burstall, and J. A. Goguen. Some fundamental algebraic tools for the semantics of computation, III: indexed categories. *Theoretical Computer Science*, 91:239–264, 1991.
- [Ter89] J. Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, The Netherlands, 1989.
- [Tho91] S. Thompson. *Type Theory and Functional Programming*. International Computer Science Series. Addison-Wesley, 1991.
- [TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

- [Tsu01] Y. Tsukada. Martin-löf's type theory as an open-ended framework. *International Journal of Foundations of Computer Science*, 12(1):31–67, 2001.
- [Val95] R. Valk. Petri nets as dynamical objects. In *Workshop on Object-Oriented Programming and Models of Concurrency, Torino, June 1995, Proceedings*, 1995.
- [Val98] R. Valk. Petri nets as token objects: An introduction to elementary object nets. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998, 19th International Conference, ICATPN '98, Lisbon, Portugal, June 22-26, 1998, Proceedings*, volume 1420 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 1998.
- [Val00] R. Valk. Relating Different Semantics for Object Petri Nets. Fachbereichsbericht FBI-HH-B-266/00, Fachbereich Informatik, Universität Hamburg, 2000.
- [Vau85] J. Vautherin. *Un Modele Algebrique, Base sur les Reseaux de Petri, pour l'Etude des Systemes Paralleles*. These de Docteur Ingenieur, Université de Paris-Sud, Centre d'Orsay, June 1985.
- [Vau87] J. Vautherin. Parallel systems specifications with coloured Petri nets and algebraic specifications. In G. Rozenberg, editor, *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets, Oxford, UK, June 1986*, volume 266 of *Lecture Notes in Computer Science*, pages 293–308. Springer-Verlag, 1987.
- [vBJ93] L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105:30–41, 1993.
- [vBJMP93] L. S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 19–61. Springer-Verlag, 1993.
- [Vir] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*. Special Issue on Rewriting Logic and its Applications. To appear.
- [Vir95] P. Viry. Rewriting modulo a rewrite system. Technical Report TR-95-20, Dipartimento di Informatica, Università di Pisa, 1, 1995.

- [Vir96a] P. Viry. Input/output for ELAN. In *RWLW'96, First International Workshop on Rewriting Logic and its Applications, Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996, Proceedings*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [Vir96b] P. Viry. A rewriting implementation of pi-calculus. Technical Report TR-96-30, Dipartimento di Informatica, Università di Pisa, 26, 1996.
- [Vog97] W. Vogler. Partial order semantics and test arcs. In *Mathematical Foundations of Computer Science 1997, 22nd International Symposium, MFCS'97, Bratislava, Slovakia, August 25-29, 1997, Proceedings*, volume 1295 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- [WB89] P. Wadler and S. Blott. How to make ad hoc polymorphism less ad hoc. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages, Austin, Texas, January 1989*, pages 60–76. ACM Press, 1989.
- [Wec92] W. Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.
- [Wer97] B. Werner. Sets in types, types in sets. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS'97, Sendai, Japan, September 23–26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- [Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. North-Holland, 1990.
- [WMG00] B.-Y. Wang, J. Meseguer, and C. A. Gunter. Specification and formal verification of a PLAN algorithm in Maude. In T. Lai, editor, *Proceedings of the 2000 ICDCS workshop on Internet 2000, Distributed Real-Time Systems, Group Computation and Communications, Wireless Networks and Mobile Computing, Distributed System Validation and Verification, Knowledge Discovery and Data Mining in the World Wide Web*, pages E:49–E:56. IEEE, April 2000.

- [Won99] W. Wong. Validation of HOL proofs by proof checking. *Formal Methods in System Design: An International Journal*, 14(2):193–212, 1999.
- [WR13] A. N. Whitehead and B. Russell. *Principia Mathematica (3 volumes)*. Cambridge University Press, 1910, 1912, 1913.
- [WWV⁺97] M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit, and J. Desel. DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen. Informatik-Bericht 88, Humboldt-Universität zu Berlin, 1997.