THEORY AND IMPLEMENTATION OF AN EFFICIENT TACTIC-BASED LOGICAL FRAMEWORK

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Aleksey Yuryevich Nogin

August 2002

© 2002 Aleksey Yuryevich Nogin

ALL RIGHTS RESERVED

THEORY AND IMPLEMENTATION OF AN EFFICIENT TACTIC-BASED LOGICAL FRAMEWORK

Aleksey Yuryevich Nogin, Ph.D.
Cornell University 2002

Formal methods are successfully used in a wide range of applications — from hardware and software verification to formalizing mathematics and education. However their impact is limited and is very far from realizing the full potential of formal methods. Our investigation of these limitations shows that they can not be avoided by simply fine-tuning existing tools and methods. We demonstrate the need to concentrate on improving and extending the underlying theory and methodology of computer-aided formal reasoning.

This thesis explores both practical and theoretical aspects of achieving these improvements; we present solutions for some of the outstanding problems and substantial improvements for others. In particular, we improve axiomatizations of the extant logics to make them more accessible to both users of the system and proof automating procedures. We also present methods for very significant speedup of the proof search process. Such additional speed means not only that the theorem prover will work faster, but also users can now take advantage of more advanced proof automation procedures that would have been prohibitively slow otherwise.

This thesis also demonstrates how these wide ranging practical and theoretical results can be brought together in a more efficient and more generic tactic-based

formal system. In particular, we present a generic derived rules mechanism and explain how such a mechanism can facilitate practical modularization of a formal system. We also present several approaches to establishing a generic layer of proof automation procedures (*tactics*) that apply to a wide variety of logical theories. Most of the ideas presented in this thesis were implemented in the MetaPRL logical framework taking advantage of an existing modular flexible design and making it even more modular and more flexible.

After implementing these ideas in the MetaPRL logical framework, we use that system to improve a formalization of NuPRL intuitionistic type theory. In particular, we show how to modularize an axiomatization of a quotient type thus creating a formalization capable of expressing important concepts that were impossible to express in the original monolithic axiomatization. We also show how to add a limited form of classical reasoning to an intuitionistic type theory in a way that preserves many constructive aspects of the theory. Several theorems in this thesis were formally proven in the MetaPRL system.

BIOGRAPHICAL SKETCH

Aleksey Nogin was born in 1976 in Moscow. In 1987 he started participating in the Evening Mathematical School classes at Moscow State Fifty-Seventh School and in 1989 he entered the mathematics class there.

In 1991 Aleksey won a diploma of the first degree ("gold medal") in the all-USSR High School Mathematical Olympiad. In 1992 Aleksey won a silver medal in the 33rd International Mathematical Olympiad.

In 1992 Aleksey graduated from the Fifty-Seventh School and was accepted into the Department of Mechanics and Mathematics in Moscow State University. There in 1994 he chose the Mathematical Logics and Theory of Algorithms specialization under professor Alexander Razborov.

In 1997 Aleksey received an Honors Diploma from Moscow State University and joined the Cornell University department of Computer Science Ph.D. program.

ACKNOWLEDGEMENTS

This thesis would not have been possible without my adviser Professor Robert Constable whose enthusiasm and love of the research were the biggest influence in my choosing Cornell graduate school and a great inspiration throughout my studies at Cornell. The value of his guidance in making me a better researcher is hard to overestimate.

This thesis would also not have been possible without my colleagues and friends Jason Hickey, who laid all the groundwork for the MetaPRL system and have been extremely helpful and supportive once I have joined the MetaPRL project; and Alexei Kopylov, who is always eager to join me in figuring out even the most intricate sides of type theory.

There is a special place in my heart for Sergei Artemov — a father, a colleague and a friend. His guidance helped me find a research area I love; he is always around when I need his advice.

I would also like to thank Mark Bickford, Stuart Allen, Lori Lorigo, Vladimir Krupski, Yegor Bryukhov and many others for all the great discussions.

I am very grateful to my committee members Greg Morrisett who was great at providing me with a new way of looking at my research and Ulric Neisser who was great at giving me a chance to look outside of my research.

I would also like to thank my office-mates Jim Ezick, Kiri Wagstaff and Yaron Minsky for bringing fun and diversity into my days in Upson Hall.

Finally, I would like to thank DARPA (grant F30602-98-2-0198), ARFL (grants F30602-98-2-0198 and F49620-00-1-0209) and ONR (grant N00014-01-1-0765) for sponsoring most of this research.

TABLE OF CONTENTS

1 Introduction **1**
- 1.1 Tactic-Based Interactive Theorem Provers . . . 1
- 1.2 Modular Theorem Provers and Logical Frameworks . . . 2
- 1.3 Structure of the Thesis and Overview of Contributions . . . 5
- 1.4 MetaPRL . . . 7
- 1.5 Conclusion . . . 8

I Generic Formal Reasoning **9**

2 Speeding up Tactic-Based Theorem Proving **10**
- 2.1 Architectural Overview . . . 11
- 2.2 The *Term* Module . . . 16
  - 2.2.1 Naive Term Implementation (`Term_std`) . . . 17
  - 2.2.2 Delayed Substitution (`Term_ds`) . . . 19
- 2.3 The *Rewriter* module . . . 23
- 2.4 Performance . . . 27
- 2.5 Summary . . . 31
- 2.6 Related work . . . 32

3 Sequent Schemata for Derived Rules **33**
- 3.1 Terms and Sequents . . . 36
- 3.2 Term Schemata . . . 38
- 3.3 Language of Sequent Schemata . . . 39
- 3.4 Semantics — Sequent Schemata . . . 41
- 3.5 Rule Specifications . . . 45
- 3.6 Conservativity . . . 48
- 3.7 Extending the Language of Sequent Schemata . . . 52
- 3.8 Related Work . . . 54

4 Logical Meta-Language: From Derived Rules to Tactics **56**
- 4.1 Syntax Simplifications . . . 57
- 4.2 MetaPRL Rule Specifications . . . 58
- 4.3 Rule Annotations . . . 61
- 4.4 Decision Procedures as Heuristics . . . 63
  - 4.4.1 JProver . . . 65
  - 4.4.2 Arithmetic . . . 66
- 4.5 Generic tactic layer . . . 66

II Type Theory **68**

5 Quotient Types — A Modular Approach **69**
5.1 NuPRL Type Theory 71
5.1.1 Propositions-as-Types 71
5.1.2 Partial Equivalence Relations Semantics 72
5.1.3 Extensional and Intensional Approaches 73
5.2 Squash Operator 74
5.2.1 Squash Operator: Introduction 74
5.2.2 Squash Operator: Axioms 75
5.2.3 Squash Operator: Derived Rules 76
5.3 Choosing the Rules 77
5.4 Intensional Set Type 79
5.4.1 Set Type: Introduction 79
5.4.2 Set Type: Traditional Approach 79
5.4.3 Set Type: A New Approach 80
5.5 Extensional Squash Operator (Esquash) 81
5.5.1 Esquash Operator: Introduction 81
5.5.2 Esquash Operator: Axioms 82
5.5.3 Esquash Operator: Derived Rules 83
5.6 Explicit Nondeterminicity 83
5.6.1 Explicit Nondeterminicity: Introduction 83
5.6.2 Explicit Nondeterminicity: Axioms 85
5.6.3 Explicit Nondeterminicity: Derived Rule 85
5.7 Intensional Quotient Type 85
5.7.1 Quotient Type: Introduction 85
5.7.2 Intensional Quotient Type: Axioms 86
5.7.3 Intensional Quotient Type: Derived Rules 87
5.8 Indexed Collections 88
5.8.1 Indexed and Predicated Collections 88
5.8.2 Collections: the Problem 89
5.8.3 Collections: a Possible Solution 90
5.9 Related Work 91

6 Functionality and Equality in Type Theory **93**
6.1 Introduction to Equality Relations in Type Theory 93
6.1.1 Subtyping and Extensional Equality 93
6.1.2 Intersection Type 95
6.1.3 Functionality Semantics of Sequents 96
6.2 The Gap 97
6.3 Functionality Structures 99
6.4 Non-standard Model 100
6.4.1 A Non-standard Functionality Structure 100
6.4.2 A Potential Counterexample 103

6.4.3 Completing the Model 103
6.4.4 Other Interesting Non-standard Models 104
6.5 Uniquely Defining a Type by its Equality Relation 105

7 Propositional Markov's Principle for Type Theory 107
7.1 Introduction 107
7.1.1 Markov's Constructivism 108
7.1.2 Markov's Principle 110
7.1.3 Type Theory 110
7.2 Constructive Recursive Mathematics in a Type Theory with a Membership Type 113
7.3 Squashed Types and Squash-Stability 114
7.4 Classical Reasoning on Squashed Types 115
7.5 Semantical Consistency of Markov's Principle 119
7.6 Squash Operator as a Modality 120
7.7 Related Work 122

A Appendix 124
A.1 Some Type Theory Rules 124
A.1.1 Structural Rules 124
A.1.2 Membership Rule 124
A.1.3 Disjunction Rules 125
A.1.4 Universal Quantifier Rules 125
A.1.5 Existential Quantifier Rules 125
A.1.6 Falsum Rules 126
A.1.7 Computation Rules 126
A.1.8 Arithmetical Rules 126
A.2 NuPRL-4 Quotient Rules 127
A.2.1 Comparison to the Proposed Rules 128
A.3 Concrete Syntax of Sequent Schemata 129

Bibliography 130

LIST OF TABLES

2.1 Refiner Interface 14
2.2 Term Syntax Examples 16
2.3 Signature for the Term Module 17
2.4 Free Variables Computation in `Term_ds` 21
2.5 Substitution Expansion for `Term_ds` 23
2.6 Term Expansion for `Term_ds` 23
2.7 Interface for the Rewriter Module 24
2.8 Virtual Machine Instructions 26
2.9 Virtual Machine Example: $(\lambda x.b_x)\ a \longrightarrow b_a$ 27
2.10 Performance on the Factorial Examples 28
2.11 Performance on the Pigeon-Hole Examples 29

3.1 Language of Sequent Schemata 40

4.1 Decomposition Tactic Examples 61

A.1 Concrete Syntax of Sequent Schemata 129

LIST OF FIGURES

1.1 General Logical Framework Structure 3

2.1 General Tactic Prover Architecture 12
2.2 Layered Refiner Modules 15
2.3 Rewriting Virtual Machine 25

Chapter 1

Introduction

Formal methods are successfully used in a wide range of applications — from hardware and software verification to formalizing mathematics and education. However their impact is often limited and is very far from realizing the full potential of formal methods. Our investigation of these limitations shows that they can not be avoided by simply fine-tuning existing tools and methods. Instead, we concentrate on improving and extending the underlying theory and methodology of computer-aided formal reasoning.

This thesis contributes in four areas. We present methods for very significant speedup of the proof search process. We describe a generic derived rules mechanism that can facilitate practical modularization of a formal system. We outline several approaches to establishing a generic layer of proof automation procedures (*tactics*) that apply to a wide variety of logical theories. We show how to modularize an axiomatization of a quotient type, making it substantially more expressive.

1.1 Tactic-Based Interactive Theorem Provers

The research in the area of automated reasoning is mostly proceeding in the two major directions. First, there are provers (such as ACL2 [KM97], EQP [McC97], Gandalf [Tam97], Otter [McC94], SETHEO [LSBB92], SPASS [Wei97] and TPS [ABI+96]) that typically work in relatively weak logics (first-order or even quantifier-free) attempting to find a proof in a fully automated way in the smallest amount of time. Second, there are provers (such as ALF [CNSvS94, MN94], Coq [BBC+96], HOL [GM93], Isabelle [PN90, Pau94], MetaPRL [Hic97, Hic01, HNK+],

NuPRL [CAB+86, ACE+00] and PVS [ORS, SORSC99]) that use higher-order logics and only attempt to do some tasks automatically while relying on interactive user guidance for more complex tasks. Following LCF [GMW79], in these semiautomated provers, proof automation is usually coded in a *meta-language* (often a variant of ML) as *tactics*.

In general, the second class of provers is more flexible. The expressivity of the higher-order logics permits concise problem descriptions, and meta-principles that characterize entire classes of problems can be proved and re-used on multiple problem instances. The semi-automated nature of these provers means being able to work in application areas both where the automated search is expected to solve the problem completely as well as where the automated search is only meant to complement the human guidance. Such flexibility makes the higher-order semi-automated provers the ideal framework for developing general methodology of formal reasoning. Consequently this thesis focuses almost exclusively on semi-automated reasoning in higher-order logics (although some of the results are applicable to other types of formal systems as well).

1.2 Modular Theorem Provers and Logical Frameworks

At a very high level, an architecture of a tactic-based theorem prover can be described as shown in Figure 1.1.

The core of the system is its *logical engine*, or *refiner* [Bat79]. It is responsible for performing the individual proof steps (such as applying a single inference rule). Next, there is the lower "support" layer for the logical theories. It usually includes basic meta-theory definitions and possibly some basic proof search mechanisms (such as basic tactics). Finally, at the top of the structure there are the logical

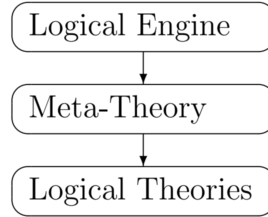

Figure 1.1: General Logical Framework Structure

theories themselves, each potentially equipped with theory-specific mechanisms (such as theory-specific proof search strategies and theory-specific display mechanisms). In a way, the structure of the prover mimics the structure of an operating system with logical engine being the "kernel" of the system, meta-theory being its "system library" and logical theories being its "user space".

We intentionally did not include any user interface in Figure 1.1. The reason for such omission is that often a user interface (such as, for example, NuPRL Editor [MA94, ACE+00] or Proof General [Asp00]) would be a separate package added on top of a formal system, rather then a part of the system itself.

There are two main approaches to building such a prover — one can build a monolithic prover or one can build a modular one. There are several advantages in a more modular architecture, especially in a research environment where we want to work on general methodology.

In a modular system with a well-defined interfaces it is easier to try out new ideas and new approaches. One can start improving a particular module, or even write a completely new one without having to modify (or even understand!) other parts of the system. This allows for a greater flexibility and also helps bringing new people (including new students) to the project.

As we will see in Chapter 2, the modular architecture also allows one to have

several implementations of some critical module. For example, it is possible to have a generic implementation and at the same time create an alternative implementations of some modules that are optimized towards a particular class of applications. This approach is especially useful in the trusted core of the system — there we can have a simple "reference" implementation that is extensively tested and checked for correctness as well as one (or more [1]) highly optimized implementations. Users can develop proofs using the optimized modules and then later double-check them by re-running (in off-line mode) using the reference implementation. This provides the confidence of knowing that proofs were accepted by *both* implementation of the module.

Similarly to the modularity of the logical engine of a formal system, the modularity of the logical theories supported by a system is also important. Some provers only support reasoning in a single monolithic logical theory, while others (often called *logical frameworks* [Pfe01b]) not only give their users a choice of what logical theory to use, but also allow users to add their own logical theories to the system.

Since this thesis aims at improving the general methodology that can be used in a wide variety of applications, we want to concentrate our efforts on most flexible and most general kind of theorem provers. As a result, this thesis primarily focuses on modular tactic-based logical frameworks.

Throughout the thesis (and especially in Chapters 2 and 4) we will continue discussing the structure of the system and we will provide a more detailed picture of an efficient architecture for a generic tactic-based logical framework.

[1] In fact, in our MetaPRL system some of the most performance-sensitive modules have up to 6 different implementations.

1.3 Structure of the Thesis and Overview of Contributions

In this thesis we assume that we already have a formal system that implements a modular architecture similar to the one presented above. We explore ways of improving such a system, of making it more efficient and more usable. Going from the most low-level functionality of the system to the most high-level, we consider each of the modules one at a time, separately analyzing efficiency and usability challenges in each of them. Consequently the structure of this thesis closely follows the general structure of a theorem prover.

We start in Chapter 2 by focusing on the logical engine, primarily on the term operations and the rewriting engine. There the biggest challenge is efficiently — we want to make sure that the basic term operations and rewriting steps are as fast as possible. We will show how we achieve a major speed-up of the logical inference engine (over two orders of magnitude on most examples, when compared to NuPRL-4 [2]). Such speed-up not only means that proof search take less time, but also that more advanced proof search techniques that were prohibitively slow in a slower system are now accessible to users.

In Chapter 3 we explore some theoretical aspects of the meta-theory used. We present a language of *sequent schemata* that gives us a mechanism for adding derived rules to the system in a way that is guaranteed to be conservative, no matter what logical theory is going to be used. Such a mechanism significantly increases the modularity and flexibility of the theorem prover. In addition, our sequent schemata language is designed to be very intuitive and easy to use. In particular, it eliminated the need for explicitly specifying which substitutions have to occur

[2] The reason for this level of speed is not yet fully understood. In particular, it is not completely clear which part of the speed-up comes from our effort and which comes from other factors.

when a rule is applied; this removed a very common source of user mistakes.

In Chapter 4 we discuss some challenges and advantages of implementing the sequent schemata language in a theorem prover and discuss the establishment of a layer of "generic" tactics that can be used across several different logical theories. We also present two approaches to reducing the burden of maintaining complex tactics.

While the first part of the thesis focuses on the general issues of formal reasoning, in the second part we switch to the *logical theories* layer of a theorem prover. Some of the most powerful and challenging logical theories implemented in theorem provers are various flavors of constructive type theory. In this thesis we concentrate on type theories derived from the Martin-Löf type theory [ML82], such as NuPRL type theory [CAB+86] (although many results are applicable to other versions of type theory and possibly other constructive theories as well).

In Chapter 5 we discuss the challenges of coming up with the best formalization for the quotient types in type theory. We provide a new highly modular axiomatization of the quotient type and explain the advantages of such modularization. In particular, we show how the new axiomatization allows us to express many important concepts that were impossible to express in the original monolithic axiomatization. We also discuss how the choices we make in the axiomatization make it more accessible and more usable, especially in the context of a theorem prover.

In Chapter 6 we discuss the types and their equality relations in type theory. We show how two types may be different despite having the same members and the same equality relation and how this leads to some intuitively very simple and seemingly correct statements being virtually unprovable. We demonstrate how to eliminate this problem by explicitly stating that any two types with the same

equality relation should be considered equal.

Finally, in Chapter 7 we show how a constructive type theory can be extended by adding a useful limited classical reasoning without sacrificing the constructive nature of the theory. We show that our extension is a generalization of Markov's principle and can be expressed in a purely propositional fragment of type theory.

1.4 MetaPRL

As a main testbed for the ideas presented in this thesis we chose MetaPRL— a modern modular tactic-based formal system. Most of the ideas presented in this thesis were implemented and successfully used in the system. In particular, all the type theory derivations presented in Part II of the thesis were developed with the help of MetaPRL and were formally checked by the system.

MetaPRL is the latest out of many formal tools developed over a period of more than 20 years by the PRL group at Cornell. [3] MetaPRL is not only a tactic-based interactive proof assistant, but it is also *a logical framework* that allows users to specify their own logical theories to reason in rather than forcing them to use a single theory. Additionally, MetaPRL is a logical programming environment that incorporates many features to simplify reasoning about programs being developed. In fact, MetaPRL is implemented as an extension of the OCaml compiler [WL99]. Finally, MetaPRL can be considered *a logical toolkit* that exports not only the "high-level" logical interface, but all the intermediary ones as well. This allows for rapid development of new logical applications without having to spend time on re-coding the basic formal functionality.

[3] See also Cornell PRL Project home page at `http://www.cs.cornell.edu/Info/Projects/NuPrl/Nuprl.html`.

MetaPRL's predecessor NuPRL [CAB+86] was successfully used for verification and automated optimization of the Ensemble group communication toolkit [LKvR+99]. This toolkit [Hay98] is being used for both military and commercial applications. Its users include BBN, Nortel Networks and NASA. MetaPRL project was started by Jason Hickey [Hic97] as a part of Ensemble verification effort to simplify formal reasoning about the program code and to address scalability and modularity limitations of NuPRL. As more effort was put into the system (with the work presented in this thesis being a big part of that effort) MetaPRL eventually grew into a very general modern system whose modularity and *logical framework* nature give it flexibility to support a very wide range of applications.

The logical theories formalized in MetaPRL include first-order logic, several variations of the NuPRL type theory [CAB+86], and Aczel's CZF set theory [Acz86].

1.5 Conclusion

In this thesis we did our best to extend the frontier on the formal methods in several important directions — greatly improved logical speed, additional flexibility of the derived rules mechanism, automated approaches to maintaining complex tactics, richer and more accessible logical theories. Now this new frontier needs to be explored. We hope that in the near future theses new ideas and their implementations will be extensively tested in applications. In turn, such exploration is likely to discover new unforeseen challenges and we are looking forward to battling them.

Part I

Generic Formal Reasoning

Chapter 2

Speeding up Tactic-Based Theorem Proving

In a tactic-based prover, automation speed has a direct impact on the level of reasoning. If proof search is slow, more interactive user guidance is needed to prune the search space, leading to excessive detail in the tactic proofs.

In this chapter we present a proving architecture that addresses the problem of speed and customization in tactic provers. We have implemented this architecture in the MetaPRL logical framework, achieving on several examples more than two orders of magnitude speed-up over the existing NuPRL-4 implementation. We obtain the speedup in two parts: our architecture is modular, allowing components to be replaced with domain-specific implementations, and we use efficient data structures to implement the proving modules.

One might think that the comparison between MetaPRL and NuPRL-4 is not very informative since NuPRL-4 uses interpreted ML and MetaPRL is implemented in OCaml. But in fact only very high-level code uses interpreted ML in NuPRL-4 while most of the time is spent performing low-level operations such as term operations and primitive rule applications. And in NuPRL-4 all the low-level operations are implemented in Lisp and are compiled by a modern Lisp compiler. This should make the comparisons relatively informative, especially when there is some evidence that the speedup might be over two orders of magnitude.

It should also be noted that MetaPRL is a *distributed* prover [Hic99], leading to additional speedups if multiple processors are used. Distribution is implemented by inserting a scheduling and communication layer between the refiner and the

tactic interface. For this work, we describe operation and performance without this additional scheduling layer.

The organization of the chapter is a follows. In Section 2.1, we revisit the prover architecture presented in Section 1.2 and show that the logic engine can be broken into three modules: a *term* module that implements the logical *language*, a term *rewriter* that applies primitive inferences, and a *proof* module that manages proofs and defines tactics.

In Sections 2.2 and 2.3 we explore the logical engine in more detail, and develop its implementations. The computational behavior of proof search is dominated by term rewriting and operations on terms. We present implementations of the modules for domains with frequent applications of substitution (like type theory), and for domains with frequent applications of unification (like first-order logic).

In Section 2.4, we compare the performance of the different implementations. We include performance measurements that compare MetaPRL's performance with NuPRL-4 on the NuPRL type theory. In our measurements, we also show how particular module implementations change the performance in different domains.

In Section 2.5 we summarize our results, and discuss the remaining issues. This work builds on the efforts of many systems over the last decade, and in Section 2.6 we give an overview of related work.

Some of the work presented in this chapter is joint with Jason Hickey — see also [HN00].

2.1 Architectural Overview

We consider a general architecture of a tactic prover consisting of three parts presented in Section 1.2 (Figure 1.1). A refinement of that architecture is shown

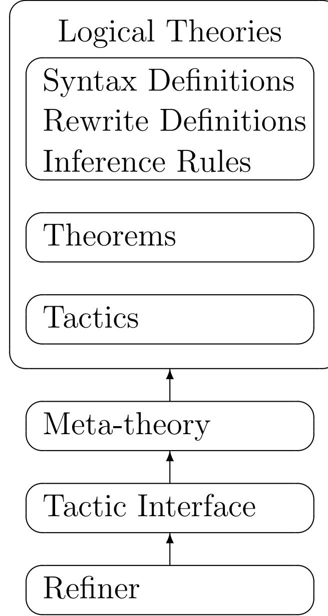

Figure 2.1: General Tactic Prover Architecture

in Figure 2.1. At the center of the system we have *logical theories* (or simply *logics*) that contain the following kinds of objects:

1. *Syntax definitions* define the *language* of a logic,

2. *Inference rules* define the primitive inferences of a logic. For instance, the first-order logic contains rules like MODUS_PONENS in a sequent calculus.

$$\frac{}{\Gamma, A \vdash A}\ \text{AXIOM} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \text{MODUS_PONENS}$$

3. *Rewrites* define computational equivalences. For example, the type theory defines functions and application, with the equivalence $(\lambda x.\, b[x])\ a \longleftrightarrow b[a]$.

4. *Theorems* provide proofs for derived inference rules and axioms.

5. *Tactics* provide theory-specific proof search automation.

The core of the system is its *logical engine* or *refiner* [Bat79] that performs two basic operations. First, it builds the basic proof procedures from the parts of a logic.

1. Syntax definitions are compiled to functions for constructing logical formulas.

2. Rewrite primitives (and derived rewrite theorems) are compiled to *conversions* that allow computational reductions to be applied during a proof.

3. Inference rules and theorems are compiled to primitive *tactics* for applying the rule, or instantiating the theorem.

The second refiner operation is the *application* of conversions and tactics, producing justifications from the proofs. The major parts of the refiner interface are shown in Table 2.1. [1] It defines abstract types for data structures that implement terms, tactic and rewrite definitions, proofs, and logics. Proof search is performed in a backward-chaining goal-directed style. The `refine` function takes a `logic` and a `tactic` search procedure, and applies it to a *goal* term to produce a *partial* `proof`. The goal and the resulting *subgoals* can be recovered with the `sub/goal_of_proof` projection functions. Proofs can be composed with the `compose proof subproofs` function, which requires that the goals of the `subproofs` correspond to the subgoals of the `proof`, and that both derivations occurred in the same logic. If an error occurs in any of the refiner functions, the `RefineError` exception is raised. The `tactic_of_conv` function creates a tactic from a rewrite definition. The final two functions, called *tacticals*, are the primitives for implementing proof search. Operationally, the `andthen tac1 tac2` tactic applies `tac1` to a goal and immediately applies `tac2` to all the subgoals, composing the result. The `orelse tac1`

[1] Throughout this chapter we will use a simplified OCaml syntax to give the component descriptions.

Table 2.1: Refiner Interface

```
module type RefinerSig = sig
   type term, tactic, conv, proof, logic
   exception RefineError
   val refine :  logic → tactic → term → proof
   val goal_of_proof :  proof → term
   val subgoals_of_proof :  proof → term list
   val compose :  proof → proof list → proof
   val tactic_of_conv :  conv → tactic
   val andthen :  tactic → tactic → tactic
   val orelse :  tactic → tactic → tactic
end
```

`tac2` is equal to `tac1` on goals where `tac1` does not produce an error, otherwise it is equivalent to `tac2`.

The `logic` data type is the concrete representation of a logic. The MetaPRL logical framework defines multiple logics in an inheritance hierarchy (partial order) where if L_{child}: `logic` inherits from L_{parent}: `logic`, all the theorems of L_{parent} are valid (and provable) in L_{child}. In contrast, the NuPRL-4 prover has a single global logic containing the syntax and rules of the NuPRL type theory.

In a prover like NuPRL-4, the refiner can be characterized as *monolithic*. There is no well-defined separation of the refiner into components, and there is no well-defined interface like the `RefinerSig` we defined above—there is *one* built-in refiner. This has made it difficult to customize and maintain NuPRL-4, and our choice in MetaPRL has been to partition the refiner into several small well-defined parts.

As we have mentioned in the introduction, this modular structure has an additional benefit: if we partition the refiner into abstract parts, we can create domain-specific implementations of its parts. While the whole refiner is a part of a trusted

code base, we are still protected from bugs in domain-specific optimization. When we need to be extra sure that everything is correct, we can do proof development using the domain-specific code and later double-check the proof using the reference implementation. And for some parts of the system we even have a debugging mode that runs two implementations side-by-side and notifies the user if they behave differently. This not only protects us from bugs introduced by the domain-specific code, but also helps us to debug the reference implementation as well.

The choice of partitioning we use is guided by the *type* definitions, producing the layered architecture shown in Figure 2.2. The lowest layer, the `Term` module, implements the basic logical *language* and basic syntactical operations (such as substitution and alpha equality testing). The `Rewriter` module provides a mechanism for making complex syntactical transformations. The `Rewriter` does not assign the logical meaning to transformation it makes, in fact it can be used to do both formal transformation (such as applying a rule to a statement being proven) and informal ones (such as converting a formal statement into a string to be displayed to the user). Finally, a *Proof* accounting module keeps track of what rules were included in a particular logical theory being used, what was proven using those rules and what is still left to prove.

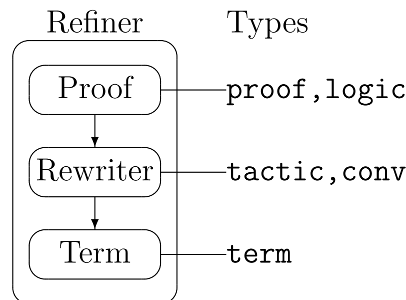

Figure 2.2: Layered Refiner Modules

Table 2.2: Term Syntax Examples

Displayed form	Term
1	`number[1]{}`
$\lambda x.b$	`lambda[]{x. b}`
$f(a)$	`apply[]{f; a}`
v	`variable["v"]{}`
$x+y$	`sum[]{x; y}`

2.2 The *Term* Module

All logical terms, including goals and subgoals, are expressed in the language of *terms*, implemented by the *term* module. The general syntax of all terms has three parts. Each term has:

1. An operator name (like "sum"), which is a unique name indicating the logic and component of a term;

2. A list of parameters representing constant values; and

3. A set of subterms with possible variable bindings.

We use the following syntax to describe terms, based on the NuPRL definition [ACHA90]:

$$\underbrace{opname}_{operator\ \ name}\ \underbrace{[p_1; \cdots ; p_n]}_{parameters}\underbrace{\{\vec{v_1}.t_1; \cdots ; \vec{v_m}.t_m\}}_{subterms}$$

A few examples are shown in Table 2.2. Variables are terms with a string parameter for their name; numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

The term module implements several basic term operations: substitution ($b[a/x]$) of a term (a) for a variable (x) in a term (b), free-variable calculations, α-equivalence,

Table 2.3: Signature for the Term Module

```
module type TermSig = sig
   (* Types and constructors:  *)
   type opname, param, term, bound_term
   val mk_opname :  string list -> opname
   val mk_int_param :  int -> param
   val mk_string_param :  string -> param
   val mk_term :  opname -> param list -> bound_term list -> term
   val mk_bterm :  string list -> term -> bound_term

   (* Destructors and other operations:  *)
   val dest_term :  term -> opname * param list * bound_term list
   val dest_bterm :  bound_term -> string list * term
   val subst :  (string * term) list -> term -> term
   val free_vars :  term -> string list
   val alpha_equal :  term -> term -> bool
end
```

etc. When a logic defines a rule, the refiner compiles the rule pattern into a sequence of term operations. The term interface is shown in Table 2.3. The abstract types `opname`, `param`, `term`, and `bound_term` represent operator names, constant parameters, terms, and bound terms (the subterms of a term). The major operations include destructors to decompose terms and bound-terms, as well as a substitution function `subst`, free variable calculations, and term equivalence.

2.2.1 Naive Term Implementation (`Term_std`)

The most immediate implementation of terms is the naive "standard" implementation, which builds the term with tupling:

```
type opname = string list
and param = Int of int | String of string
and term = opname * param list * bound_term list
and bound_term = string list * term
```

While this structure is easy to implement, it suffers from the poor performance of the substitution algorithm. The following pseudo-code gives an outline of that algorithm.

```
let rec subst sub t =
   if t is a variable then
      if (t, t') ∈ sub then t' else t
   else let (opname, params, bterms) = t in
      (opname, params, List.map (subst_bterm sub) bterms)
and subst_bterm sub (vars, t) =
   let sub' = remove (v, t') from sub if v ∈ vars in                (2.1)
   let vars', sub'' = rename binding variables to avoid capture in  (2.2)
   (vars', subst sub'' t)
```

The `sub` argument is a list of `string/term` pairs that are to be simultaneously substituted into the term given by the second argument. The main part of the substitution algorithm is in the part for substituting into bound terms. In step (2.1), the substitution is modified by removing any `string/term` pairs that are freshly bound by the binding list `vars`, and in step (2.2), the binding variables are renamed if they intersect with any of the free variables in the terms being substituted.

This algorithm takes time roughly linear in the size of the term on which

the substitution is performed. Furthermore, each substitution copies each term fully. Substitution is a very common operation in MetaPRL— each application of an inference rule involves at least one substitution. [2] The next implementation performs lazy substitution, useful in domains like type theory.

2.2.2 Delayed Substitution (`Term_ds`)

If substitution is frequently performed on the same terms over and over again, it is often more efficient to save intermediate computations for use in future substitution operations. We use three main optimizations: we save free-variable calculations, we perform lazy substitution, and we provide special representations for commonly occurring terms.

When a substitution is performed on a term for the first time, we compute the set of free variables of that term, and save them for later use. When a substitution is applied, the free-variables set is used to discard the parts of the substitution for variables that do not occur free in the term. This saves time, and it also saves space by reusing subterms where the substitution has no effect instead of unnecessarily copying them. Memory savings, in turn, further improve performance by improving the CPU cache efficiency and reducing the garbage collection time.

During proof search, most tactic applications fail, and only a part of the substitution result is usually examined in the proof search. In this common case, it is more efficient to delay the application of a substitution until the substitution

[2] Testing for α-equivalence also takes linear time. One way to decrease the cost would be to use a normalized representation (such as a DeBruijn representation). However, term *destruction* on the normalized representation can be expensive because of the need to rename variables that become free (what are the subterms of $\lambda x.\lambda y.xy$ and $\lambda x.\lambda y.yx$?). These renamings can be delayed, as the next Section shows, but the cost of equivalence testing will increase.

results are actually requested by the `dest_term` function.

In `Term_ds` we also optimize two commonly-occurring terms: *variables* and *sequents*. Rather than using the term encoding of variables, we provide a custom representation using a string. The *sequent* optimization uses a custom data structure to give constant-time access to the hypotheses, instead of the usual linear-time encoding. These "custom" terms are abstract optimizations—they do not change the `Term` interface definition. For each custom term, we add special-case handlers to each of the generic term functions in the module.

The following definition of terms uses all of these optimizations (the definitions for the `bound_term`, `opname` and `param` types are unchanged). The definition of sequents, which we omit, uses arrays to represent the hypotheses and conclusions of the sequent.

```
type term =
   { free_vars : VarsDelayed
               | Vars of string set;
     core : Term of (opname * param list * bound_term list)
          | Subst of (subst * term)
          | Var of string
          | Sequent of sequent }
and subst = (string * term) list
and sequent = ···
```

The `free_vars` field caches the free variables of the term, using `VarsDelayed` as a placeholder until the variable set is computed. The `core` field stores the term value, using the `Term` variant to represent values where a substitution has been expanded, the `Subst` variant to represent delayed substitutions, and the `Var`

Table 2.4: Free Variables Computation in `Term_ds`

```
let rec free_vars = function
   { free_vars = Vars fv } -> fv
 | { core = core } as t ->
      let fv = match core with
         Term (_, _, bterms) -> Set.map_list free_vars_bterm bterms
       | Subst (sub, t) -> free_vars_subst sub (free_vars t)
       | Var v -> Set.singleton v
       | Sequent seq -> free_vars_sequent seq
      in (t.free_vars ← Vars fv); fv

and free_vars_bterm (bvars, t) =
   Set.subtract_list (free_vars t) bvars
and free_vars_subst sub fv =
   Set.union
      (Set.subtract_list fv (List.map fst sub))
      (Set.map_list free_vars (List.map snd sub))
```

and `Sequent` variants for custom terms. We maintain the following invariants on `Subst`: substitution lists are never empty, and the domain of the substitution is included in the free-variables of the term.

The free-variables computation is one of the more complex operations on this data structure. As shown in Table 2.4, when the free variables are computed for a term, there are three main cases: if the free variables have already been computed, they are returned; if the `core` is a `Term`, the free variables are computed from the subterms; and if the `core` is a delayed substitution, the substitution is used to modify the free variables of the inner term.

If the free variables haven't already been computed, the `free_vars` function computes them, and assigns the value to the `free_vars` field of the term. In the `Term` case, the free variables are the union of the free variables of the subterms, where any new binding occurrences have been removed. In the `Subst (sub, t)` case, the

free variables are computed for the inner term `t`, then the variables being replaced are removed from the resulting set, and then the free variables of the substituted terms are added.

The `subst` function has a simple implementation: eliminate parts of the substitution that have no effect (in order to maintain the invariant), and save the result in a `Subst` pair if the resulting substitution is not empty.

```
let subst sub t =
   let fv = free_vars t in
      match remove (v, t') from sub if v ∉ fv with      (2.3)
         [] -> t (* substitution has no effect *)
       | sub' ->
            { free_vars = VarsDelayed; core = Subst (sub', t) }
```

The `set` implementation determines the complexity of substitution. If the `set` lookup takes $O(1)$, then pruning (2.3) takes time linear in the number of variables in `sub`.

The effect of the substitution is delayed until the term is destructed. The `dest_term` function is required to expand the substitution by one step. We use the `get_core` function, shown in Table 2.5, to expand the top-level substitutions in the term. If the substitution was applied to a `Term`, `get_core` will push it down to the immediate subterms. After the substitution is expanded, `get_core` will store the result in the `core` field to save time on the next `get_core` invocation. As usual, we omit the code for sequents.

Note that the `List.assoc` in the `Var` case will never fail, due to our invariants.

As shown in Table 2.6, the `dest_term` function first uses `get_core` to expand the top-level substitution (if any), and then it returns the parts of the term. To

Table 2.5: Substitution Expansion for `Term_ds`

```
let rec get_core = function
   { core = Subst (sub, t') } as t ->
      let core' = match get_core t' with
         Var v -> get_core (List.assoc v sub) (* always succeeds *)
       | Term (opname, params, bterms) ->
            Term (opname, params,
               List.map (do_bterm_subst sub) bterms)
       | Sequent seq -> Sequent (sequent_subst sub seq)
      in (t.core ← core'); core'
 | { core = simple_core } -> simple_core

and do_bterm_subst sub (vars, t) =
   let sub' = remove (v, t) from sub if v ∈ vars in
   let vars', sub'' = rename binding variables to avoid capture in
      (vars', subst sub'' t)
```

Table 2.6: Term Expansion for `Term_ds`

```
let rec dest_term t = match get_core t with
   Term (opname, params, bterms) -> (opname, params, bterms)
 | Var v -> (mk_opname ["variable"], [String v], [])
 | Sequent s -> dest_sequent s
```

preserve the external interface of the term module, it is also required to convert the custom terms back to their original form.

2.3 The *Rewriter* module

The rewriter performs term manipulations for rule applications. Inference rules and computational rewrites are both expressed using second-order patterns. [3] For

[3] The exact formal semantics of these patterns will be presented in Chapter 3.

Table 2.7: Interface for the Rewriter Module

```
module Rewrite (Term : TermSig) : sig
   type redex_prog, con_prog, state
   exception RewriteError
   val compile_redex : term -> redex_prog
   val apply_redex : redex_prog -> term -> state
   val compile_contractum : redex_prog -> term -> con_prog
   val build_contractum : con_prog -> state -> term
end
```

example, the rewrite for beta-reduction is expressed with the following pattern:

$$(\lambda x.b_x)\ a \longrightarrow b_a$$

In this rewrite, the variable a is a *pattern* variable, representing the "argument" term. The variable b_x is a *second-order pattern* variable, representing a term with a free variable x. The pattern b_a represents a *substitution*, with a substituted for x in b. The $(\lambda x.b_x)\ a$ is called the *redex*, and the substitution b_a is called the *contractum*.

In NuPRL-4 the computation and inference engines are implemented as separate interpreters that are parametrized by the rewriting patterns. In MetaPRL we combine these functions and improve performance by compiling to a rewriting virtual machine. As shown in Table 2.7 the MetaPRL rewriter module provides four major functions. The `compile_redex` function takes a redex pattern, expressed as a term, and it compiles it to a redex *program*. The `apply_redex` function applies a pre-compiled program to a specific term, raising the `RewriteError` exception if the pattern match fails, or returning a `state` that summarizes the result. The `compile_contractum` compiles a contractum pattern against a particular redex program, and the `build_contractum` function takes the contractum program and

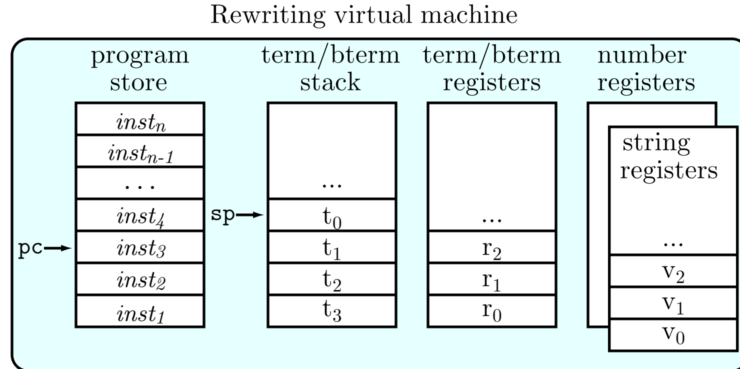

Figure 2.3: Rewriting Virtual Machine

the result of a redex application, and produces the final contractum term.

Currently, the rewrite module compiles redices to bytecode programs that perform pattern matching, storing the parts of the term being matched in several register files. Contracta are also compiled to bytecode programs that construct the contractum term using the contents of the register file. Applying a rewrite will cause corresponding bytecode to be interpreted in a virtual machine. [4]

The virtual machine has the four parts shown in Figure 2.3:

1. a *program* store and program counter for the rewrite program,
2. a *term/bterm stack* with a stack pointer to manage the current term being rewritten,
3. a *term/bterm* register file,
4. a *parameter* register file for each type of parameter.

[4]It may be possible to achieve further speed-up by compiling rewrites instead of using an interpreted bytecode. In [Rhi02, Section 5] and [Rhi01, pages 105–106] Morten Rhiger describes his work on specializing MetaPRL rewriter code for particular rewrites. In this thesis however we only present the interpreted approach.

Table 2.8: Virtual Machine Instructions

Matching		Constructors	
`dest_term`	`opname`$[p_1; \cdots ; p_n].bc$	`mk_term`	`opname`$[p_0; \cdots ; p_n].bc$
`dest_bterm`	$v_1, \ldots, v_n$	`mk_bterm`	$v_1, \ldots, v_n$
`so_var`	$r[v_1; \cdots ; v_n]$	`so_subst`	$r.bc$
`match_term`	$r[t_1; \cdots ; t_n]$		
`not_free`	$v_1, \ldots, v_n$		

p_i: parameter register
v_i: string register
r: term register
t_i: literal term
bc: arity of bterm

The instructions for the machine are shown in Table 2.8. The matching instruction `dest_term` checks if the term at the top of the term stack has the operator name `opname`, and if it has the right number of bound terms and parameters of the given types. If it succeeds, the parameters are saved in the parameter registers, the term is popped from the term stack, and the bound terms are pushed onto the stack. The `mk_term` instruction does the opposite: it retrieves the parameter values from the register file, pops bc bound terms from the stack, adds the `opname` and pushes the resulting term onto the stack. The `dest_bterm` and `mk_bterm` functions are used to save and restore binding variables for the term at the top of the stack.

The `so_var` instruction pops a term from the term stack and saves it in term register r, along with the free variables in $v_1, \ldots, v_n$. The corresponding constructor `so_subst` pops bc terms from the stack, substitutes them for the variables $v_1, \ldots, v_n$ in term r, and pushes the result onto the stack. The `match_term` instruction is used during matching for redices like $(x+x) \rightarrow 2x$ that contain common subterms. Finally, the `not_free` instruction is used to ensure variables $v_1, \ldots, v_n$ do not occur freely in the term on top of the term stack.

Table 2.9: Virtual Machine Example: $(\lambda x.b_x)\ a \longrightarrow b_a$

Redex		Contractum	
`dest_term`	apply[].2	`so_subst`	r_2.0
`dest_bterm`		`so_subst`	r_1.1
`dest_term`	lambda[].1		
`dest_bterm`	v_1		
`so_var`	$r_1[v_1]$		
`dest_bterm`			
`so_var`	$r_2[]$		

The example in the Table 2.9 gives the code for a beta-reduction. The first `dest_term` instruction matches the outermost `apply` term and pushes the function and argument onto the stack. The `dest_bterm` operations remove the binding variables of the subterms, and the `so_var` instructions stores the results to the register file. At the end of a match against the term $(\lambda x.b)\ a$, register r_1 contains b, register r_2 contains a, and register v_1 contains x. When the contractum is constructed, the first instruction pushes a onto the stack; and the second instruction pops a, substitutes it for x in b, and pushes the result.

2.4 Performance

All measurements were done on a Linux 400MHz Pentium machine, with 512MB of main memory, and all times are in seconds. For MetaPRL performance we used MetaPRL version from Spring 2000 with optimizations that are presented in this work, but with neither multi-processor distribution of [Hic99] nor run-time code specialization of [Rhi01, Rhi02]. [5]

We group the performance measurements into two parts. For the first part,

[5] Those two techniques are likely to provide additional performance benefits, but they are outside the scope of this work.

Table 2.10: Performance on the Factorial Examples

Configuration	Argument value			
	100	250	400	650
`Term_std`	0.35	2.05	5.42	16.0
`Term_ds`	0.42	2.41	6.32	18.4
NuPRL-4	55	330	>1800	>1800

we compare the speed of the MetaPRL prover (using the modular refiner) with the NuPRL-4 prover. For the first example, we perform pure evaluation based on the following definition of the factorial function:

$$\text{rewrite } \mathit{fact}\{i\} \longleftrightarrow \mathit{if}\ i = 0\ \mathit{then}\ 1\ \mathit{else}\ i * \mathit{fact}\{i-1\}$$

We used the following evaluation algorithm: recursively traverse the term top-down, performing beta-reduction, unfolding the `fact` definition (taking care to evaluate the argument first), etc. This algorithm stresses search during rewriting. Roughly speaking, evaluation should be quadratic in the factorial argument: each term traversal is linear in the size of the term, and the size of the term grows linearly with each traversal (rewriting does not use tail-recursion), until the final base case is reached and the value is computed. Table 2.10 lists the performance numbers.

On this example, the NuPRL-4 took between 125 and 160 times longer on the problems where it finished within 30 minutes. On the two larger problems, we terminated the computation after 30 minutes. [6] In MetaPRL, the largest problem performs about 14 million attempted rewrites.

This table also shows a difference between the term module implementations. The "naive" term module performs better on this example because the recursive

[6] NuPRL-4 *can* evaluate these terms. The built-in term evaluator, which bypasses the refiner, evaluates the largest example in about 22 seconds.

Table 2.11: Performance on the Pigeon-Hole Examples

		Problem size			Memory
Configuration	Tactic	2	3	4	(Max MB)[8]
`Term_std`	`pigeonT`	<0.1	2.53	94.0	126
`Term_ds`	`pigeonT`	<0.1	0.71	17.0	20.8
NuPRL-4	`pigeonT`	0.5	89	>1800	
`Term_std`	`propDecideT`	0.3	238	>1800	
`Term_ds`	`propDecideT`	0.13	55.0	>1800	
NuPRL-4	`propDecideT`	21.9	>1800	>1800	

traversals of the term expand most of the delayed substitutions.

The next example also compares MetaPRL with NuPRL-4, on the pigeonhole problem stated in propositional logic. [7] The pigeonhole problem of size i proves that $i+1$ "pigeons" do not fit into i "holes." The `pigeonT` tactic performs a search customized to this domain, and the `propDecideT` tactic is a generic decision procedure for *intuitionistic* propositional logic (based on Dyckoff's algorithm [Dyc92]). Both search algorithms use only propositional reasoning and both explore an exponential number of cases in i.

As shown in Table 2.11, in this example `Term_ds` works between 125 and 170 times faster than NuPRL-4. And the delayed-substitution implementation of terms performs significantly better than the naive implementation, partly because of the efficient substitution in the application of the rules for propositional logic, and also because the `Term_ds` module preserves a great deal of sharing of common subterms. On the largest problem the `pigeonT` tactic performs about 1.57 million primitive

[7] This formalization of pigeon-hole principle and methods we are using to prove it are obviously highly inefficient. However this formalization provided us with a nice way of comparing the performance of simple propositional proof search in the two systems.

[8] This does not include the space that the system occupied after the initial loading — 19 MB with `term_std` and 20.5 MB with `term_ds`

inference steps.

For the last examples, we give a few comparisons between the MetaPRL modules in two additional domains. The GEN problems is a heredity problem in a large first-order database. The NUPRL problem is an automated rerun of all proof transcripts in the NuPRL type theory. The transcripts contain a mix of low-level proof steps, such as lemma application and application of inductive reasoning, to higher-level steps that include verification-condition automation and proof search. The transcripts contain about 2,500 interactive proof steps.

	Problem	
Configuration	GEN	NUPRL
`Term_std`	20.4	39.3
`Term_ds`	14.4	36.6

We don't include performance measurements for NuPRL-4 on these examples, because the system differences require a porting effort (for instance, NuPRL-4 does not currently implement a generic first-order proof search procedure). In our experience with NuPRL-4, proofs with several hundred interactive steps tend to take several minutes to replay.

Once again, the `Term_ds` module performs better than the naive terms, due to the frequent use of substitution in applications of the rules of these theories. The times for proof replay include the time spent loading the proof transcripts and building the tactic trees. This cost is similar for both term implementations, and the performance numbers are comparable. The first-order problem, GEN, performs proof search by resolution, using the refiner to construct a primitive proof tree only when a proof is found. This final step is expensive, because each resolution

step has to be justified by the refiner. The final successful proof in this problem performs about 41 thousand primitive inference steps.

2.5 Summary

We have achieved significant speedups for tactic proving. As shown in Section 2.4, our new prover design shows speedups of more than two orders of magnitude over the NuPRL-4 system. More investigation is required to determine the exact source of this speedup, however there is some evidence that most of it is due to efficient implementations of the prover components, with an additional part due to the modular design, which allows the prover to be customized with domain-specific implementations. In addition, the MetaPRL system is programmed in OCaml, an efficient modular language. In contrast, NuPRL-4 tactics are programmed in *classic* ML, which is compiled to Common Lisp, and the NuPRL-4 refiner is implemented in Common Lisp.

There are a few avenues left to explore. Since we compile rewrites to bytecode, it is natural to wonder what the effect of compiling to native code would be. Also, while we currently do not optimize the proof module, there is significant overhead in composing and saving the primitive proof trees. In some domains, we may be able to perform proof compression, or delay the composition of proofs. For the domain of first-order logics, we might achieve additional speedup using specific refiner modules: a first-order term module would contain custom representations for terms in disjunctive normal form and sequents (sequents provide particularly poor representations for large first-order problems), and the rewrite module would optimize inference by resolution.

2.6 Related work

Harrison's HOL-Light [Har96] shares some features with the MetaPRL implementation. Harrison's system is implemented in Caml-Light, and both systems require fewer computational resources than their predecessors. Howe [How98] has taken another approach to enhancing speed in NuPRL-4. The programming language defined by the NuPRL type theory is *untyped*, leading to frequent production of well-formedness (verification) conditions. Using type annotations, Howe was able to speed up rewriting in NuPRL-4 by a factor of 10. We haven't attempted to apply Howe's ideas to MetaPRL implementation of NuPRL type theory, but we believe that MetaPRL performance can be further improved using his approach.

Basin and Kaufmann [BK91] give a comparison between the NuPRL-3 system and NQTHM [BM79] (the predecessor of the ACL2 [KM97] system). The NQTHM prover uses a quantifier-free variant of Peano arithmetic. Basin and Kaufmann's measurements showed that NQTHM was roughly 15 times faster than NuPRL-3 for *different formalizations* of Ramsey's theorem.

Chapter 3

Sequent Schemata for Derived Rules

In an interactive theorem prover it is very useful to have a mechanism allowing users to prove some statement in advance and then reuse the derivation in further proofs. Often it is especially useful to be able to *abstract away* the particular derivation. For example, suppose we wish to formalize a data structure for labeled binary trees. If binary trees are not primitive to the system, we might implement them in several ways, but the details are irrelevant. The more important feature is the inference rule for induction. In a sequent logic, the induction principle would be similar to the following:

$$\frac{\Gamma \vdash P(\mathit{leaf}) \qquad \Gamma, a\colon \mathit{btree}, P(a), b\colon \mathit{btree}, P(b) \vdash P(\mathit{node}(a,b))}{\Gamma, x\colon \mathit{btree} \vdash P(x)}$$

If this rule can be established, further proofs may use it to reason about binary trees *abstractly* without having to unfold the btree definition. This leaves the user free to replace or augment the implementation of binary trees as long as she can still prove the same induction principle for the new implementation. Furthermore, in predicative logics, or in cases where well-formedness is defined logically, the inference rule is strictly more powerful than its propositional form.

If a mechanism for establishing a derived rule is not available, one alternative is to construct a proof "script" or tactic that can be reapplied whenever a derivation is needed. There are several problems with this. First, it is inefficient — instead of applying the derived rule in a single step, the system has to run through the whole proof each time. Second, the proof script would have to unfold the btree definition, exposing implementation detail. Third, proof scripts tend to be fragile, and must be reconstructed frequently as a system evolves. Finally, by looking at

a proof script or a tactic code, it may be hard to see what exactly it does, while a derived rule is essentially self-documenting.

This suggests a need for a *derived rules mechanism* that would allow users to derive new inference rules in a system and then use them as if they were *primitive rules* (i.e. axioms) of the system. Ideally, such mechanism would be general enough not to depend on a particular logical theory being used. Besides being a great abstraction mechanism, derived rules facilitate the proof development by making proofs and partial proofs easily reusable. Also, a derived rule contains some information on how it is supposed to be used [1] and as we will see in Section 4.3 such information can be made available to the system itself. This can substantially reduce the amount of information a user has to provide in order for the system to know how to use such a new rule in the proof automation procedures.

In this chapter we describe a purely syntactical way of dealing with derived rules. The key idea of our approach is in using a special higher-order language for specifying rules; we call it a *sequent schemata* language. From a theoretical point of view, we take some logical theory and express its rules using sequent schemata. Next we add the same language of sequent schemata to the theory itself. After that we allow extending our theory with a new rule $\frac{S_1 \ \cdots \ S_n}{S}$ whenever we can prove S from S_i in the expanded theory (we will call such a rule a *derived rule*). We show that no matter what theory we started with, such a double-extended theory is always a conservative extension of the original one, so our maneuver is always valid.

In case of a theorem prover (such as MetaPRL [Hic01, HNK$^+$], which implements

[1] For example, an implication $A \Rightarrow B$ can be stated and proved as an A elimination rule or as a B introduction rule, depending on how we expect it to be used.

this approach), the user has only to provide the axioms of the base theory in a sequent schemata language and the rest happens automatically. The system immediately allows the user to mix the object language of a theory with the sequent schemata meta-language. Whenever a derived rule is proven in a system, it allows using that rule in further proofs as if it were a basic axiom of the theory. [2] This chapter shows that such a theorem prover would not allow one to derive anything that is not derivable in a conservative extension of the original theory.

As we explain in Section 3.2, a key idea of sequent schemata is that whenever a binding occurrence of a variable is explicitly mentioned in a schema, all the bound occurrences of the same variable have to be explicitly mentioned as well. This gives us an ability to always specify all substitutions implicitly, which makes the language much easier to use and eliminates a very common source of user errors. Additionally, this makes it possible to automatically detect many common user errors and typos.

This chapter is arranged as follows — first in Section 3.1 we give a brief description of the base language of *plain* terms and sequents that the schemata would stand for. Next, in Section 3.2 we provide an informal overview of the term schemata and discuss some basic ideas of the language. In Section 3.3 we introduce the complete syntax of sequent schemata language in a rigorous way. Then in Section 3.4 we explain what a sequent schema stands for, that is when a *plain* sequent matches some sequent schema. In Section 3.5 we show how rules are specified using sequent schemata, and we describe how a derived rules mechanism can be

[2] A system might also allow use in the reverse order — first state a derived rule, use it, and later "come back" and prove the derived rule. Of course in such system a proof would not be considered complete until all the derived rules used in it are also proven. Such an approach allows one to "test-drive" a derived rule before investing time into establishing its admissibility.

implemented using the sequent schemata language. In Section 3.6 we argue what properties of the sequent schemata can guarantee admissibility of the rules derived using sequent schemata.

For the sake of brevity and clarity we focus on a simple version of sequent schemata (which is still sufficient for formalizing Martin-Löf style type theories, including the NuPRL one [CAB+86]). In Section 3.7 we present several ways of extending the sequent schemata language; in particular we explain how to represent constants in the sequent schemata framework. After that, in Section 3.8 we discuss related work and compare our derived rules approach to approaches taken by several other theorem provers.

This chapter discussed the theoretical aspects of the sequent schemata language. In Chapter 4 we will see how a variant of the sequent schemata language is implemented in MetaPRL proof assistant [Hic01, HNK+] using its fast rewriting engine (as described in Chapter 2).

Throughout the chapter we will ignore the issues of concrete syntax and will assume we are only dealing with abstract syntax.

Parts of this chapter are joint work with Jason Hickey — see also [NH02].

3.1 Terms and Sequents

Before we describe the language of sequent schemata, we need to give a brief description of the base language of *plain* terms and sequents that the schemata would stand for. We assume that all syntactically well-formed sentences of the theory we are extending have the form

$$x_1\colon A_1;\, x_2\colon A_2; \cdots; x_n\colon A_n \vdash C \tag{3.1}$$

where

1. Each A_i and C is a *term.* As described in Section 2.2, terms are constructed from variables using a set of operators [3] (including 0-arity operators — constants). Each operator may potentially introduce binding occurrences (for example λ would be a unary operator that introduces one binding occurrence).

2. Each hypothesis A_i introduces *a binding occurrence* for variable x_i. In other words, each x_i is bound in hypotheses A_j ($j > i$) and the conclusion C.

3. All sequents are *closed.* Each free variable of C must be one of x_i ($i = 1, \ldots, n$) and each free variable of A_j must be one of x_i ($i < j$).

Example 3.1.1. $x : \mathbb{Z} \vdash x \geq 0$ and $x : \mathbb{Z}; y : \mathbb{Z} \vdash x > y$ and $\vdash 5 > 4$ are syntactically well-formed (but not necessarily true) sequents in a theory that includes a 0-ary operator (constant) $\mathbb{Z}$ and binary operators $>$ and $\geq$.

Example 3.1.2. $x : \mathbb{Z}; y : \mathbb{Z} \vdash x = y \in \mathbb{Z}$ and $x : \mathbb{Z} \vdash \exists y : \mathbb{Z}\,.\,(x = y \in \mathbb{Z})$ are well-formed (but not necessarily true) sequents in a theory that includes a 0-ary operator $\mathbb{Z}$, a ternary operator $\cdot = \cdot \in \cdot$ and a binary operator $\exists v \in \cdot . \cdot$ that introduces a binding occurrence in its second argument.

Remark 3.1.3. We assume that the language of the theory under consideration contains at least one closed term; [4] we will use $\bullet$ when we need to refer to an arbitrary closed term.

[3] In this simple version of the language we ignore the internal structure of operators (e.g. opnames and parameters).

[4] This is not normally a limitation because the λ operator or an arbitrary constant can normally be used to construct a closed term.

As in LF [HHP93] we assume that object level variables are mapped to meta-theory variables (denoted by x, y, z). We will call these meta-theory variables *first-order*, but of course they may have a higher-order meaning in the object theory. Similarly we assume that the object-level binding structure is mapped to the meta-level binding structure and we will introduce a separate context binding structure in the sequent schemata language. We also consider alpha-equal terms (including sequents and sequent schemata) to be identical and we assume that substitution avoids capture by renaming.

3.2 Term Schemata

The goal of the sequent schemata design is to produce a simple pattern-like language that can account for the binding structure, free variables and sequents. To accomplish this, we first add second-order variables in the style of Huet and Lang [HL78] to the language.

Example 3.2.1. A term schema "$\lambda x.\, t[x]$" (where t is a unary second-order variable) matches any λ-term. A term schema "$\lambda x.\, t'$" (where t' is a second-order variable of arity 0) matches any λ-term $\lambda y. \cdots$ whose body does not have any free occurrences of y.

The above example illustrates a key idea of the language — whenever a binding occurrence of some variable is explicitly mentioned in a schema, all the bound occurrences of the same variable must be mentioned as well. Note that in $\lambda x.\, t'$, t' may match a term that has various binding and bound occurrences of some variables. However since we have explicitly mentioned the binding occurrence of x and did not include any bound occurrences of x, then there can not be any free

occurrences of x in the term that t' would match.

3.3 Language of Sequent Schemata

The sequent schemata language resembles higher-order abstract syntax presented in Pfenning and Elliott [PE88]. The idea of sequent schemata is to use higher-order context variables to describe sequent and other contexts as well as to use second-order variables to describe terms. In all rule specifications, these meta-variables will be implicitly universally quantified.

We assume there exist an infinite supply of context variables $\mathcal{C}$ and second order variables $\mathcal{V}$, that are disjoint from each other and from ordinary object-level variables. We will assume an arity function $\alpha : (\mathcal{C} \cup \mathcal{V}) \rightarrow \mathbb{N}$ and a function $\beta : (\mathcal{C} \cup \mathcal{V}) \rightarrow \{\text{finite subsets of } \mathcal{C}\}$ that determines the contexts that a variable may depend on. We will assume that for each value of α and β there are infinitely many variables in each of the two classes.

In this thesis we will use abstract syntax when discussing the language of sequent schemata. [5] We will denote context variables by $\mathbf{C}$, $\mathbf{H}$, $\mathbf{J}$ [6] and second-order variables by A, B, C and V. For clarity we will write the value of β as a subscript for all the variables (although it can usually be deduced from context and in the MetaPRL system we rarely need to specify it explicitly — see Section 4.1).

The language of sequent schemata is outlined in Table 3.1.

Remark 3.3.1. The language of sequent schemata is essentially untyped, although we could describe all the terms free of higher-order variables as having the type `term`, the second-order variables as having the type $\texttt{term}^n \rightarrow \texttt{term}$ and the context

[5] For concrete syntax used in MetaPRL system, see Appendix A.3.
[6] In other chapters of this thesis we will also use Γ and Δ to for context variables.

Table 3.1: Language of Sequent Schemata

Syntax	**Intended meaning**
S ::= $\vdash T$ | $Hs; S$. In other words S ::= $Hs_1; \cdots; Hs_n \vdash T$ $(n \geq 0)$	**Sequent schema.** *Hs*'s specify the *hypotheses* of a sequent and T specifies its *conclusion*.
Hs ::= $\mathbf{C}_{\{\mathbf{C}_1;\cdots;\mathbf{C}_k\}}[T_1;\cdots;T_n]$ | $x : T$ where $\mathbf{C}, \mathbf{C}_i \in \mathcal{C}$, $\alpha(\mathbf{C}) = n$, T and T_i are terms and x is an object-level variable.	**Hypotheses specification.** The first variant is used to specify a sequent context — a sequence (possibly empty) of hypotheses. In general, a context may depend on some arguments and T_i specify the values of those arguments. $\mathbf{C}_i$ are the contexts that introduce the variables that are allowed to occur free in $\mathbf{C}$ itself (not in its arguments). The second variant specifies a single hypothesis that introduces a variable x. As in (3.1) this is a binding occurrence and x becomes bound in all the hypotheses specifications following this one, as well as in the conclusion.
SOV ::= $V_{\{\mathbf{C}_1;\cdots;\mathbf{C}_k\}}[T_1;\cdots;T_n]$ where $V \in \mathcal{V}$, $\alpha(V) = n$, $\mathbf{C}_i \in \mathcal{C}$, T and T_i are terms.	**Second-order variable occurrences.** Second-order variables in sequent schemata language are the ordinary second-order variables as in [HL78] except that we need $\beta(V)$ to be able to specify the names of contexts which introduced the variables that can occur free in V.
T — a term built using operators from variables and second-order variables.	**Term specification.** Term specifications are ordinary terms except that they may contain second-order variables.

variables as having the type $(\texttt{term}^m \rightarrow \texttt{term}) \rightarrow \texttt{term}^n \rightarrow \texttt{term}$.

Remark 3.3.2. It is worth mentioning that a *plain* sequent (as described in (3.1)) is just a particular case of a sequent schema. Namely, a plain sequent is a sequent schema that contains neither sequent contexts, nor second-order variables.

Example 3.3.3.

$$\mathbf{H}_{\{\}}[];\, x : A_{\{\mathbf{H}\}}[];\, \mathbf{J}[x]_{\{\mathbf{H}\}} \;\vdash\; C_{\{\mathbf{H};\mathbf{J}\}}[x^2]$$

is a sequent schema in a language that contains the $\cdot^2$ unary operator. In this schema $\mathbf{H}$ and $\mathbf{J}$ are context variable, x is an ordinary (first-order) variable, A and C are second-order variables. This schema matches many plain sequents, including the following ones:

$$x : \mathbb{Z} \vdash (x^2) \in \mathbb{Z}$$

$$y : \mathbb{Z};\, x : \mathbb{Z};\, z : \mathbb{Z} \vdash (z + y) \in \mathbb{Z}$$

Here is how the same schema would look in a simplified syntax of Section 4.1:

$$\mathbf{H};\, x : A;\, \mathbf{J}[x] \;\vdash\; C[x^2]$$

3.4 Semantics — Sequent Schemata

Informally speaking, the main idea of sequent schemata is that whenever a binding occurrence of some variable is explicitly mentioned (either *per se* or as a context), it can only occur freely in places where it is explicitly mentioned, but can not occur freely where it is omitted. Second-order variables are meant to stand for contexts that can not introduce new bindings and context variables are meant to stand for contexts that may introduce an arbitrary number of new bindings (with β being used to restrict where those new bound variables may potentially occur freely).

To make the above more formal, we first need to define what we mean by free and bound variables in sequent schemata.

Definition 3.4.1 (free variables). *The notions of free variables (or, more specifically, free first-order variables)* $\mathbf{FOV}$ *and free context variables* $\mathbf{CV}$ *are defined as follows:*

- *Free variables of plain terms (free of higher-order variables) are defined as usual. Plain terms do not have free context variables.*
- *For* $\Upsilon \in (\mathcal{C} \cup \mathcal{V})$, $\mathbf{FOV}(\Upsilon_{\{\cdots\}}[T_1; \cdots ; T_n]) \;=\; \bigcup_{1\le i\le n} \mathbf{FOV}(T_i)$.
- $\mathbf{CV}(\Upsilon_{\{\mathbf{C}_1;\cdots;\mathbf{C}_k\}}[T_1; \cdots ; T_n]) \;= \{\mathbf{C}_1; \cdots ; \mathbf{C}_n\} \cup \bigcup_{1\le i\le n} \mathbf{CV}(T_i)$.
- $\mathbf{FOV}(\vdash T) \;=\; \mathbf{FOV}(T)$ *and* $\mathbf{CV}(\vdash T) \;=\; \mathbf{CV}(T)$
- *In a sequent schema* $x : T;\, S$, *the hypothesis specification* $x : T$ *binds all free occurrences of* x *in* S. *Hence,*

$$\begin{aligned}\mathbf{FOV}(x : T;\, S) &= (\mathbf{FOV}(S) - \{x\}) \cup \mathbf{FOV}(T)\\ \mathbf{CV}(x : T;\, S) &= \mathbf{CV}(S) \cup \mathbf{CV}(T).\end{aligned}$$

- *In a sequent schema* $\mathbf{C}_{\{\mathbf{C}_1;\cdots;\mathbf{C}_k\}}[T_1; \cdots ; T_n];\, S$ *the hypothesis specification* $\mathbf{C}_{\{\cdots\}}[\cdots]$ *binds the free occurrences of* $\mathbf{C}$ *in* S. *Hence,*

$$\begin{aligned}\mathbf{CV}(\mathbf{C}_{\{\mathbf{C}_1;\cdots;\mathbf{C}_k\}}[T_1; \cdots ; T_n];\, S) \;&= \{\mathbf{C}_1; \cdots ; \mathbf{C}_k\} \cup (\mathbf{CV}(S) - \{\mathbf{C}\}) \cup\\ &\qquad \bigcup_{1\le i\le n} \mathbf{CV}(T)\\ \mathbf{FOV}(\mathbf{C}_{\{\mathbf{C}_1;\cdots;\mathbf{C}_k\}}[T_1; \cdots ; T_n];\, S) \;&= \mathbf{FOV}(C) \cup \mathbf{FOV}(S) \cup \bigcup_{1\le i\le n} \mathbf{CV}(T).\end{aligned}$$

- *For all objects* $\star$ *such that* $\mathbf{FOV}(\star)$ *and* $\mathbf{CV}(\star)$ *are defined (see also Definitions 3.4.3 and 3.4.4 below) we will denote* $\mathbf{FOV}(\star) \cup \mathbf{CV}(\star)$ *as* $\mathbf{Vars}(\star)$.

Definition 3.4.2. *We will call a sequent schema S* closed *when* $\mathbf{Vars}(S) = \emptyset$.

Definition 3.4.3. *A substitution function is a pair of a term and a list of first-order variables. For a substitution function $\sigma = \langle T; x_1, \cdots, x_n\rangle$ ($n \geq 0$) we will say that the arity of σ is n; the free occurrences of x_i in T are bound in σ, so $\mathbf{FOV}(\sigma) = \mathbf{FOV}(T) - \{x_1; \cdots; x_n\}$ and $\mathbf{CV}(\sigma) = \mathbf{CV}(T)$. By $\sigma(T_1, \cdots, T_n)$ we will denote $[T_1/x_1, \cdots, T_n/x_n]T$ — the result of simultaneous substitution of T_i for x_i ($1 \leq i \leq n$) in T. As usual, we will consider two alpha-equal substitutions to be identical.*

We will say that the substitution function is trivial, when T is a closed term.

Definition 3.4.4. *Similarly, a context substitution function is a pair of a list of hypothesis specification and a list of first-order variables. For a context substitution function*

$$\Sigma = \langle Hs_1, \cdots, Hs_k; x_1, \cdots, x_n\rangle \quad (k, n \geq 0)$$

we will say that the arity of Σ is n, $\mathbf{FOV}(\Sigma) = \mathbf{FOV}(Hs_1; \cdots; Hs_k \vdash \bullet) - \{x_1; \cdots; x_n\}$ and $\mathbf{CV}(\Sigma) = \mathbf{CV}(Hs_1; \cdots; Hs_k \vdash \bullet)$, where $\bullet$ is an arbitrary closed term (see Remark 3.1.3). We will say that Σ introduces a set of bindings

$$\mathbf{BV}(\Sigma) = \bigcup_{1 \leq i \leq k} \begin{cases} \{x\}, & \textit{when } Hs_i \textit{ is } x : T \\ \{\mathbf{C}\}, & \textit{when } Hs_i \textit{ is } \mathbf{C}_{\{\cdots\}}[\cdots] \end{cases}$$

We will say that a context substitution function is trivial when $k = 0$.

Definition 3.4.5. *A function R on $\mathcal{C} \cup \mathcal{V}$ is a schema* refinement function *if it "respects" α and β. Namely, the following conditions must hold:*

1. *For all $V \in \mathcal{V}$, $R(V)$ is a substitution function of arity $\alpha(V)$.*

2. *For all $\mathbf{C} \in \mathcal{C}$, $R(\mathbf{C})$ is context substitution function of arity $\alpha(\mathbf{C})$.*

3. *For all* $\Upsilon \in (\mathcal{C} \cup \mathcal{V})$, $\mathbf{Vars}(R(\Upsilon)) \subseteq \mathbf{BV}(R(\beta(\Upsilon)))$ *where* $\mathbf{BV}(R(\beta(\Upsilon)))$ *is a notation for* $\bigcup_{\mathbf{C} \in \beta(\Upsilon)} \mathbf{BV}(R(\mathbf{C}))$.

Definition 3.4.6. *If R is a refinement function, we will extend R to terms, hypothesis specifications, sequent schemata and substitution functions as follows:*

1. $R(T) \ = \ T$, *when T is a plain term.*

2. $R(V_{\{\cdots\}}[T_1; \cdots ; T_n]) \ = \ R(V)(R(T_1), \cdots , R(T_n))$. *From Definition 3.4.5 we know that $R(V)$ is a substitution function of an appropriate arity and from Definition 3.4.3 we know how to apply it to a list of terms, so this definition is valid.*

3. $R(x : T) \ = \ x : R(T)$ *and* $R(\mathbf{C}_{\{\cdots\}}[T_1; \cdots ; T_n]) \ = \ R(\mathbf{C})(R(T_1), \cdots , R(T_n))$.

4. $R(\vdash T) \ = \vdash \ R(T)$ *and* $R(Hs;\, S) \ = \ R(Hs);\, R(S)$.

5. $R(\langle T; x_1, \cdots , x_n \rangle) \ = \ \langle R(T); x_1, \cdots , x_n \rangle$ *and*
   $R(\langle Hs_1, \cdots , Hs_k; x_1, \cdots , x_n \rangle) \ = \ \langle R(Hs_1), \cdots , R(Hs_k); x_1, \cdots , x_n \rangle$
   (as usual, we assume that x_i are automatically alpha-renamed to avoid capture).

Lemma 3.4.7. *If S is a closed sequent schema and R is a refinement function, then $R(S)$ is also a closed sequent schema.*

Proof. This property follows immediately from condition (3) in Definition 3.4.5 of refinement functions.

Definition 3.4.8. *We say that a sequent schema (possibly a plain sequent) S* matches *a sequent schema S'* iff *S is $R(S')$ for some refinement function R.*

Example 3.4.9.

1. Schema $\vdash \lambda x.A_{\{\}}[x]$ is matched by every well-formed $\vdash \lambda x.(\cdots)$ no matter what $\cdots$ is.

2. Schema $\vdash \lambda x.A_{\{\}}$ is matched by every well-formed $\vdash \lambda x.(\cdots)$ as long as $\cdots$ has no free occurrences of x. But in case $\cdots$ does have free occurrences of x, we would not be able to come up with a refinement function without violating Lemma 3.4.7.

3. Schema $\mathbf{H}; x : A_{\{\mathbf{H}\}} \vdash C_{\{\mathbf{H}\}}$ essentially specifies that any matching sequent must have at least one hypothesis and that the variable introduced by the last hypothesis can not occur freely in the conclusion of the sequent. In particular, it is matched by $x : \mathbb{Z} \vdash 5 \in \mathbb{Z}$ using refinement function

$$\mathbf{H} \rightsquigarrow \langle ; \rangle \qquad A \rightsquigarrow \mathbb{Z} \qquad C \rightsquigarrow 5 \in \mathbb{Z}$$

It is also matched by $\mathbf{H}; x : \mathbb{Z}; J_{\{\mathbf{H}\}}[x]; y : \mathbb{Z} \vdash x \in \mathbb{Z}$ using refinement function

$$\mathbf{H} \rightsquigarrow \left\langle \mathbf{H}\,,\, x : \mathbb{Z}\,,\, J_{\{\mathbf{H}\}}[x]\ ;\ \right\rangle \qquad A \rightsquigarrow \mathbb{Z} \qquad C \rightsquigarrow x \in \mathbb{Z}$$

However it is not matched by neither $x : \mathbb{Z} \vdash x \in \mathbb{Z}$ nor $\vdash 5 \in \mathbb{Z}$.

4. Every sequent schema (including every plain sequent) matches the schema $\mathbf{H} \vdash A_{\{\mathbf{H}\}}$.

5. See Example 3.3.3.

3.5 Rule Specifications

Definition 3.5.1. *If S_i $(0 \leq i \leq n,\ n \geq 0)$ are closed sequent schemata, then* $\dfrac{S_1 \ \cdots \ S_n}{S_0}$ *is* a rule schema *(or just* a rule*).*

Definition 3.5.2. *We will say that a rule* $\dfrac{S_1 \ \cdots \ S_n}{S_0}$ *is* an instance *of* $\dfrac{S'_1 \ \cdots \ S'_n}{S'_0}$ *when for some refinement function R we have $S_i = R(S'_i)$ $(0 \leq i \leq n)$. We will call such an instance* plain *if all S_i are plain sequents.*

Example 3.5.3. The following is the rule specification for the weakening (thinning) rule:

$$\frac{\mathbf{H}_{\{\}}[];\ \mathbf{J}_{\{\mathbf{H}\}}[] \vdash B_{\{\mathbf{H};\mathbf{J}\}}[]}{\mathbf{H}_{\{\}}[];\ x : A_{\{\mathbf{H}\}}[];\ \mathbf{J}_{\{\mathbf{H}\}}[] \vdash B_{\{\mathbf{H};\mathbf{J}\}}[]}$$

Note that condition (3) of Definition 3.4.5 guarantees that x will not occur free in whatever would correspond to $\mathbf{J}$ and B in instances of this rule.

Here is how this rule would look if we take all the syntax shortcuts described in the Section 4.1:

$$\frac{\mathbf{H};\ \mathbf{J} \vdash B}{\mathbf{H};\ x : A;\ \mathbf{J} \vdash B}$$

This is exactly how this rule is written in the MetaPRL system.

Example 3.5.4. A `Cut` rule might be specified as

$$\frac{\mathbf{H}_{\{\}}[];\ \mathbf{J}_{\{\mathbf{H}\}}[] \vdash C_{\{\mathbf{H}\}}[] \qquad \mathbf{H}_{\{\}}[];\ x : C_{\{\mathbf{H}\}}[];\ \mathbf{J}_{\{\mathbf{H}\}}[] \vdash B_{\{\mathbf{H};\mathbf{J}\}}[]}{\mathbf{H}_{\{\}}[];\ \mathbf{J}_{\{\mathbf{H}\}}[] \vdash B_{\{\mathbf{H};\mathbf{J}\}}[]}$$

Note that in the first assumption of this rule C is inside the context $\mathbf{J}$, however $\beta(C)$ can not contain $\mathbf{J}$ since otherwise the second assumption would not be closed. In fact, this particular example shows why we need to keep track of β and can not just always deduce it from context — see also Remark 3.6.6.

With Section 4.1 syntax shortcuts `Cut` would look as

$$\frac{\mathbf{H};\ \mathbf{J} \vdash C_{\{\mathbf{H}\}} \qquad \mathbf{H};\ x : C;\ \mathbf{J} \vdash B}{\mathbf{H};\ \mathbf{J} \vdash B}$$

Assume we have an arbitrary set V of plain sequents. We do not assume anything at all about this V — it can be a set of all sequents derivable in a certain proof system, a set of all sequents valid in certain semantics, or anything else.

Definition 3.5.5. *We say that a rule* $\dfrac{S_1 \ \cdots \ S_n}{S_0}$ $(n \geq 0)$ *is* admissible *for* V *when for every plain instance* $\dfrac{S'_1 \ \cdots \ S'_n}{S'_0}$*, whenever* $S'_i \in V$ *for* $1 \leq i \leq n$*, then* S'_0 *is also in* V*.*

Remark 3.5.6. Informally speaking, Definitions 3.5.2 and 3.5.5 say that in rule schemata all second-order and context variables are implicitly universally quantified.

Definition 3.5.7. *We will say that a rule* $\dfrac{S_1 \ \cdots \ S_m}{S}$ $(m \geq 0)$ *is* derived *from a set of rules* $\mathcal{R}$ *if there exists a derivation sequence* $S_1, \ \cdots, \ S_m, \ \cdots, \ S_n = S$ *where every schema after* S_m *is derived from the previous ones using a rule from* $\mathcal{R}$*. In other words, for every* $k > m$ *there must exist some* $1 < i_1, \cdots, i_l < k$ *such that* $\dfrac{S_{i_1} \ \cdots \ S_{i_l}}{S_k}$ *is an instance of a rule from* $\mathcal{R}$*.*

Now we can give a complete description of the *mental* procedure (in case of an automated system, the system will do most of the work) for adding derived rules support to a logical theory. Suppose V is the set of all the plain sequents that are valid in the theory. First, we formalize the theory using rule schemata that are admissible for V. Second, we expand the language of the theory from plain sequents to plain schemata and we allow the use of arbitrary instances of the rules in the proofs, not just the plain instances. And whenever we can prove S_0 from S_i $(1 \leq i \leq n)$, we allow adding the rule $\dfrac{S_1 \ \cdots \ S_n}{S_0}$ to the list of rules and allow using such *derived rules* in addition to the rules present in the initial formalization of the theory.

In the next section we prove that this approach leads to a conservative extension of the original theory.

From the point of view of a user of an automated system that implements

this approach, in order to start formal reasoning in some logical theory the user only needs to describe the syntax of the theory (for example, give a list of the operators and their corresponding arities) and then to provide the system with the list of the primitive rules (e.g. axioms) of the theory in the sequent schemata syntax. [7] After that the system would immediately allow the user to start using the combined language of sequent schemata and of base theory in derivations and whenever user would prove schemata S from assumptions S_i $(1 \leq i \leq n)$, the system would allow using the rule $\frac{S_1 \; \cdots \; S_n}{S_0}$ in further derivations on an equal footing with the primitive rules of the base theory [8]. The main theorem of the next sections guarantees that the derived rules would not allow proving anything that is not derivable directly from the primitive rules.

3.6 Conservativity

Theorem 3.6.1. *The matching relation is transitive. That is, if R_1 and R_2 are refinement functions, then $R = R_2 \circ R_1$ is also a refinement function.*

Proof. This property follows from the definitions of Section 3.4. Since application of a refinement function does not change the arity of a substitution function (see Definitions 3.4.3, 3.4.4 and 3.4.6 (5)) and since R_1 is a refinement function, the arity conditions of Definition 3.4.5 will hold for R. Hence we only need to prove that the free variables condition (3) of Definition 3.4.5 holds for R.

Consider $\Upsilon \in (\mathcal{C} \cup \mathcal{V})$. The proofs of condition (3) for the cases $\Upsilon \in \mathcal{C}$ and $\Upsilon \in \mathcal{V}$

[7] This does not imply that the set of rules have to be finite. MetaPRL, for example, provides the facility to specify an infinite family of rules by providing the code capable of recognizing valid instances.

[8] As we mentioned earlier, some systems might chose to allow user to “test-drive” a derived rule before forcing the user to prove the rule.

are very similar, so we will only present the latter. We know that $R_1(\Upsilon)$ is some substitution function $\langle T; x_1, \cdots, x_n\rangle$. We also know that $R(\Upsilon) = R_2(R_1(\Upsilon)) = \langle R_2(T); x_1, \cdots, x_n\rangle$.

From Definitions 3.4.3 and 3.4.4 we know that $\mathbf{Vars}(R(\Upsilon)) = \mathbf{Vars}(R_2(T)) - \{x_1; \cdots; x_n\}$. From Definitions 3.4.1 and 3.4.6(1,2) we can deduce that

$$\mathbf{Vars}(R_2(T)) = \mathbf{FOV}(T) \cup \left(\bigcup_{\{V \in \mathcal{V} | V \text{ occurs in } T\}} \mathbf{Vars}(R_2(V)) \right). \tag{3.2}$$

Let us consider $\mathbf{FOV}(T)$ first. Since R_1 is a refinement function, from the condition (3.4.5) (3) for it we know that $\mathbf{FOV}(T)$ (except for x_i, but we do not need to consider x_i since they are not in $\mathbf{Vars}(R(\Upsilon))$) are all in $\mathbf{BV}(R_1(\beta(\Upsilon)))$ and from the definition of $\mathbf{BV}$ and the Definition 3.4.6 (3) we know that an application of a refinement function preserves all the first-order variables of $\mathbf{BV}$, so $\mathbf{FOV}(T) \subseteq \mathbf{BV}(R(\beta(\Upsilon)))$.

Now to cover the rest of (3.2) consider some $V \in \mathcal{V}$ such that V occurs in T. From Definition 3.4.1 we know that $\beta(V) \subseteq \mathbf{CV}(T) \subseteq \mathbf{Vars}(R_1(\Upsilon))$. And since R_1 is a refinement function, $\mathbf{Vars}(R_1(\Upsilon)) \subseteq \mathbf{BV}(R_1(\beta(\Upsilon)))$, so

$$\beta(V) \subseteq \mathbf{BV}(R_1(\beta(\Upsilon))) \tag{3.3}$$

Now consider some $\mathbf{C} \in \beta(V)$. We know from (3.3) that $\mathbf{C} \in \mathbf{BV}(R_1(\beta(\Upsilon)))$ and from the Definitions 3.4.4 and 3.4.6 (3) that if $\mathbf{C} \in \mathbf{BV}(R_1(\beta(\Upsilon)))$, then $\mathbf{BV}(R_2(\mathbf{C})) \subseteq \mathbf{BV}(R_2(R_1(\beta(\Upsilon))))$. This means that $\forall \mathbf{C} \in \beta(V). \mathbf{BV}(R_2(\mathbf{C})) \subseteq \mathbf{BV}(R_2(R_1(\beta(\Upsilon))))$ and if we take the union over all $\mathbf{C} \in \beta(\Upsilon)$ and recall that $R = R_2 \circ R_1$, we will get $\mathbf{BV}(R_2(\beta(\Upsilon))) \subseteq \mathbf{BV}(R(\beta(\Upsilon)))$. Since R_2 is a refinement function, condition (3) dictates that $\mathbf{Vars}(R_2(V)) \subseteq \mathbf{BV}(R_2(\beta(V)))$, so $\mathbf{Vars}(R_2(V)) \subseteq \mathbf{BV}(R(\beta(\Upsilon)))$ which takes care of the remaining part of (3.2). $\square$

Note that there are two alternative ways of defining the value of $R_2 \circ R_1$ on terms, hypothesis specifications, sequent schemata and substitution functions. The first is to define $(R_2 \circ R_1)(\Upsilon) = R_2(R_1(\Upsilon))$ and the second is to define $(R_2 \circ R_1)$ based on its value on $\mathcal{C} \cup \mathcal{V}$ using Definition 3.4.6.

Lemma 3.6.2. *The two definitions above are equivalent.*

Proof. This follows trivially from Definition 3.4.6 and the fact that

$$\sigma([T_1/x_1; \cdots ; T_n/x_n]T) = [\sigma(T_1)/x_1; \cdots ; \sigma(T_n)/x_n](\sigma(T))$$

(again, we rely on the assumption that x_i's are automatically alpha-renamed as needed to avoid capture).

Theorem 3.6.3. *For any V, every instance of an admissible rule is also an admissible rule.*

Proof. If $\mathcal{R}$ is an admissible rule schema and $\mathcal{R}'$ is its instance, then from Theorem 3.6.1 we know that every instance of $\mathcal{R}'$ will also be an instance of $\mathcal{R}$. Because of that and by Definition 3.5.5, if $\mathcal{R}$ is admissible, $\mathcal{R}'$ must be admissible too. □

Lemma 3.6.4. *Let us define $R_{\mathtt{triv}}$ to be a function that always returns a trivial substitution (see Definitions 3.4.3 and 3.4.4), namely*

$$R_{\mathtt{triv}}(\Upsilon) = \begin{cases} \left\langle \bullet; x_1, \cdots, x_{\alpha(\Upsilon)} \right\rangle, & \textit{when } \Upsilon \in \mathcal{V} \\ \left\langle ; x_1, \cdots, x_{\alpha(\Upsilon)} \right\rangle, & \textit{when } \Upsilon \in \mathcal{C} \end{cases}$$

Such $R_{\mathtt{triv}}$ would be a refinement function.

Proof. The conditions of Definition 3.4.5 are obviously true for $R_{\mathtt{triv}}$ — by construction it returns substitutions of the right arity and condition (3) is satisfied since $\mathbf{Vars}(R_{\mathtt{triv}}(\Upsilon))$ will always be an empty set.

Theorem 3.6.5. *For any V, if* $\mathcal{R} = \dfrac{S_1 \ \cdots \ S_m}{S}$ *and* $\mathcal{R}' = \dfrac{S \ S'_1 \ \cdots \ S'_n}{S'}$ *($m, n \geq 0$) are admissible rules, then the rule* $\dfrac{S_1 \ \cdots \ S_m \ S'_1 \ \cdots \ S'_n}{S'}$ *is also admissible.*

Proof. Suppose that R is a refinement function,

$$\frac{R(S_1) \ \cdots \ R(S_m) \ \ R(S'_1) \ \cdots \ R(S'_n)}{R(S')}$$

is a plain instance of the rule in question and all $R(S_i)$ $(1 \leq i \leq m)$ and $R(S'_j)$ $(1 \leq j \leq n)$ are derivable. We need to establish that $R(S')$ is also derivable.

Let R' be $R_{\mathtt{triv}} \circ R$. We know that whenever R returns a plain sequent, R' will return the same plain sequent (since an application of a refinement function does not change plain terms and plain sequents). Also, $R_{\mathtt{triv}}$ and hence R' will always turn every schema into a plain sequent.

We know that $R(S_i) = R'(S_i)$ are derivable plain sequents and we know that $\mathcal{R}$ is an admissible rule, so by applying R' to $\mathcal{R}$ we get that $R'(S)$ is also derivable. Similarly, we know that $R(S'_j) = R'(S'_j)$ are derivable and by applying R' to admissible rule $\mathcal{R}'$ we get that $R'(S) = R(S)$ is derivable. □

Remark 3.6.6. In Example 3.5.4 we saw that β can not always be deduced from context. In fact it was the `Cut` example that illuminated the need for an explicit β — without it we could construct a counterexample for Theorem 3.6.5 by taking $\mathcal{R}'$ to be the `Cut`.

Theorem 3.6.7 (Conservativity). *For an arbitrary set of rule schemata, any plain sequent that can be derived from it using the language of sequent schema and the derived rules approach (as described at the end of Section 3.5), can also be derived directly from the original set of rules using only plain sequents in the derivation.*

Proof. Suppose we have a set $\mathcal{T}$ of rule schemata. Let $V(\mathcal{T})$ be all the plain sequents that can be derived directly from $\mathcal{T}$ using only plain sequents in the derivation. From the Definition 3.5.5 of admissibility, every rule in $\mathcal{T}$ is admissible for $V(\mathcal{T})$.

By induction on k in Definition 3.5.7 we can show using Theorems 3.6.3 and 3.6.5 that adding new derived rules preserves admissibility. Hence any rule that we could derive using rules from $\mathcal{T}$ (and rules derived using rules in $\mathcal{T}$, and rules derived using those rule, *etc.*) will still be admissible for $V(\mathcal{T})$.

Now suppose we have derived a plain sequent S from $\mathcal{T}$ using the derived rules mechanism. Since all the rules we could use in the process were admissible for $V(\mathcal{T})$ then by the definition of admissibility, $S \in V(\mathcal{T})$. Hence by construction of $V(\mathcal{T})$, S can be derived directly from rules in $\mathcal{T}$ using only plain sequents in the derivation. □

Corollary 3.6.8. *Our procedure of extending a logical theory with derived rules (as described at the end of Section 3.5) will produce a conservative extension of the original theory.*

3.7 Extending the Language of Sequent Schemata

For simplicity in this chapter we presented a minimal calculus necessary for adding a derived rule mechanism to a type theory. However, by appropriately extending the language of sequent schemata we can apply our technique to other logical theories (including more complicated versions of type theories). Of course, for each of these extensions we need to make sure that the matching relation is still transitive and that refinements can not turn a closed schema into a non-closed one. However these proofs are usually very similar to the ones presented in this chapter.

One obvious extension is an addition of meta-variables that would range over some specific kind of constants. In particular, in MetaPRL formalization of NuPRL type theory [Hic01] we use meta-variables that range over integer constants, meta-variables that range over string constants, *etc.* As usual, we consider all these meta-variables to be universally quantified in every rule schema.

Before we can present other extensions of the language of sequent schemata, we first need to better understand the way we are using sequent contexts to describe sequents. If we introduce special "sequent" operators **hyp** and **concl**, a sequent described in (3.1) can be rewritten as

$$\mathbf{hyp}\Big(A_1;\, x_1.\, \mathbf{hyp}\Big(A_2;\, x_2.\, \cdots \mathbf{hyp}\Big(A_n;\, x_n.\, \mathbf{concl}\big(C\big)\Big) \cdots \Big)\Big) \tag{3.4}$$

Notice that **hyp** is a binary operator that introduces a binding occurrence in its second argument, so (3.4) has the same binding structure as (3.1). Now if we rewrite $\mathbf{C}_{\{\dots\}}[T_1; \cdots; T_n];\, S$ in the same style, we get $\mathbf{C}_{\{\dots\}}[S'; T_1; \cdots; T_n]$ where S' is a rewrite of S. In this notation, we can say that $\mathbf{C}$ acts like an ordinary second-order variable on its 2-nd through n-th arguments, but on its first argument it acts in a following way:

1. $\mathbf{C}$ is bound in its first argument, but not in others. In other words, while a second-order variable can not bind free variables of its arguments, a context instance can bind free variables of its first argument.

2. The substitution function $R(C)$ has to use its first argument exactly once. In other words, each instance of $\mathbf{C}_{\{\dots\}}[S'; T_1; \cdots; T_n]$ would have exactly one occurrence of S' (while it may have arbitrary number of occurrences of each T_i including 0 occurrences of it).

3. **C** stands for a chain of **hyp** operators with the S' at the end of the chain. In some instances of $\mathbf{C}_{\{\cdots\}}[S'; T_1; \cdots; T_n]$ parts of that chain may be represented by other context variables, but in a plain instance it will always be a chain.

It can be easily shown that the language of sequent schemata would still have all necessary properties if we add some (or all) of the following:

- "General" contexts that do not have restriction (3) or even restriction (2).
- Contexts for operators that have the same binding structure as **hyp** (note that we do not have to add a new kind of contexts for each operator — we can just make the operator name be a parameter). In particular, in type theory it might be useful to have contexts tied to the dependent product operator and to the dependent intersection operator [Kop00].

3.8 Related Work

Most modern proof assistants allow their users to prove some statement in advance and then reuse it in further proofs. However, most of those mechanisms have substantial limitations.

NuPRL [CAB+86] allows its users to prove and reuse theorems (which must be closed sentences). Unfortunately, many elimination and induction principles can not be stated as theorems.

In addition to a NuPRL-like theorems mechanism, HOL [GM93] also has a derived rules mechanism, but in reality HOL's "derived rules" are essentially tactics that apply an appropriate sequence of primitive axioms. While this approach guarantees safety, it is extremely inefficient. According to [NSM01], in a recent version of HOL many of the basic derived rules were replaced with the primitive

versions (presumably to boost the efficiency of the system), and currently there is nothing in the system that attempts to check whether these rules are still truly derivable. In fact, the commented out tactic code that would "properly" implement those derived rules is no longer compatible with the current version of the HOL system.

Logical frameworks (such as Isabelle [Pau94] and LF [HHP93] based systems, including Elf [Pfe89]) use a rather complex meta-theory that can be used to allow its users to prove and reuse derived rules. Still, we believe that by directly supporting sequents-based reasoning and directly supporting derived rules, the sequent schemata language can provide a greater ease of use (by being able to make more things implicit) and can be implemented more efficiently (by optimizing the logical engine for the refinement functions as defined in 3.4.5 and 3.4.6 as we have described in Chapter 2). Additionally, sequent schemata allow us to prove the conservativity result once and for all the logical theories, while in a more general approach we might have to make these kinds of arguments every time as a part of proving an *adequacy* theorem for each logical theory being formalized.

Chapter 4

Logical Meta-Language: From Derived Rules to Tactics

There are three main reasons for using a well-defined logical meta-language in a logical framework. First, in a logical framework we want users to be able to specify new and modify existing rules of a theory, so we need some language for specifying rules. Second, as we saw in Chapter 3, an appropriate logical meta-language can be used to implement a derived rules mechanism. Finally, as we will see in Section 4.3 when the rules are specified in a well-defined logical meta-language, we can add some reflective properties to the system by giving some parts of the system access to the *text* of the rules present in it.

In the MetaPRL system [Hic01, HNK$^+$] we use a variant of the sequent schemata language described in Chapter 3 as such a logical meta-language. In Section 4.1 we will present some language simplifications used in MetaPRL. Next, in Section 4.2 we will describe the way MetaPRL sequent schemata are used to specify the rules and how those rule specifications get processed by the rewriting engine presented in Section 2.3. In Section 4.3 we will show how the development and maintenance of a certain class of *generic* tactics can be greatly facilitated by giving these tactics access to the text of the rules. Next, in Section 4.4 we will present another approach to building generic tactics thus establishing a whole layer of generic tactics, as described in Section 4.5.

4.1 Syntax Simplifications

In Chapter 3 we have presented a very verbose syntax of sequent schemata language. That verbosity was useful when presenting the theory of the language. However for implementation in the MetaPRL system we tried to come up with a simpler syntax that would allow omitting some redundant information while still preserving those redundancies that are helpful in detecting bugs and typos.

The biggest (and probably the most obvious) redundancy in the sequent schemata language is the need for specifying the value of β for all the second-order and context variables. Since a sequent schema must be closed, for every instance of a context or a second-order variable $\Upsilon_{\beta(\Upsilon)}[\cdots]$ we know that all the variables of $\beta(\Upsilon)$ must be bound in that schema. This gives us an upper limit on $\beta(\Upsilon)$. In cases when the actual $\beta(\Upsilon)$ is equal to that upper limit, we may omit the $\beta(\Upsilon)$ subscript and deduce the value of β from the rest of the schema. Our experience shows that we can take this shortcut in almost all of the cases, but as we have seen in Example 3.5.4, sometimes the value of β still has to be explicitly specified.

Additionally, when a context or a second-order variable has an arity of zero, then according to Section 3.3 we have to write $\Upsilon_{\beta(\Upsilon)}[]$, but in this case it is natural to omit the argument part $[]$.

Remark 4.1.1. When both of the omissions mentioned above are performed for a second-order variable V of arity 0, it becomes possible to confuse it with a first-order variable V. However, we can still distinguish them in a context of a rule schema — if a particular occurrence of V is bound, it must be a first-order variable (since second-order variables can not be bound) and if it is free, it must be a second-order variable (since by Definition 3.5.1 sequent schemata in a rule

can not have free first-order variables).

Example 4.1.2. The thinning rule of Example 3.5.3

$$\frac{\mathbf{H}_{\{\}}[];\ \mathbf{J}_{\{\mathbf{H}\}}[]\ \vdash\ B_{\{\mathbf{H};\mathbf{J}\}}[]}{\mathbf{H}_{\{\}}[];\ x : A_{\{\mathbf{H}\}}[];\ \mathbf{J}_{\{\mathbf{H}\}}[]\ \vdash\ B_{\{\mathbf{H};\mathbf{J}\}}[]}$$

would become

$$\frac{\mathbf{H};\ \mathbf{J}\ \vdash\ B}{\mathbf{H};\ x : A;\ \mathbf{J}\ \vdash\ B}$$

when all syntax simplifications are applied. This is exactly how this rule is written in the MetaPRL system (using the concrete syntax presented in Appendix A.3).

Remark 4.1.3. While these simplifications reduce redundancies of the language we use, our experience suggests that the simplified language still contains enough redundancy to be able to successfully detect most common typographical errors and other common mistakes. In particular, by using the second-order variables to specify the substitutions indirectly and by requiring sequent schemata to be closed, we make it much harder to forget to specify that a certain substitution needs to be performed or a certain variable needs to be renamed — a mistake that might be easy to make in a system where rule specification language contains an explicit substitution operator.

4.2 MetaPRL Rule Specifications

In the MetaPRL system rule specifications have a dual purpose. First, they provide a rule schema (as described in Section 3.5) which specifies valid instances of that rule (see Definitions 3.5.1 and 3.5.2). Second, they specify a way to point at a particular instance of that rule when applying the rule to a particular proof goal.

Remark 4.2.1. We should emphasize that MetaPRL refiner expects the rule instance to always be uniquely specified. Of course, this does not prohibit writing advanced tactics that would use, for example, a higher-order unification (or type inference, or heuristics, or something else) to find out the right rule instance and than instruct the system to apply the appropriate instance. However such a tactic would be outside of the kernel of the system. This approach provides users with the freedom of choosing whatever algorithms they want to come up with the best rule instance. It also allows reducing the trusted core of the system by only requiring that it could check whether a particular instance is applicable, and delegating the job of finding an appropriate instance to user space.

In MetaPRL a rule specification contains a rule schema $\dfrac{S_1 \ \cdots \ S_n}{S}$ together with a list of argument specifications $As_1, \cdots, As_m$ $(m \geq 0)$ where each As_i is either a closed term schema or a context variable name. [1] Each rule specification is passed to the rewriter (*cf.* Section 2.3) for compilation, with $As_1, \cdots, As_m$, S as a redex and $S_1, \cdots, S_n$ as a contractum.

Before admitting a rule specification MetaPRL rewriter checks that each context variable occurring in one of the S_i or S is also listed as one of the arguments. It also makes sure that any second-order or context variable occurring anywhere in the rule specification (*i.e.* in one of the S_i, S or As_i) has a single [2] "reference" occurrence in the redex (*i.e.* in one of the As_i or S) such that:

1. The reference occurrence can not be inside the argument of another context or second-order variable occurrence.

[1] This is a simplification. There are other kinds of arguments that are used to specify the refinement of the parameter meta-variables described in Section 3.7.

[2] If the same variable has several occurrences that qualify for the "reference status", the first one will be used as a "reference" one.

2. All arguments of the reference occurrence of our variable must be distinct bound first-order variables.

When the rule is applied the reference occurrences will be used by the rewriter to determine the value of the substitution function on that variable. For second-order variables this is accomplished by compiling reference occurrences to `so_var` instruction, while all the other occurrences will be compiled to `match_term` instructions.

MetaPRL compiles each rules specification to *a primitive tactic* capable of applying the specified rule. In order to apply the rule to a goal sequent schema G, this tactic will be passed (by the user or by a higher-level tactic) arguments $A_1, \ldots, A_m$. Because of the conditions (1)–(2) above, there may only be at most one list of subgoals $G_1, \ldots, G_n$ such that $A_1, \ldots, A_m$, G, $G_1, \ldots, G_n$ is a valid instance of $As_1, \cdots, As_m$, S, $S_1, \cdots, S_n$. Invoking the tactic would cause the system to make sure that

- For each $1 \leq i \leq m$, if As_i is a context variable, A_i is *an address* that specifies which hypothesis specifications of G should be matched to that context variable; and

- If As_i is a term schema, then A_i is a closed term. [3]

The system would then pass $A_1, \ldots, A_m$, G to the rewriter as a potential match for the redex $As_1, \cdots, As_m$, S and the rewriter will output the corresponding subgoals $G_1, \ldots, G_n$, if they exist.

From the user's perspective the argument schemata provide a way for passing into the rewriter all the information that it would not be able to deduce by match-

[3] As in Section 4.1 this requirement will be used to distinguish second-order 0 arity variables from first-order ones.

Table 4.1: Decomposition Tactic Examples

	Goal sequent		Desired subgoals
Conclusion decomposition	$\cdots \vdash A \wedge B$	$\Longrightarrow$	$\cdots \vdash A$ and $\cdots \vdash B$
Hypothesis decomposition	$\cdots; A \wedge B; \cdots \vdash \cdots$	$\Longrightarrow$	$\cdots; A; B; \cdots \vdash \cdots$

ing G with S. For example, if S does not contain reference occurrences of some of the variables used in the rule, the user has to add extra arguments that contain such occurrences.

4.3 Rule Annotations

Some basic tactics are often designed to behave very differently in different contexts. One of the best examples of such tactic in a *decomposition tactic* [Jac95, Section 3.3] present both in NuPRL (where it is called "`D`") and in MetaPRL (where it is called "`dT`"). When applied to the conclusion of a goal sequent, it will try to decompose the conclusion into simpler ones, normally by applying [4] an appropriate introduction rule. When applied to a hypothesis, the decomposition tactic would try to break the hypothesis into simpler ones, usually by applying an appropriate elimination rule.

Example 4.3.1. The desired behavior for the decomposition tactic on $\wedge$-terms is shown in Table 4.1.

Whenever a theory is extended with a new operator, decomposition tactic needs to be updated in order for it to know how to decompose this new operator. More generally, whenever a new rule (including a new axiom, a new derived rule or a

[4]Since proof search is done from the conclusion to premises, it may be more accurate to say that the tactic will *back-chain* through the rule.

new theorem) is added to a system it is often desirable to update some tactic (or possibly several tactics) so that it makes use of the newly added rule. In order to facilitate such updates authors of most commonly used "updatable" tactics would often provide "hooks" allowing one to specify how the new rule should be used without having to go back and modify the code of the tactic.

Normally, such a "hook" would require users to specify which rule or tactic should be added as well as when to make use of this new rule (or tactic). For example, after a $\wedge$ introduction rule is added to the system, the decomposition tactic would be updated with the information that if the conclusion is a $\wedge$-term, then the new introduction rule should be used.

It turns out that writing such tactic updates takes some expertize and is time consuming, even in the presence of "hooks". This often prevents users from making full use of the tactic update functionality. On the other hand it also turns out that most of the needed information is already present in the system when the new rule is added — if the rules are expressed using a well-defined logical meta-language (such as the sequent schemata language), then we can use the text of the rules itself as a source of information.

Consider a situation when a new rule is added to the system and the system is notified that the decomposition tactic need to be updated. If the system would be given access both to the *text* of the rule and the primitive tactic for *applying* the rule, it will have most (if not all) of the information on how to update the decomposition tactic! It is clear what tactic should be added to the decomposition strategy — the primitive tactic that would apply the newly added rule. And by examining the text of the rule, the conditions for applying it can be usually deduced directly. For example, if the new rule is an `xyz` introduction rule for some operator

`xyz`, then we know that it should be applied whenever the decomposition tactic is used to decompose an `xyz` conclusion.

In a way this approach makes a system reflexive — it becomes capable of using not only the *meaning* of the rules in its proof search, but their *syntax* as well.

In MetaPRL this approach is implemented via a mechanism of *resource annotations* on the rules. An annotation would cause a special kind of "hook" function to be invoked, which would receive the rule text and the primitive tactic as inputs. When necessary, the user can supply an extra argument to serve as an additional hint for the "hook" function.

Example 4.3.2. The resource annotation for the $\wedge$ introduction rule in MetaPRL would be written simply as

```
{| intro [] |}
```

which specifies that the conclusion decomposition tactic needs to be notified of the new rule.

The resource annotation for the $\wedge$ elimination rule in MetaPRL would be written as

```
{| elim [ThinOption thinT] |}
```

which specifies that the conclusion decomposition tactic needs to be notified of the new rule and that by default it should thin out the original $\wedge$ hypothesis after applying the elimination rule.

4.4 Decision Procedures as Heuristics

Another very interesting category of proof automation procedures in a theorem prover is automated decision procedures. A traditional approach to implementing

a decision procedure in a theorem prover is to implement it as a part of the *trusted* code base — whenever the decision procedures says "Yes" the prover would immediately consider the current goal to be proven. Unfortunately, there are many disadvantages to this approach. In particular, this increases the size of the trusted code. It usually requires a significant effort to prove that the chosen decision procedure is valid and is compatible with the logical theory being used.

However the biggest price of "trusted" decision procedures is the loss of flexibility. Changing or extending the logical theory will now require having to invest additional time into making sure that the modified theory is still compatible with all the assumption all the trusted decision procedures make. Improving trusted decision procedures becomes equally hard. In a way, trusted decision procedures and the logical theory become "locked in" together with either one becoming much harder (and much more "expensive") to modify.

As it turns out, proving that the decision procedure was correct *in a particular instance* is much easier then proving that it will *always* be correct and can often be established automatically. This gives us an alternative when integrating a decision procedure into an interactive theorem prover, provided it can be enhanced to output some evidence along with the "yes" answer. Such decision procedure can be implemented as a tactic rather than being a trusted code. The new tactic would check (using the decision procedure) whether the current goal is valid and if the answer is "yes", it would try interpreting the evidence outputted by the decision procedure and use it to re-prove the goal.

In a way such tactic becomes a heuristic that can be still used even when the theory is not known to be completely compatible with the decision procedure (or even when it is known to be incompatible). The worse that could happen is that

in certain cases the tactic would fail or will produce an incomplete proof (e.g. produce some subgoals that will still be left to prove).

4.4.1 JProver

This approach was used by Stephan Schmitt for implementing the JProver decision procedure in MetaPRL. JProver is an automated prover for first-order intuitionistic and classical logic based on the connection method [Bib87, KO99], with a tool for generating proof objects in the style of sequent proofs [KS00].

JProver is implemented on top of MetaPRL core in a very generic way, using MetaPRL as a theorem proving toolkit. JProver takes as its input a small `JLogic` module that represents the logic of the proof assistant with which JProver will cooperate. The `JLogic` module describes which terms implement logical connectives, how to access subterms from those connectives, and how to convert JProver's generic representation of a sequent proof into the internal data structures of the proof assistant.

In order to be able to call JProver from some proof assistant, one would need to write a *logic module* that consists of two components: a piece of OCaml code for communicating with that proof assistant (using whatever communication protocol developers would choose) and a `JLogic` module capable of decoding the sequent received from that communication code and of encoding JProver's response into a form the communication code expects.

As described in [SLKN01], JProver was integrated into the MetaPRL implementation of the NuPRL type theory and into the NuPRL-4 system. Later, Huang Guan-Shieng have integrated JProver into Coq proof assistant. [5]

[5] See `http://coqcvs.inria.fr/cgi-bin/cvswebcoq.cgi/~checkout~/V7/`

4.4.2 Arithmetic

In his work on formalizing arithmetic in MetaPRL Yegor Bryukov also takes advantage of this approach. Using techniques developed by Tobias Mayr [May97] he implements arithmetical decision procedures as tactics with explicitly formulating all the arithmetical axioms.

4.5 Generic tactic layer

Sections 4.3 and 4.4 show two complimentary approaches to building a rich layer of *generic tactics*. The great advantage of such generic tactics is that they can be implemented once and then reused in a wide range of logical theories with no or a little additional effort. And in the JProver case Huang Guan-Shieng was able to integrate the existing JProver code into Coq assistant without having any previous familiarity with the source MetaPRL system and without ever asking any member of MetaPRL group for advice. This example shows that our generic tactics approach is very viable not only for sharing across various logical theory is a single logical framework, but also for sharing across a variety of different theorem provers.

Both approaches to generic tactics are essentially trading in a human-intensive approach for a computer-intensive one. In case of updatable tactic we have the system itself extracting the relevant information from the text of the rules, instead of requiring users to provide it. In case of decision procedures we eliminate the need for manually establishing the validity of a procedure, replacing it with a computer system for post-processing proofs that come out of the procedure. In [Arm00] Bill Arms investigates a similar trend in the area of digital libraries. He notes that "computing power is much cheaper than human expertise, more so every year"

`contrib/jprover/README` for more information on Coq JProver integration.

and he argues that as computers become more powerful, the switch to computer-intensive approaches will keep gaining attractiveness.

Part II

Type Theory

Chapter 5

Quotient Types — A Modular Approach

The NuPRL type theory differs from most other type theories used in theorem provers in its treatment of equality. In Coq's Calculus of Constructions, for example, there is a single global equality relation which is not the desired one for many types (e.g. function types). The desired equalities have to be handled explicitly, which is quite burdensome. As in Martin-Löf type theory [ML82] (of which the NuPRL type theory is an extension), in NuPRL each type comes with its own equality relation (the extensional one in the case of functions), and the typing rules guarantee that well-typed terms respect these equalities. Semantically, a quotient of a given NuPRL type is trivial to define: it is just the same type equipped with a new equality relation.

Such quotient types have proved to be an extremely useful mechanism for natural formalization of various notions in type theory. For example, rational numbers can be naturally formalized as a quotient type of the type of pairs of integer numbers (which would represent the numerator and the denominator of a fraction) with the appropriate equality predicate.

Somewhat surprisingly, it turns out that formulating rules for these quotient types is far from being trivial and numerous applications of NuPRL [BCH+00, LKvR+99] have run into difficulties. Often a definition involving a quotient looks plausible, but after some (sometimes substantial) work it turns out that some key property is unprovable, or false.

A common source of problems is that in the NuPRL type theory all true equality predicates are uniformly witnessed by a single canonical constant. This means that

even when we know that two elements are equal in a quotient type, we can not in general recover the witness of the equality predicate. In other words, $a = b \in (A/\!/E)$ (where "$A/\!/E$" is a quotient of type A with equivalence relation E) does not always imply $E[a;b]$ (however it does imply $\neg\neg E[a;b]$).

Another common class of problems occurs when we consider some predicate P on type A such that we can show that $P[a] \Leftrightarrow P[b]$ for any $a, b \in A$ such that $E[a;b]$. Since $P[a] \Leftrightarrow P[b]$ does not necessary imply that $P[a] = P[b]$, P may still turn out not to be a well-formed predicate on the quotient type $A/\!/E$. [1]

These problems suggest that there is more in the concept of quotient types, than just the idea of changing the equality relation of a type. In this chapter we show how we can decompose the concept of quotient type into several simpler concepts and to formalize quotient types based on formalization of those simpler concepts.

We claim that such a "decomposed" theory makes operating with quotient types significantly easier. In particular we show how the new type constructors can be used to formalize the notion of indexed collections of objects. We also claim that the "decomposition" process makes the theory more modular. In particular, we show how to reuse one of the new type constructors to improve and simplify the rules for the set type.

For each of the new (or modified) type constructors, we present a set of derivation rules for this type — both the axioms to be added to the type theory and the rules that can be derived from these axioms. As we will explain in Section 5.3, the particular axioms we use were carefully chosen to make the theory as modular as possible and to make them as usable as possible in a tactic-based interactive

[1] P will be a function from $A/\!/E$ to $\mathbf{Prop}/\!/\Leftrightarrow$, rather than a predicate (a function from $A/\!/E$ to $\mathbf{Prop}$), where $\mathbf{Prop}$ is a type (universe) of propositions.

prover. All the new rules were checked and found valid in S. Allen's semantics of type theory [All87a, All87b]; these proofs are rather straightforward, so we omit them here. Proofs of all the derived rules were developed and checked in the MetaPRL system [Hic01, HNK+, HAB+].

Although we focus on the NuPRL type theory, we believe that many ideas presented here are relevant to managing witnessing and functionality information in a constructive setting in general.

This chapter is organized as follows. First, in Section 5.1 we describe some features of the NuPRL type theory that are necessary for understanding this work. Sections 5.2, 5.5, 5.6 and 5.7 each describe a module introducing its own primitive constructor that together form the new axiomatization of the quotient type constructor. In Section 5.4 we demonstrate the modularity advantages of the new axiomatization by reusing some of the previously presented modules to simplify and enhance the formalization of set types. Section 5.3 explains our approach and motivations in choosing particular axioms. Finally, Section 5.8 shows how the new axiomatization allows us to formalize the notion of collections that seems impossible to formalize in the original axiomatization.

Previous revisions of this work were published as [Nog02a, Nog02b].

5.1 NuPRL Type Theory

5.1.1 Propositions-as-Types

The NuPRL type theory adheres to the *propositions-as-types principle*. This principle means that a proposition is identified with the type of all its witnesses. A proposition is considered true if the corresponding type is inhabited and is consid-

ered false otherwise. In this chapter we will use words "*type*" and "*proposition*" interchangeably; same with "*witness*" and "*member*".

5.1.2 Partial Equivalence Relations Semantics

The key to understanding the idea of quotient types is understanding the most commonly used semantics of the NuPRL type theory (and some other type theories as well) — the PER (partial equivalence relations) semantics [Tro73, All87a, All87b]. In PER semantics each type is identified with a set of objects and an equivalence relation on that set that serves as an *equality relation* for objects of that type. This causes the equality predicate to be three-place: "$a = b \in C$" stands for "a and b are equal elements of type C", or, semantically, "a and b are related by the equality relation of type C".

Remark 5.1.1. Note that in this approach an object is an element of a type *iff* it is equal to itself in that type. This allows us to identify $a \in A$ with $a = a \in A$.

According to PER approach, whenever something ranges over a certain type, it not only has to span the whole type, it also has to respect the equality of that type.

Example 5.1.2. In order for a function f to be considered a function from type A to type B, not only for every $a \in A$, $f(a)$ has to be B, but also whenever a and a' are equal in the type A, $f(a)$ should be equal to $f(a')$ in the type B. Note that in this example the second condition is sufficient since it actually implies the first one. However it is often useful to consider the first condition separately.

Example 5.1.3. Now consider a set type $T := \{x : A \mid B[x]\}$ (cf. Section 5.4). Similarly to Example 5.1.2 above, in order for T to be a well-formed type, not only

$B[a]$ has to be a well-formed type for any $a \in A$, but also for any $a = a' \in A$ it should be the case that $B[a]$ and $B[a']$ are equal.

5.1.3 Extensional and Intensional Approaches

In this chapter we devote significant amount of attention to discussion of choices between what we call *intensional* and *extensional* approaches to certain type operators. The difference between these approaches is in deciding when two objects should be considered equal. In general, in the intensional approach two objects would be considered equal if their *internal* structure is the same, while in the extensional approach two objects would be considered equal if they exhibit the same external behavior.

Example 5.1.4. In the NuPRL type theory the function equality is extensional. Namely, we say that $f = f' \in (A \to B)$ *iff* they both are in $A \to B$ and for all $a \in A$, $f(a) = f'(a) \in B$. [2]

Example 5.1.5. It is easy to define an extensional equality on types: $A \; =_{\mathsf{e}} B$ *iff* A and B have the same membership and equality relations. [3] However, in the NuPRL type theory the main equality relation on types is intensional. For example, if A and B are two non-empty types, then $(A \to \bot) = (B \to \bot)$ only when $A = B$, even though we have $(A \to \bot) \; =_{\mathsf{e}} (B \to \bot)$ since they are both empty types. [4]

---

[2]It is interesting to note that this causes the type $\bot \to \bot$ (where $\bot$ is an empty type) to be a type that all functions belong to and are all equal in.

[3]We will discuss extensional equality in more detail in Section 6.1.1.

[4]Strictly speaking, the NuPRL type theory does not contain Martin-Löf's "$A = B$" judgment form. Instead, NuPRL uses proposition of the form $A = B \in \mathbb{U}_i$ where $\mathbb{U}_i$ is the i-th universe of types. However in this chapter we will often omit "$\in \mathbb{U}_i$" for simplicity.

Example 5.1.6. Some type constructors, such as a set (cf. Section 5.4) or a quotient (cf. Section 5.7) one, include a predicate. When introducing such a constructor into the NuPRL type theory, there are often two choices for handling the predicate in the equality definition:

Completely intensional. For two types to be equal, their corresponding predicates have to be equal as well. For example $\{x : A \mid B[x]\} = \{x : A' \mid B'[x]\}$ iff $A = A'$ and for all $a = a' \in A$, $B[a]{=}B'[a']$.

Somewhat extensional. The predicates have to imply one another, for example $\{x : A \mid B[x]\} = \{x : A' \mid B'[x]\}$ iff $A = A'$ and for all $a = a' \in A$, $B[a]{\Leftrightarrow}B'[a']$.

Essentially, in the intensional case the map $x \rightsquigarrow B[x]$ has to respect A's equality relation in order for $\{x : A \mid B[x]\}$ to be well-formed and in the extensional case $B[x]$ only needs to respect it up to $\Leftrightarrow$ (logical *iff*).

We will continue the discussion of the differences between these two choices in Sections 5.2.3 and 5.7.1.

5.2 Squash Operator

5.2.1 Squash Operator: Introduction

The first concept that is needed for our formalization of quotient types is that of hiding the witnessing information of a certain true proposition thus only retaining the information that the proposition is known to be true while hiding the information on *why* it is true.

To formalize such notion, for each type A we define a type $\left[A\right]$ ("squashed A") which is empty *if and only if* A is empty and contains a single canonical element

$\bullet$ [5] when A is inhabited. Informally one can think of $[A]$ as a proposition that says that A is a *non-empty type*, but "squashes down to a point" all the information on *why* A is non-empty. The `squash` operator is *intensional*, e.g. $[A] = [B]$ iff $A = B$ (see also Remark 5.2.2).

Remark 5.2.1. In [Jac95, Section 3.7.3] Paul Jackson defined the `squash` operator as $[A] := \{x : Unit \mid A\}$. However here our goal is to formalize the set type using the `squash` operator and not the other way around, so we do not use this definition.

The `squash` operator (sometimes also called `hide`) was introduced in [CAB+86]. It is also used in [Cal98, Jac95, Hic01, HAB+] [6].

In the next section we will present the axiomatization we chose for the `squash` operator and we will explain our choices in Section 5.3.

5.2.2 Squash Operator: Axioms

First, whenever A is non-empty, $[A]$ must be non-empty as well:

$$\frac{\Gamma \vdash A}{\Gamma \vdash [A]} \qquad (SquashIntro)$$

Second, if we know $[A]$ and we are trying to prove an equality (or a membership) statement, we can allow "unhiding" contents of A and continue with the proof:

$$\frac{\Gamma;\, x : A;\, \Delta \vdash t_1 = t_2 \in C}{\Gamma;\, x : [A];\, \Delta \vdash t_1 = t_2 \in C} \qquad (SquashElim)$$

---

[5] MetaPRL system uses the unit element () or "it" as a $\bullet$, NuPRL uses `Ax` and [Tho91] uses *Triv*.

[6] In MetaPRL `squash` was first introduced by J.Hickey as a replacement for NuPRL's hidden hypotheses mechanism, but eventually it became clear that it gives a mechanism substantially widely useful than NuPRL's hidden hypotheses.

(assuming x does not occur free in Δ, t_i and C — we use the sequent schemata syntax of Chapter 3 for specifying rules.) This rule is valid because in Martin-Löf type theory equality has no *computational context* and is always witnessed by $\bullet$, so knowing the witness of A does not add any "additional power".

Finally, the only possible element of a `squash` type is $\bullet$:

$$\frac{\Gamma;\, x : \left[A\right];\, \Delta[\bullet] \;\vdash\; C[\bullet]}{\Gamma;\, x : \left[A\right];\, \Delta[x] \;\vdash\; C[x]} \qquad (\textit{SquashMemElim})$$

As mentioned in the introduction, all these new axioms can be proved sound in Allen's semantics of type theory [All87a, All87b]. All soundness proofs are very straightforward, and we omit them in this chapter. We also omit some purely technical axioms (such as well-formedness ones) that are unnecessary for understanding this work. [7]

5.2.3 `Squash` Operator: Derived Rules

Here are the rules that can be derived from the axioms we have introduced above. First, whenever $\left[A\right]$ is non-empty, $\bullet$ must be in it:

$$\frac{\Gamma \;\vdash\; \left[A\right]}{\Gamma \;\vdash\; \bullet \in \left[A\right]} \qquad (\textit{SquashMemIntro})$$

Second, using (*SquashMemElim*) we can prove a stronger version of (*SquashElim*):

$$\frac{\Gamma;\, x : A;\, \Delta[\bullet] \;\vdash\; t_1[\bullet] = t_2[\bullet] \in B[\bullet]}{\Gamma;\, x : \left[A\right];\, \Delta[x] \;\vdash\; t_1[x] = t_2[x] \in B[x]} \qquad (\textit{SquashElim2})$$

Third, we can prove that squashed equality implies equality:

$$\frac{\Gamma \;\vdash\; \left[t_1 = t_2 \in A\right]}{\Gamma \;\vdash\; t_1 = t_2 \in A} \qquad (\textit{SquashEqual})$$

[7] The full listing of the MetaPRL rules should be available online [HAB$^+$]. An earlier version of MetaPRL implementation of type theory is presented in [Hic01, Section 14].

Remark 5.2.2. Note that if we would have tried to make the `squash` operator extensional, we would have needed an extra well-typedness assumption in the (*SquashElim*) rule (as we had to do in (*EsquashElim*) rule in Section 5.7.2) which would have made it useless for proving well-typedness and membership statements. In particular, the (*SquashEqual*) rule (as well as any reasonable modification of it) would not have been valid.

Next, we can prove that if we can deduce a witness of a type A just by knowing that some unknown x is in A (we call such A a *squash-stable* type — see Section 7.3), then $\lceil A \rceil$ implies A:

$$\frac{\Gamma \vdash \lceil A \rceil \qquad \Gamma; x : A \vdash t \in A}{\Gamma \vdash A} \qquad (\mathit{SquashStable})$$

Finally, we can prove that we can always eliminate the squashes in hypotheses not only when the conclusion is an equality (as in (*SquashElim*) and (*SquashElim2*)), but also when it is a squash [8]:

$$\frac{\Gamma; x : A; \Delta[\bullet] \vdash \lceil C[\bullet] \rceil}{\Gamma; x : \lceil A \rceil; \Delta[x] \vdash \lceil C[x] \rceil} \qquad (\mathit{Unsquash})$$

5.3 Choosing the Rules

For each of the concepts and type operators we discuss in this chapter there might be numerous different ways of axiomatizing it. When choosing a particular set of axioms we were using several general guidelines.

First, in a context of an interactive tactic-based theorem prover it is very important to ensure that each rule is formulated in a *reversible* way whenever possible. By reversible rule we mean a rule where conclusion is valid *if and only if* the

[8] In general, it is true whenever the conclusion is squash-stable.

premises are valid. This means that it is always "safe" to apply such a rule when (backward) searching for a proof of some sequent — there is no "danger" that back-chaining through the rule would turn a provable statement into a statement that is no longer true. This property allows us to add such rules to proof tactics more freely without having to worry about a possibility that applying such tactic can bring the proof into a "dead end". [9] For example, among the `squash` axioms of Section 5.2.2 only (*SquashIntro*) is irreversible and the other axioms are reversible.

Second, we wanted to make sure that each rule makes the smallest "step" possible. For example, the (*SquashElim*) rule only eliminates the `squash` operator, but does not attempt to eliminate the witness of the `squash` type while the (*SquashMemElim*) only eliminates the witness of the `squash` type and does not attempt to eliminates the `squash` operator. This gives users a flexibility to "steer" proofs exactly where they want them to go. Of course, we often do want to make several connected steps at once, but that can be accomplished by providing derived rules [10] while still retaining the flexibility of the basic axioms. For example, the (*SquashElim2*) allows one to both eliminate the `squash` operator and its witness in a single step, while still using (*SquashElim*) or (*SquashMemElim*) when only one and not the other is needed. As we will see in Section 5.4 this "small step" requirement is especially important for the irreversible rules.

Finally, it is important for elimination rules to match corresponding intro-

[9]Of course if a tactic is designed to fall back when it fails to find a complete derivation for the statement being proved, it would not become dangerous when we allow it to use an irreversible rule (although it might become more likely to fail). But if a tactic is only meant to propel the proof further without necessarily completing it (such as for example NuPRL's `Auto` and MetaPRL's `autoT`), then allowing such tactic to use irreversible rules can make things substantially less pleasant to the user.

[10]See Chapter 3 for a description of MetaPRL's derived rules mechanism.

duction rules in their "power". [11] Such balance helps insure that most rules are reversible not only with respect to validity, but also with respect to provability (which is obviously needed to make applying such rules truly "safe" in a theorem prover).

5.4 Intensional Set Type

5.4.1 Set Type: Introduction

The decomposition of the axiomatization of quotient types into smaller pieces has an additional advantage (besides making quotient types easier to reason about) of making the theory more modular. The type operators that we use for formalizing quotient types can be now reused when formalizing other types as well. To illustrate this, we will show how the traditional formalization of the set types can be greatly improved and simplifies using the `squash` operator.

Informally, $\{x : A \mid B[x]\}$ is a type containing all elements $x \in A$ such that $B[x]$ is a true proposition. The key property of set type is that when we have a witness $w \in \{x : A \mid B[x]\}$, we know that $w \in A$ and we know that $B[w]$ is non-empty; but in general we have no way of reconstructing a witness for $B[w]$.

5.4.2 Set Type: Traditional Approach

Set types were first introduced in [Con83] and were also formalized in [Bac84, CAB+86, Hic01, NP83, NPS90]. In those traditional implementations of type

[11] More specifically, elimination rules should be *locally complete* and *locally sound* with respect to the introduction rules, as described in [PD01]. But since we believe that this third guideline is not as crucial as the first two, we chose not provide a detailed discussion of it.

theory the rules for set types are somewhat asymmetric. When proving something like

$$\frac{\Gamma;\, y : A \vdash y \in A' \qquad \Gamma;\, y : A;\, z : B[y] \vdash B'[y]}{\Gamma;\, y : \{x : A \mid B[x]\} \vdash y \in \{x : A' \mid B'[x]\}}$$

one was forced to apply the set elimination rule before the set introduction rule. As we will see in a moment, the problem was that the traditional set introduction rule is irreversible and would go "too far" if one applies it right away. It would yield a subgoal $\Gamma;\, y : \{x : A \mid B[x]\} \vdash B'[y]$ that would only be valid if one could reconstruct a proof witness of $B'[y]$ without having access to the witness of $B[y]$.

5.4.3 Set Type: A New Approach

Using the `squash` operator we only need [12] the following two simple axioms to formalize the set type:

$$\frac{\Gamma;\, y : A;\, z : \big[B[y]\big];\, \Delta[y] \vdash C[y]}{\Gamma;\, y : \{x : A \mid B[x]\};\, \Delta[y] \vdash C[y]} \qquad (\mathit{SetElim})$$

$$\frac{\Gamma \vdash t \in A \qquad \Gamma \vdash \big[B[t]\big] \qquad \Gamma \vdash \{x : A \mid B[x]\}\ \mathrm{Type}}{\Gamma \vdash t \in \{x : A \mid B[x]\}} \qquad (\mathit{SetIntro})$$

Now we can explain the problem with the traditional approach [Con98, CAB^{+}86, NP83, NPS90, Tho91] — there the set introduction rule is somewhat analogous to applying (*SetIntro*) and then as much (*Unsquash*) as possible and then (*SquashIntro*). Such rule does too many things at once and one of those things (*SquashIntro*) is irreversible. With such rule we can only deconstruct the set operator in the conclusion when the irreversible part of this rule would not render the resulting subgoals unprovable.

The reason this traditional formalization required a rule that does so much at once was the lack of a way to express the intermediate results. In a sense, in that

[12] As in Section 5.2.2 we omit some unessential axioms.

implementation, set (and quotient) types had at least two separate "jobs" — one was to change the type membership (equality) and another — to hide the proof of the membership (equality) predicate. And there was only a single collection of rules for both of the "jobs", which made the rules hard to use.

The `squash` operator now takes over the second "job" which allows us to express the properties of each of the two jobs in a separate set of rules. Our rules (*SetElim*) and (*SetIntro*) are now both reversible, both perform only a singly small step of the set type and they exactly match each other. The set introduction rule now does only that — introduces the set type into the conclusion of the sequent and leaves it to the `squash` rules (such as (*Unsquash*) and (*SquashIntro*)) to manage the "hiding/unhiding the proof predicate" aspect. We believe this makes the theory more modular and easier to use.

5.5 Extensional `Squash` Operator (`Esquash`)

5.5.1 `Esquash` Operator: Introduction

The second concept that is needed for our formalization of quotient types is that of "hiding" the intensional structure of a certain proposition; essentially we need the concept of "being extensional" — as we will see in Section 5.7, even the intensional quotient type has some extensionality in it. In order to make the theory modular, we want to express the concept of the extensionality directly, not through some complex operator for which the extensionality is just a "side-effect". As we mentioned in Remark 5.2.2, the `squash` operator needs to be intensional, so we will need to define a new operation.

The operation we will use, called `esquash`, acts very similar to `squash` except

for "esquashed" types being equal whenever they are simultaneously non-empty or simultaneously empty. This way `esquash` completely "hides" both the witnesses of a type and its intensional structure, leaving only the information on whether a type is non-empty or not.

5.5.2 `Esquash` Operator: Axioms

First, equality — two `esquash` types are equal *iff* they are simultaneously true or simultaneously false:

$$\frac{\Gamma \vdash [\![A]\!] \Leftrightarrow [\![B]\!]}{\Gamma \vdash [\![A]\!] = [\![B]\!]} \qquad (\mathit{EsquashEquality})$$

Second, `esquash` of an intensional type is equivalent to `squash` [13]:

$$\frac{\Gamma \vdash [\![A]\!] \qquad \Gamma \vdash A\,\text{Type}}{\Gamma \vdash [A]} \qquad (\mathit{EsquashElim})$$

$$\frac{\Gamma \vdash [A]}{\Gamma \vdash [\![A]\!]} \qquad (\mathit{EsquashIntro})$$

Finally, the only member of a non-empty `esquash` type is $\bullet$:

$$\frac{\Gamma;\, x : [\![A]\!];\, \Delta[\bullet] \vdash C[\bullet]}{\Gamma;\, x : [\![A]\!];\, \Delta[x] \vdash C[x]} \qquad (\mathit{EsquashMemElim})$$

Remark 5.5.1. We could define the `esquash` operator as

$$[\![A]\!]_i \;:=\; A = \text{True} \in \big(x, y : \mathbb{U}_i /\!/ (x \Leftrightarrow y)\big)\;.$$

Unfortunately, this definition increases the universe level. With this definition if $A \in \mathbb{U}_i$, then $[\![A]\!]_i$ is in $\mathbb{U}_{i+1}$. This can create many difficulties, especially when we want to be able to iterate the `esquash` operator. And in any case we want to formalize quotient types using the `esquash` operator, not the other way around.

[13] $x : T \vdash [\![A[x]]\!]$ only requires $A[x]$ to be non-empty when $x \in T$. However since `squash` is intensional, $x : T \vdash \big[A[x]\big]$ also requires $A[x] = A[x']$ when $x = x' \in T$. Because of this we need the well-typedness condition in (*EsquashElim*).

Remark 5.5.2. In MetaPRL J. Hickey had initially defined an `esquash` operator using the extensional quotient [14]:

$$[\![A]\!] := \texttt{tt} = \texttt{ff} \in (x, y : \mathbb{B} \mathbin{/\!/_e} (x = y \in \mathbb{B} \vee A)) \ .$$ [15]

This definition does not increase the universe level like the previous one, but on the other hand it requires an extensional quotient type while the previous one works with both intensional and extensional quotients. Another problem with this definition is that almost all NuPRL-4 rules on quotient types require one to prove that the equality predicate is actually intensional, so it would be impossible to prove the properties of `esquash` from this definition using NuPRL-4 rules.

5.5.3 `Esquash` Operator: Derived Rules

First, using (*EsquashMemElim*) we can prove that any non-empty `esquash` type has an • in it:

$$\frac{\Gamma \vdash [\![A]\!]}{\Gamma \vdash \bullet \in [\![A]\!]} \qquad (\mathit{EsquashMemIntro})$$

Second, we can derive a more general and complex version of (*EsquashElim*):

$$\frac{\Gamma;\, x : [A];\, \Delta[x] \vdash B[x] \qquad \Gamma;\, x : [\![A]\!];\, \Delta[x] \vdash A\,\mathrm{Type}}{\Gamma;\, x : [\![A]\!];\, \Delta[x] \vdash B[x]} \qquad (\mathit{EsquashElim2})$$

5.6 Explicit Nondeterminicity

5.6.1 Explicit Nondeterminicity: Introduction

The third piece of the quotient puzzle is the explicit nondeterminicity type. At first, this type was introduced just as a technical trick allowing us to express

[14]See Section 5.7.1 for more on extensional and intensional quotient types.

[15]Where $\mathbb{B}$ is the type of booleans and `tt` ("true") and `ff` ("false") are its two members.

stronger elimination rules for quotient types. However it seems that this notion is also a useful tool to have on its own.

At first, we considered adding the nd operation similar to *amb* in [McC63] and to the approach used in [How96]. The idea was to have $\mathsf{nd}\{t_1; t_2\}$ which can be either t_1 or t_2 nondeterministically. Then we were going to say that the expression that contains nd operators is well-formed iff its meaning does not depend on choosing which of nd's arguments to use. The problem with such an approach is that we need some way of specifying that several occurrences of the same $\mathsf{nd}\{t_1; t_2\}$ have to be considered together — either all of them would go to t_1 or all of them would go to t_2. For example, we can say that $\mathsf{nd}\{1; -1\}^2 = 1 \in \mathbb{Z}$ (which is true), but if we expand the ^{2} operator, we will get $\mathsf{nd}\{1; -1\} * \mathsf{nd}\{1; -1\} = 1 \in \mathbb{Z}$ which is only true if we require both nd's in it to expand to the same thing.

The example above suggests using some index on nd operator, which would keep track of what occurrences of nd should go together. In such a case it is natural for that index to be of the type $(\mathbb{B}/\!/\mathrm{True})$ and as it turns out, this type represents a key idea that is worth formalizing on its own. As "usual", since we want to express the properties of the quotient types using the $\mathsf{ND} == (\mathbb{B}/\!/\mathrm{True})$ type, it can not be defined using the quotient operator and needs to be introduced as a primitive.

The basic idea behind this ND type is that it contains two elements, say `tt` and `ff` and $\mathtt{tt} = \mathtt{ff} \in \mathsf{ND}$. In addition to these two constants we also need the `if ... then ... else ... fi` operator such that `if tt then` t_1 `else` t_2 `fi` is computationally equivalent to t_1 and `if ff then` t_1 `else` t_2 `fi` is computationally equivalent to t_2. For simplicity, we "borrow" these constants and this operator from $\mathbb{B}$ [16], but we could have created new ones, it does not really matter here. We

[16]Which means that ND is just $\mathbb{B}/\!/\mathrm{True}$.

will write "$\mathsf{nd}_x\{t_1; t_2\}$" as an abbreviation for "$\mathtt{if}\ x\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2\ \mathtt{fi}$".

5.6.2 Explicit Nondeterminicity: Axioms

$$\frac{\Gamma;\, u : A[\mathtt{tt}] = A[\mathtt{ff}];\, y : A[\mathtt{tt}];\, x : \mathsf{ND};\, \Delta[x; y] \,\vdash\, C[x; y]}{\Gamma;\, x : \mathsf{ND};\, y : A[x];\, \Delta[x; y] \,\vdash\, C[x; y]} \quad (\textit{ND-elim})$$

$$\frac{\Gamma \,\vdash\, C[\mathtt{tt}] = C[\mathtt{ff}] \qquad \Gamma \,\vdash\, C[\mathtt{tt}]}{\Gamma;\, x : \mathsf{ND} \,\vdash\, C[x]} \quad (\textit{ND-elim2})$$

Notice that (*ND-elim*) does not completely eliminate the ND hypothesis, but only "moves" it one hypothesis to the right, so to completely eliminate the ND hypothesis, we will need to apply (*ND-elim*) repeatedly and then apply (*ND-elim2*) in the end.

For the purpose of formalizing the quotient operators we only need the two rules above. A complete formalization of ND would also include the axiom

$$\frac{}{\vdash \mathtt{tt} = \mathtt{ff} \in \mathsf{ND}} \quad (\textit{ND-intro})$$

5.6.3 Explicit Nondeterminicity: Derived Rule

$$\frac{\Gamma \,\vdash\, t[\mathtt{tt}] = t[\mathtt{ff}] \in A}{\Gamma;\, x : \mathsf{ND} \,\vdash\, t[x] \in A} \quad (\textit{ND-memb})$$

5.7 Intensional Quotient Type

5.7.1 Quotient Type: Introduction

The quotient types were originally introduced in [CZ84]. They are also presented in [CAB$^+$86, Tho91].

While extensional quotient type can be useful sometimes, usually the intensional quotient type is sufficient and the extensionality just unnecessary complicates proofs by requiring us to prove extra well-typedness statements. In addition

to that NuPRL formalization of quotient types (see Appendix A.2 and [CAB+86]) does not allow one to take full advantage of extensionality since most of the rules for the quotient type have an assumption that the equality predicate is in fact intensional. While early versions of the NuPRL type theory considered extensional set and quotient types, these problems forced the change of set constructor (which is used substantially more often than the quotient) into an intensional one.

In order to avoid the problems outlined above, in this chapter we introduce the intensional quotient type as primitive, and we concentrate our discussion of quotient types on intensional quotient types. But since we have the `esquash` operator in our theory, an extensional quotient type can be naturally defined if needed, using $A \underset{e}{/\!/} E := A \underset{i}{/\!/} [\![E]\!]$ and an extensional set type can be defined the same way: $\{x : A \mid_e P[x]\} := \{x : A \mid_i [\![P[x]]\!]\}$.

5.7.2 Intensional Quotient Type: Axioms

Two intensional quotient types are equal when both the quotiented types are equal, and the equality relations are equal:

$$\frac{\Gamma \vdash A = A' \qquad \Gamma; x : A; y : A \vdash E[x; y] = E'[x; y] \qquad \text{“}E \text{ is an ER over } A\text{”}}{\Gamma \vdash (A/\!/E) = (A'/\!/E')}$$

(*IquotEqualIntro*)

where “E is an ER over A” is just an abbreviation for conditions that force E to be an equivalence relation over A.

Next, when two elements are equal in a quotient type, the equality predicate must be true on those elements. However, we know neither the witnesses of this predicate nor its intensional structure, therefore the equality in a quotient type

only implies the `esquash` of the equality predicate:

$$\frac{\Gamma;\, u : x = y \in (A/\!\!/E);\, v : [\![E[x;y]]\!];\, \Delta[u] \,\vdash\, C[u]}{\Gamma;\, u : x = y \in (A/\!\!/E);\, \Delta[u] \,\vdash\, C[u]} \qquad (\textit{IquotEqualElim})$$

The opposite is also true — we only need to prove the `esquash` of the equality predicate to be able to conclude that corresponding elements are equal in the quotient type:

$$\frac{\Gamma \,\vdash\, [\![E[x;y]]\!] \qquad \Gamma \,\vdash\, x \in (A/\!\!/E) \qquad \Gamma \,\vdash\, y \in (A/\!\!/E)}{\Gamma \,\vdash\, x = y \in (A/\!\!/E)} \qquad (\textit{IquotMemEqual})$$

Note that this rule has equality [17] in the quotient type in both the conclusion and the assumptions, so we still need a "base case" — an element of a base type will also be an element of any well-typed quotient of that type:

$$\frac{\Gamma \,\vdash\, x \in A \qquad \Gamma \,\vdash\, (A/\!\!/E)\,\mathrm{Type}}{\Gamma \,\vdash\, x \in (A/\!\!/E)} \qquad (\textit{IquotMemIntro})$$

Finally, we need to provide an elimination rule for quotient types. It turns out that being functional over some equivalence class of a quotient type is the same as being functional over an ND of any two elements of such class, so we can formulate the elimination rule as follows:

$$\frac{\Gamma;\, u_1 : A;\, u_2 : A;\, v : E[u_1;u_2];\, x : \mathsf{ND};\, \Delta[\mathsf{nd}_x\{u_1;u_2\}] \,\vdash\, \big[C[\mathsf{nd}_x\{u_1;u_2\}]\big]}{\Gamma;\, u : A/\!\!/E;\, \Delta[u] \,\vdash\, \big[C[u]\big]} \qquad (\textit{IquotElim})$$

5.7.3 Intensional Quotient Type: Derived Rules

From (*IquotElim*) and (*SquashEqual*) we can derive

$$\frac{\Gamma;\, u_1 : A;\, u_2 : A;\, v : E[u_1;u_2];\, x : \mathsf{ND};\, \Delta[\mathsf{nd}_x\{u_1;u_2\}] \,\vdash\, t[u_1] = t[u_2] \in C}{\Gamma;\, u : A/\!\!/E;\, \Delta[u] \,\vdash\, t[u] \in C} \qquad (\textit{IquotElim2})$$

[17] As we explained in Remark 5.1.1, in Martin-Löf type theory membership is just a particular case of the equality.

From (*IquotEqualElim*) and (*EsquashElim2*), we can derive

$$\frac{\Gamma;\ u : x = y \in (A /\!/ E);\ v : \big[E[x;y]\big];\ \Delta[u] \ \vdash\ C[u] \qquad \Gamma;\ u : x = y \in (A /\!/ E);\ \Delta[u] \ \vdash\ E[x;y]\,\text{Type}}{\Gamma;\ u : x = y \in (A /\!/ E);\ \Delta[u] \ \vdash\ C[u]} \quad (\textit{IquotEqualElim2})$$

which is equivalent to NuPRL's (*quotient-equalityElimination*) (see Appendix A.2. However, (*IquotEqualElim*) is more general than (*quotient-equalityElimination*).

Example 5.7.1. We now can prove things like $x : \mathsf{ND};\, y : \mathsf{ND} \ \vdash\ \mathsf{nd}_x\{2;4\} = \mathsf{nd}_y\{4;6\} \in \mathbb{Z}_2$ where $\mathbb{Z}_2$ is $\mathbb{Z}$ quotiented over a "$\mathtt{mod}\ 2$" equivalence relation.

5.8 Indexed Collections

5.8.1 Indexed and Predicated Collections

Consider an arbitrary type T in universe $\mathbb{U}$. We want to define the type of collections of elements of T. Such a type turned out to be very useful for various verification tasks as a natural way of representing sets of objects of a certain type (states, transitions, *etc*). We also want to formalize collections in the most general way possible, without assuming anything about T. In particular, we do not want to assume that T is enumerable or that equality on T is decidable. And in fact, the constant problems we were facing when trying to formalize collections properly were the main reason for the research that lead to this work on quotient types.

There are at least two different approaches we can take to start formalizing such collections.

1. We can start formalizing collections as pairs consisting of an index set $I : \mathbb{U}$ and an index function $f : (I \to T)$. In other words, we can start with the type $I : \mathbb{U} \times (I \to T)$.

2. We can start formalizing collections by concentrating on membership predicates of collections. In other words, we can start with the type $T \to \mathbf{Prop}$, where **Prop** is a type (universe) of propositions.

It is easy to see that these two approaches are equivalent. Indeed, if we have a pair $\langle I, f\rangle$, we can get a predicate $\lambda t.\, \exists i \in I.\, f(i) = t \in T$ and if we have a predicate P, we can take a pair $\langle \{t : T \mid P(t)\}, \lambda t.\, t\rangle$. Because of this isomorphism, everywhere below we will allow ourselves to use $T \to \mathbf{Prop}$ as a base for the collection type even though $I : \mathbb{U} \times (I \to T)$ is a little closer to our intuition about collections.

Clearly, the type $T \to \mathbf{Prop}$ is not quite what we want yet since two different predicates from that type can represent the same collection. An obvious way of addressing this problem is to use a quotient type. In other words, we want to define the type of collections as

$$\mathsf{Col}_i\{T\} \;:=\; c_1, c_2 : (T \to \mathbb{U})/\!\!/(\forall t \in T.\, c_1(t) \Leftrightarrow c_2(t)). \tag{5.1}$$

5.8.2 Collections: the Problem

Once we have defined the type of collections, the next natural step is to start defining various basic operations on that type. In particular, we want to have the following operations:

- A predicate telling us whether some element is a member of some collection:

$$\forall c \in \mathsf{Col}_i\{T\}.\, \forall t \in T.\, \mathtt{mem}(c; t) \in \mathbf{Prop}$$

- An operator that would produce a union of a family of collections:

$$\forall I \in \mathbb{U}.\, \forall C \in (I \to \mathsf{Col}_i\{T\}).\, \bigcup_{i \in I} C(i) \in \mathsf{Col}_i\{T\}$$

And we want our operators to have the following natural properties:

- $\forall c \in T \to \mathbb{U}.\, \forall t \in T.\, c(t) \Rightarrow \mathtt{mem}(c;t)$
- $\forall c \in T \to \mathbb{U}.\, \forall t \in T.\, \neg c(t) \Rightarrow \neg\mathtt{mem}(c;t)$
- $\forall c_1, c_2 \in \mathsf{Col}_i\{T\}.\, \Big(\big(\forall t : T.\, \big(\mathtt{mem}(c_1;t) \Leftrightarrow \mathtt{mem}(c_2;t)\big)\big) \Leftrightarrow c_1 = c_2 \in \mathsf{Col}_i\{T\}\Big)$ (note that the $\Leftarrow$ direction follows from the typing requirement for mem).
- $\forall I \in \mathbb{U}.\, \forall C : I \to \mathsf{Col}_i\{T\}.\, \forall t \in T.\, \big(\exists i : I.\, \mathtt{mem}(C(i);t) \Rightarrow \mathtt{mem}(\bigcup_{i \in I} C(i);t)\big)$ Note that we do not require an implication in the opposite direction since that would mean that we will have to be able to reconstruct i constructively just from some indirect knowledge that it exists. Instead we only require
- $\forall I \in \mathbb{U}, C \in I \to \mathsf{Col}_i\{T\}, t \in T. \Big(\neg\Big(\exists i : I.\, \mathtt{mem}\big(C(i);t\big)\Big) \Rightarrow \neg\Big(\mathtt{mem}\big(\bigcup_{i \in I} C(i);t\big)\Big)\Big)$

It turned out that formulating these operations with these properties is very difficult [18] in NuPRL-4 type theory with its monolithic approach to quotient types. The problems we were constantly experiencing when trying to come up with a solution included mem erroneously returning an element of $\mathbf{Prop}/\!\!/\Leftrightarrow$ instead of **Prop**, union being able to accept only arguments of type

$$C_1, C_2 : \big(I \to (T \to \mathbf{Prop})\big) /\!\!/ \big(\forall i \in I.\, \forall t \in T\, C_1(i;t) \Leftrightarrow C_2(i;t)\big)$$

(which is a subtype of the $I \to \mathsf{Col}_i\{T\}$ type that we are interested in), *etc.*

5.8.3 Collections: a Possible Solution

Now that we have $[\ ]$ and $[\![\ ]\!]$ operators, it is relatively easy to give the proper definitions. If we take $\mathtt{mem}(c;t) := [\![c(t)]\!]$ and $\bigcup_{i \in I} C(i) := \lambda t. \exists i \in I. \mathtt{mem}(C(i);t)$, we

[18]Several members of NuPRL community made numerous attempts to come up with a satisfactory formalization. The formalization presented in this chapter was the only one that worked.

can prove all the properties listed in Section 5.8.2. These proofs were successfully carried out in the MetaPRL proof development system [Hic01, HNK+, HAB+].

5.9 Related Work

Semantically speaking, in this chapter we formalize exactly the same quotient types as NuPRL does. However the formalization presented here strictly subsumes the NuPRL's one. All the NuPRL rules can be derived from the rules presented in this chapter. Consequently, in a system that supports a derived rules mechanism, any proof that uses the original NuPRL axiomatization for quotient types would still be valid under our modular axiomatization. For a more detailed comparison of the two axiomatizations see Appendix A.2.1.

In [Cou01] Pierre Courtieu attempts to add to Coq's Calculus of Constructions a notion very similar to quotient type. Instead of aiming at "general" quotient type, [Cou01] considers types that have a "normalization" function that, essentially, maps all the members of each equivalence class to a canonical member of the class. Courtieu shows how by equipping a quotient type with such normalization function, one can substantially simplify handling of such a quotient type. In a sense, `esquash` works the same way — it acts as a normalization function for the **Prop**//⇔. The main difference here is that instead of considering a normalization function that returns *an existing* element of each equivalence class, with `esquash` we utilize the open-ended nature of the type theory to equip each equivalence class with *a new* normal element.

In [Hof95a, Hof95b] Martin Hofmann have studied the intensional models for quotient types in great detail. This work is in a way complimentary to such studies — here we assume that we already have some appropriate semantical foundation

that allows quotient types and try to come up with an axiomatization that would be most useful in an automated proof assistant.

Chapter 6

Functionality and Equality in Type Theory

In Chapter 5 we saw how the axiomatization of the quotient types can be improved by making it more modular. Still, even with these enhancements the axiomatization appears to be insufficient in certain cases. In Section 6.2 we will explore some statements that are clearly true in the PER semantics (*cf.* Section 5.1.2), but are believed unprovable in the current axiomatization of the type theory.

It turns out that the limitation lies not in the axiomatization of the quotient type, but in the general treatment of equality. As we will explain in Section 6.4, it appears that the common formalization of type theory does not prohibit an existence of two types that are unequal despite having the same members and the same equality relation. In Section 6.5 we will show how to overcome this limitation by adding a natural general rule stating the uniqueness of a type with a given members and equality relation. Such a rule makes the theory substantially more complete and closer to our intuition and to the PER semantics.

6.1 Introduction to Equality Relations in Type Theory

First, we need to explain several type theory constructions and concepts.

6.1.1 Subtyping and Extensional Equality

Very informally speaking, A can be considered a subtype of B (denoted $A \subseteq B$) *iff* any function that takes elements of B as input will also work if given an element

of A as input. In other words, A can be considered a subtype of B when for any f and C, whenever $f \in B \to C$, then also $f \in A \to C$.

In PER semantics (*cf.* Section 5.1.2) this means that $A \subseteq B$ *iff* every element of A is also an element of B *and* whenever two elements are equal in A, they are also equal in B. In other words, A is a subtype of B if A's equality relation (considered as a set of pairs) is a subset of B's equality relation.

Example 6.1.1. If T is a type and P_1, P_2 are predicates on T such that $\forall t : T.P_1(t) \Rightarrow P_2(t)$, then $\{t : T \mid P_1(t)\} \subseteq \{t : T \mid P_2(t)\}$.

Example 6.1.2. $\mathbb{Z}_4 \subseteq \mathbb{Z}_2$ (where $\mathbb{Z}_n$ is $\mathbb{Z}//(\texttt{mod}\ n)$ — the type of integers quotiented over the "$\texttt{mod}\ n$" relation) since whenever two integers are equal `mod` 4 they will also be equal `mod` 2.

Remark 6.1.3. Note that in Example 6.1.2 above, the subtype actually has *more* distinct elements than the supertype. This may seem counterintuitive at first, but it does follow the informal explanation we gave at the beginning of this section. Indeed, as we explained in Example 5.1.2, if f is in $Z_2 \to C$, then it must take equal elements of Z_2 to equal elements of C. But if f respects the `mod` 2, then it would definitely respect a weaker `mod` 4 equality relation and f will be in $Z_4 \to C$.

The rules for subtype are the following: [1]

$$\frac{\Gamma;\, x : A \vdash\ x \in B \qquad \Gamma \vdash\ A\,\mathrm{Type}}{\Gamma \vdash\ A \subseteq B} \qquad (\mathit{SubtypeIntro})$$

$$\frac{\Gamma \vdash\ A \subseteq B \qquad \Gamma \vdash\ t_1 = t_2 \in A}{\Gamma \vdash\ t_1 = t_2 \in B} \qquad (\mathit{SubtypeElim})$$

Remark 6.1.4. It is pretty easy to see that $A \subseteq B$ *iff* $\lambda x.x \in (A \to B)$ (provided both A and B are well-formed types).

[1]As in previous chapter, we omit "technical" rules that are not necessary for understanding the work presented here.

We have briefly mentioned extensional equality on types in Example 5.1.5. In PER semantics, two types are considered extensionally equal when they have the same elements and equality. In other words, $A \ =_{\mathsf{e}} B$ *iff* $A \subseteq B$ *and* $B \subseteq A$.

6.1.2 Intersection Type

As we will see in Section 6.2 there is a substantial gap between the PER understanding of the extensional equality and what follows from the existing axiomatizations. In order to demonstrate this gap we will use the quotient type (as presented in Section 5.7) and the intersection type [CDC80, Pot80, Pie91].

Informally speaking, $A \bigcap B$ is the largest (with respect to the subtyping relation) type that is a subtype of both A and B. In PER semantics, $A \bigcap B$ is a type whose equality relation is the intersection of equality relations of A and B. In other words, something is a member of $A \bigcap B$ *iff* it is a member of both A and B; and two elements are equal in $A \bigcap B$ *iff* they are equal in both A and B.

Example 6.1.5. If T is a type and P_1, P_2 are predicates on T, then

$$\Big(\{t : T \mid P_1(t)\} \bigcap \{t : T \mid P_2(t)\}\Big) \ =_{\mathsf{e}} \{t : T \mid P_1(t) \wedge P_2(t)\}$$

Here the equality on common members is the same and to take the intersection of members we only need to combine the set membership predicates.

Example 6.1.6. Suppose we have some type T and two equivalence relations E_1, E_2 on T. Then according to the PER model the following must be true:

$$(T /\!/ E_1) \bigcap (T /\!/ E_2) \ =_{\mathsf{e}} T /\!/ (E_1 \wedge E_2) \tag{6.1}$$

Indeed, both types have the same members (same as in T), and similarly to Example 6.1.5 above, to take the intersection of equality relations we just need to conjoin the quotient equality predicates.

The rules for the intersection type are the following: [2]

$$\frac{\Gamma \vdash t_1 = t_2 \in A \qquad \Gamma \vdash t_1 = t_2 \in B}{\Gamma \vdash t_1 = t_2 \in (A \bigcap B)} \qquad (\mathit{IsectEq})$$

$$\frac{\Gamma;\ x : (A \bigcap B);\ \Delta[x];\ a : A;\ b : B \vdash C[a; b]}{\Gamma;\ x : (A \bigcap B);\ \Delta[x] \vdash C[a; b]} \qquad (\mathit{IsectElim})$$

6.1.3 Functionality Semantics of Sequents

One might think that the (*IsectElim*) rule above is unnecessarily complicated. To see why it can not be replaced with something as simple as, for example

$$\frac{\Gamma;\ x : A;\ x \in B;\ \Delta[x] \vdash C[x]}{\Gamma;\ x : (A \bigcap B);\ \Delta[x] \vdash C[x]} \qquad (\mathit{IsectElimWrong})$$

we have to take into account the PER semantics of sequents. The key to understanding the PER semantics of sequents is the rule

$$\frac{x : A \vdash t[x] \in B}{\vdash \lambda x.t[x] \in (A \rightarrow B)} \qquad (\mathit{FunMemIntro})$$

(which is an extremely natural rule that we would expect to be true in any "reasonable" semantics). We know (see Example 5.1.2) that in order for "$\lambda x.t[x] \in (A \rightarrow B)$" to be true, t has to respect A's equality relation. Hence in order for the sequent $x : A \vdash t[x] \in B$ to be considered true, it is insufficient for $t[x]$ to be in B for all $x \in A$ and it is necessary for t to also map equal elements of A into equal elements of B.

As shown in [All87a] the way to extend the above to the case of multiple hypotheses in a sequent is to specify that $\Gamma \vdash t \in C$ is valid when

$$\forall \vec{x}.\ \left(\begin{array}{l} \left(\vec{x} \in \Gamma[\vec{x}]\ \&\ \forall \vec{y}.\ (\vec{x} = \vec{y} \in \Gamma[\vec{x}] \Rightarrow \Gamma[\vec{x}] = \Gamma[\vec{y}])\right) \ \Rightarrow \\ \qquad\qquad \forall \vec{y}.(\vec{x} = \vec{y} \in \Gamma[\vec{x}] \Rightarrow t[\vec{x}] = t[\vec{y}] \in C[\vec{x}] = C[\vec{y}]) \end{array} \right) \qquad (6.2)$$

[2] Here we present Alexei Kopylov's formulation of the rules (as they are currently implemented in the MetaPRL system [HAB$^+$]) which is a simplification of the formulation presented in [Hic01, Section 14.22].

where Γ is $x_1 : A_1;\ x_2 : A_2[x_1];\ \ldots;\ x_n : A_n[x_1; \ldots; x_{n-1}]$, both t and C potentially have free occurrences of x_i, "$\vec{x} \in \Gamma[\vec{y}]$" is an abbreviation for "for all $i = 1..n$, $x_i \in A_i[y_1; \ldots; y_{i-1}]$", "$\vec{x} = \vec{y} \in \Gamma[\vec{z}]$" is an abbreviation for "for all $i = 1..n$, $x_i = y_i \in A_i[z_1; \ldots; z_{i-1}]$", "$\Gamma[\vec{x}] = \Gamma[\vec{y}]$" is an abbreviation for "for all $i = 1..n$, $A_i[x_1; \ldots; x_{i-1}] = A_i[y_1; \ldots; y_{i-1}]$, and "$u = v \in A = B$" is an abbreviation for "$A = B$ and $u = v \in A$"

Informally, (6.2) states that for all the x_i such that each A_i is well-formed and respects the equality relation of A_j (for all $j < i$), then t has to be a well-formed member of C and both t and C have to respect the equality relations of all the A_i.

Now we can explain what is wrong with the (*IsectElimWrong*) "rule". There, unless A is a subtype of B, $x \in B$ is not functional over $x : A$. That makes it possible for the assumption of the rule to be trivially true, while the conclusion can still easily be false.

Remark 6.1.7. The arrangement of quantifiers in (6.2) is somewhat arbitrary. There are two other meaningful ways of formulating the sequents validity. However one of them makes the natural induction rule invalid in certain cases and the other [Men88, Section 4.3] makes the ND and quotient elimination rules invalid.

6.2 The Gap

For most type constructors their axiomatization gives an exact description of what members and what equality relation a constructed type must have. For example, from (*IquotMemEqual*) and (*IquotEqualElim*) (see Section 5.7) we know that the quotient type has the equality relation specified by its equality predicate. Similarly, we know that $A \bigcap B$ type has the equality relation that is the intersection of equality relations of A and B. However, what we do not know is whether type's

equality relation uniquely (with respect to extensional equality) determines that type.

It turns out that although (6.1) of Example 6.1.6 is trivially true in PER semantics, it appears to be unprovable in our type theory. It is easy to prove that two types have the same *equality relation*, [3] but it does not seem possible to prove that those types are actually extensionally equal.

Remark 6.2.1. Note that only the $\supseteq$ direction is hard, the $\subseteq$ direction has a very straightforward proof.

As we will see in Section 6.4 it is possible to come up with a semantics (similar to that of [All87a]) that assigns each type not only a partial equality relation on terms, but some other information as well (for example, a second equality-like relation). In such semantics, most of the rules we have discussed so far will be valid, however the sequent

$$x : \big((A/\!/E_1)\bigcap(A/\!/E_2)\big) \vdash x \in \big(A/\!/(E_1 \wedge E_2)\big)$$

would not be valid there (for some A and E_i), even despite the two types having the same equality relation. Moreover, there are several variations of such non-standard semantics, with different (and often very small) sets of rules being invalid in each one.

It is still possible (although unlikely) that there exists some esoteric proof of (6.1) (which, clearly, would have to use at least one rule from each set of rules that contradict a particular non-standard semantics). However there definitely

[3] By establishing

$$\forall x, y \in T.\, x = y \in \big((T/\!/E_1)\bigcap(T/\!/E_2)\big) \Rightarrow x = y \in \big(T/\!/(E_1 \wedge E_2)\big)$$

does not exist any straightforward derivation that would correspond to the intuition of the PER approach to understanding type theory. This reveals a significant gap between the existing type theory axiomatization and the PER models. It also shows that a number of very simple statements (and more importantly — a number of statements that we would like to be able to prove in our type theory) fall into this gap. Just consider the following instance of (6.1):

Example 6.2.2. It is trivial to see that in PER semantics $\mathbb{Z}_2 \bigcap \mathbb{Z}_3 \;=_{\mathtt{e}} \mathbb{Z}_6$ (where $\mathbb{Z}_n$ is $\mathbb{Z}$ quotiented over a "$\mathtt{mod}\ n$" equivalence relation) since both sides have the same members (all natural numbers, same as $\mathbb{Z}$) and the same equality relation (since being equal both $\mathtt{mod}$ 2 and $\mathtt{mod}$ 3 is equivalent to being equal $\mathtt{mod}$ 6).

It appears that there is no way to formalize the above reasoning in the current version of type theory. And while the equality can still be proved, the proof has to use a lot of facts about the integer type, far beyond just simple arithmetical facts about `mod`s.

6.3 Functionality Structures

As we have mentioned in Remark 6.1.4, $A \subseteq B$ *iff* $\lambda x.x \in (A \to B)$. In order for some function to be a member of $A \to B$, it has to be functional over A, in oder words it has to respect *functionality structures* (such as membership and equalities) of A. Similarly, in order for A to be a subtype of B it should be the case that B contains all functionality structures that A contains.

Now let us put aside the PER notion that the only functionality structures are membership and equality and explore type theory rules from a point of view of arbitrary functionality structures. For example, for the intersection type we can prove $(A \bigcap B) \subseteq A$ and $(A \bigcap B) \subseteq B$, which means that $A \bigcap B$ can only have

functionality structures that are present in both A and B. We can also derive

$$\frac{x : C \vdash x \in A \qquad x : C \vdash x \in B}{x : C \vdash x \in (A \bigcap B)}$$

which means that $A \bigcap B$ is *the largest* type containing all functionality structures that both A and B contain.

Similarly, for the quotient constructor we can derive

$$\frac{x : A;\ y : A;\ u : E[x; y] \vdash x = y \in B}{x : (A /\!/ E) \vdash x \in B}$$

which means that $A /\!/ E$ is the *smallest* type containing all functionality structures of A and equivalence relation E.

As we will see in the next section, most of the rules are compatible with some functionality structures that go beyond just an equality relation. And in a model with such additional functionality structures it may be possible for two types to have different functionality structures (and hence not be extensionally equal) while having the same equality relation. In other words, with existing rules it might possible to have a model where a type is *not* uniquely determined by its equality relation. In particular, it might be possible to have a model where the intersection type of (6.1) more functionality structures, then the quotient and equality (6.1) no longer holds.

6.4 Non-standard Model

6.4.1 A Non-standard Functionality Structure

In this section we will sketch a model of the NuPRL type theory where types have some "functionality structure" (as described in Section 6.3) beyond just an equality relation. Namely, in this model each type is going to be associated with both an

equality relation (as in the usual PER model) and a special relation $\prec$. We will require that $\prec$ respects equality (e.g. $=\subseteq\prec$).

To provide a formal description of this model, we will follow the recursive approach of [All87a]. In [All87a, Section 4.2] Stuart Allen shows how to define recursively a two-place relation τ that assigns equality relations to terms representing types. The same way we can define a three-place τ' such that $\tau'(T,\alpha,\beta)$ would mean that T is a type with equality relation α and the $\prec$-relation β. We can make sure that $\forall T,\alpha,\beta.\,\tau'(T,\alpha,\beta)$ *implies* $(\alpha\subseteq\beta)$. Following [All87a] we will denote "$\exists\alpha,\beta.\,\tau'(T,\alpha,\beta)\,\&\,\alpha(a,b)$" as "$a=b\in T$" and "$\exists\alpha,\beta.\,\tau'(T,\alpha,\beta)\,\&\,\beta(a,b)$" as "$a\prec b\in T$".

Most of the steps defining τ' for complex types from more simple ones are the same as for τ in [All87a]. In particular, the equality in a complex type is defined based on equality in simpler ones. The $\prec$ relation in a complex type is defined in a similar manner based on $\prec$ in simpler types.

Example 6.4.1. The equality and $\prec$ for a product type are defined the usual way:

$$\langle a,b\rangle=\langle a',b'\rangle\in A\times B \;\textit{iff}\; a=a'\in A \;\textit{and}\; b=b'\in B$$
$$\langle a,b\rangle\prec\langle a',b'\rangle\in A\times B \;\textit{iff}\; a\prec a'\in A \;\textit{and}\; b\prec b'\in B$$

Similarly, for the types themselves:

$$(A\times B)=(A'\times B') \;\textit{iff}\; (A=A' \;\textit{and}\; B=B')$$
$$(A\times B)\prec(A'\times B') \;\textit{iff}\; (A\prec A' \;\textit{and}\; B\prec B')$$

There are a few exceptions, mainly where the functionality requirements or an equality type is concerned. In order for something to be functional in this model,

it has to respect both the equality and the $\prec$. In particular, for functions we define

$$f \in (A \to B) \;\; \textit{iff} \;\; \forall a, a' \in A. \left(\begin{array}{c} \big((a = a' \in A) \textit{ implies } (f(a) = f(a') \in B\big) \\ \textit{and} \\ \big((a \prec a' \in A) \textit{ implies } (f(a) \prec f(a') \in B\big) \end{array} \right)$$

while equality and $\prec$ for functions are still defined the same way as in ordinary PER models:

$$f = f' \in (A \to B) \textit{ iff } \forall a \in A. (f(a) = f'(a) \in B$$
$$f \prec f' \in (A \to B) \textit{ iff } \forall a \in A. (f(a) \prec f'(a) \in B$$

Remark 6.4.2. Still, the $\prec$ on function *types* is defined the usual way:

$$(A \to B) \prec (A' \to B') \textit{ iff } \big((A \prec A') \textit{ and } (B \prec B')\big)$$

There is no contravariance in the function domain (as it happens, for example, with subtyping relation). This follows from the fact that the $\to$ operator itself needs to be functional (as a two-place function from a type universe to itself), which means that it has to respect $\prec$ on each of its arguments.

As we mentioned in Section 6.3 $A \subseteq B$ informally means that B contains all the functionality structures of A and $A \;=_{\mathsf{e}} B$ means that A and B have identical functionality structures. In the $\prec$ model this translates into requiring $(=_A) \subseteq (=_B)$ *and* $(\prec_A) \subseteq (\prec_B)$ for $A \subseteq B$ and requiring that $(=_A) = (=_B)$ *and* $(\prec_A) = (\prec_B)$ for $A \;=_{\mathsf{e}} B$.

Similarly, $A \bigcap B$ is a type whose functionality structures are intersections of functionality structures on A and B:

$$a = b \in (A \textstyle\bigcap B) \textit{ iff } \big(a = b \in A \textit{ and } a = b \in B\big)$$
$$a \prec b \in (A \textstyle\bigcap B) \textit{ iff } \big(a \prec b \in A \textit{ and } a \prec b \in B\big)$$

From Section 6.3 we know that $A /\!/ E$ must be the smallest type such that $A \subseteq (A /\!/ E)$ which includes E as equality. In the $\prec$ model this means that $\prec_{A/\!/E}$ must be the transitive closure of $(\prec_A) \cup (=_{A/\!/E})$ and $=_{A/\!/E}$ must be E limited to members of A.

6.4.2 A Potential Counterexample

While we will continue defining our $\prec$ model of type theory in Section 6.4.3, we have already given enough details to be able to explain what happens to (6.1) in this model.

Consider some type T that has four elements — a, b, c and d that are all non-equal (in other words, T's equality relation is trivial). The only non-trivial $\prec$ relations in T are $a \prec b \in T$ and $c \prec d \in T$. Now consider E_1 that makes a and c equal and E_2 that makes b and d equal. [4]

For this T, $T /\!/ (E_1 \wedge E_2)$ will be the same as T since $E_1(x, y)$ and $E2(x, y)$ will only be simultaneously true when $x = y \in T$. However we would have $a \prec d \in (T /\!/ E_1)$ and $a \prec d \in (T /\!/ E_2)$ (since $\prec$ has to respect the equality relation), so we'll have $a \prec d \in \big((T /\!/ E_1) \bigcap (T /\!/ E_2)\big)$, therefore the two types $T /\!/ (E_1 \wedge E_2)$ and $(T /\!/ E_1) \bigcap (T /\!/ E_2)$ of (6.1) will have different $\prec$ relations and will be unequal.

6.4.3 Completing the Model

There seems to be many alternative ways of formulating the remaining definitions in the model. We will briefly describe one of them.

[4] It is easy to make sure such E_1 and E_2 are well-formed and functional over T — for example by enumerating all the possible equalities: $E_1[x; y] \ := \ (x = a \in T \wedge y = a \in T) \vee (x = a \in T \wedge y = c \in T) \vee \ldots$.

The functionality requirement for sequents may be defined in the following way:

$$\forall \vec{x}. \left(\begin{array}{c} \vec{x} \in \Gamma[\vec{x}] \ \& \ \forall \vec{y}. \left(\begin{array}{l} (\vec{x} = \vec{y} \in \Gamma[\vec{x}] \Rightarrow \Gamma[\vec{x}] = \Gamma[\vec{y}]) \ \& \\ (\vec{x} \prec \vec{y} \in \Gamma[\vec{x}] \Rightarrow \Gamma[\vec{x}] \prec \Gamma[\vec{y}]) \end{array} \right) \\ implies \\ \forall \vec{y}. \left(\begin{array}{l} (\vec{x} = \vec{y} \in \Gamma[\vec{x}] \Rightarrow t[\vec{x}] = t[\vec{y}] \in C[\vec{x}] = C[\vec{y}]) \ \& \\ (\vec{x} \prec \vec{y} \in \Gamma[\vec{x}] \Rightarrow t[\vec{x}] \prec t[\vec{y}] \in C[\vec{x}] \prec C[\vec{y}]) \end{array} \right) \end{array} \right)$$

Note that in general we do not require two types in a $\prec$ relation to be in any way "compatible" with each other. We even allow situations when $A \prec B$ is true despite one type being empty while the other is not. This requires us to be very careful when defining the meaning of sequents in this model. This is also the reason the definition is asymmetric.

The $\prec$ on equality types would also be somewhat asymmetric:

$$(a = b \in T) \prec (a' = b' \in T') \ \mathit{iff} \ \big((a \prec a' \in T) \ \mathit{and} \ (b \prec b' \in T) \ \mathit{and} \ (T \prec T')\big)$$

The same asymmetry would exist for the dependent product too:

$$(\langle a, b\rangle \prec \langle a', b'\rangle \in (x : A \times B[x]) \ \mathit{iff} \ a \prec a' \in A \ \mathit{and} \ b \prec b' \in B[a]$$

6.4.4 Other Interesting Non-standard Models

One can try to create a non-standard model by adding an extra equivalence relation to each type (in addition to the usual equality relation, similar to how we have added an extra relation in our model above). If we add an extra equivalence relation F such that $F \subseteq E$ (where E is the usual equality relation), then we would get a reasonable model, but it would not be a counterexample — (6.1) will still be true in such a model.

We could also try to add an extra equivalence relation $\approx$ such that $E \subseteq\approx$. In order for the equality predicate to be well-formed in such model, we'll have to define $(a = b \in T) \approx (a' = b' \in T)$ as $(a \approx' a \in T) \wedge (b \approx b' \in T)$. Since $\approx$ does not imply equality, we can have $(a = b \in T) \approx (a' = b' \in T)$ and $a = b \in T$ without $a' = b' \in T$. This means that we might have $A \approx B$ where A and B have different membership relations, even have one empty while other is not. It is unclear whether there exists a reasonable definition of $(a = b \in T) \approx (a' = b' \in T')$ that would make this a valid model (just consider a case when a is not a member of T' and a' is not a member of T).

6.5 Uniquely Defining a Type by its Equality Relation

In the previous section we have demonstrated that the type theory can have some pretty unexpected models. We can eliminate these undesired models by adding a new rule stating that a type is uniquely determined by its equality relation. There are many alternative ways of formulating such a rule. One of them is the following:

$$\frac{\begin{array}{l}\Gamma \vdash X \subseteq A \subseteq (X \mathbin{/\!/_{i}} True) \\ \Gamma \vdash X \subseteq B \subseteq (X \mathbin{/\!/_{i}} True) \\ \Gamma; x_1 : X; x_2 : X \vdash (x_1 = x_2 \in A) \Leftrightarrow (x_1 = x_2 \in B)\end{array}}{\Gamma \vdash A =_{\mathsf{e}} B} \quad (EqEq)$$

The $(EqEq)$ essentially says that whenever two subtypes A and B of a type X have the same members as X and the same equality relations, then the two types must be equal. This rule can be used to prove that $(A \mathbin{/\!/_{i}} E_1) \bigcap (A \mathbin{/\!/_{i}} E_2)$ and $A \mathbin{/\!/_{i}} (E_1 \wedge E_2)$ are always extensionally equal (provided both are types).

The $(EqEq)$ is a relatively straightforward formalization of our intuition, but at the expense of being unnecessarily complex. An equivalent simpler formulation

is

$$\frac{\Gamma \vdash A \subseteq B \qquad \Gamma \vdash B \subseteq (A \mathbin{/\!\!/_{i}} True)}{\Gamma \vdash B \subseteq \big(u, v : A \mathbin{/\!\!/_{i}} (u = v \in B)\big)} \qquad (Antiquotient)$$

This rule essentially says that if A and B have the same members, with A having a more refined equality, then once A's equality is adjusted to match B's, then the two types will be equal. And since the existing rules for quotient already state that the quotient must be "small", we only need to state that the quotient is sufficiently "big" in the conclusion of this rule.

It is possible to simplify it further to a stronger more general rule: [5]

$$\frac{\Gamma;\, x_1 : X;\, x_2 : X \vdash (x_1 = x_2 \in A) \Rightarrow (x_1 = x_2 \in B)}{\Gamma;\, x : A;\, [\![x \in X]\!] \vdash x \in B} \qquad (EqMemEq)$$

This rule mentions neither quotients nor intersection, it only talks about equality. However from this rule we can easily derive (6.1). This shows that (*EqMemEq*) indeed captures a fundamental property of PER approach to the type theory.

[5] A large part of this simplification step was discovered by Alexei Kopylov.

Chapter 7

Propositional Markov's Principle for Type Theory

7.1 Introduction

In this chapter we will show that the `squash` operator we have presented in Section 5.2 provides a way of formulating constructive recursive mathematics in a type theory. This allows creating a theory where many significant aspects of classical and constructive reasoning can coexist.

The `squash` operator can also be considered a modality. The propositional logic equipped with this modality can express a principle that allows turning classical proofs of squash-stable propositions into constructive ones. This principle is valid in the standard type theory semantics if we consider it in the classical meta-theory. Therefore this principle does not destroy the computational nature of type theory in a sense that we can always extract a witness term from a derivation.

It turns out that the principle we introduce implies Markov's principle providing us a propositional analog of Markov's principle. It is rather surprising that such analog exists because normally one needs quantifiers in order to formulate Markov's principle.

We also show an equivalent way of defining the same principle using a membership type instead of the `squash` operator.

The main goal of this work is to get a better understanding of the type theory. In practice, the NuPRL group continues to use purely intuitionistic reasoning for most purposes. In the MetaPRL system the Markov's principle is implemented in

a separate module which leaves the choice between the purely intuitionistic theory and the extended theory to individual users.

Parts of this chapter is a joint work with Alexei Kopylov — see also [KN01].

7.1.1 Markov's Constructivism

Among the many existing approaches to constructivism (see [Bee85, BR88, TvD88] for an overview) we are especially interested in the constructive recursive mathematics (CRM) approach developed by Markov [Mar54, Mar62] and in constructive type theories (especially those that are based on Martin-Löf type theory [ML82]). In this chapter we demonstrate how to apply the ideas of CRM to a constructive type theory thus creating an interesting hybrid type theory that combines many strengths of both approaches to constructive mathematics.

According to Markov's CRM approach, constructive objects are algorithms, where algorithms are understood as finite strings in a finite alphabet. All logical connectives are understood computationally. That is, a statement is true if and only if there exists an algorithm that produces a *witness* of this statement. For example, a witness for $\forall x.A(x) \vee \neg A(x)$ is an algorithm that for a given x tells us either that $A(x)$ is true (and provides a witness for $A(x)$) or that $\neg A(x)$ is true (and provides a witness for $\neg A(x)$). That means that $\forall x.A(x) \vee \neg A(x)$ is true only for decidable predicates A. Since not all predicates are decidable, Markov's school has to reject the rule of excluded middle.

Remark 7.1.1. Note that a witness of a proposition does not necessarily "prove" that proposition. For example, $\forall x.A(x) \vee \neg A(x)$ is *true* when there is a decision algorithm for A, but it does not necessarily mean that there exists a *proof* [1] that

[1] In this chapter we use terms *proof* and *derivation* interchangeably.

this algorithm works properly (i.e. always terminates and gives the correct answer).

In this respect the constructive recursive mathematics differs from the Brouwer-Heyting-Kolmogorov's intuitionism. There a statement is considered to be true when there exists a proof of it. The disjunction $A \vee B$ is true when there is a proof of A or there is a proof of B. Since according to Gödel, there is a statement that can be neither proven nor falsified, we have that $A \vee \neg A$ is not always true. Therefore intuitionists also reject the law of excluded middle, but for a different reason.

The question arisen in the CRM is *which means is one allowed to use in order to establish that a particular algorithm is indeed a witness for the given proposition?* This is not an obvious question since the termination problem is undecidable. Even if algorithm terminates for every input, we can not test it explicitly, because there are infinitely many possible inputs. But to establish that an algorithm is applicable to an object a, the algorithm does not have to be executed explicitly from the beginning to the end. According to Markov [Mar62] we can prove this by contradiction. That is, we are allowed use some classical reasoning to prove that a particular algorithm has some particular properties.

Markov school of constructive mathematics accepts the following principle: "*If it is not a case that a program does not terminate, then it does terminate.*" [Mar54, Mar62]. The following informal reasoning justifies this principle. If we have a proof that a program terminates using classical reasoning, then to obtain a constructive proof of the termination, we should run the program until it stops. We know that it must eventually stop (since we believe in classical logic). And when it stops we will have a *constructive* proof of its termination.

7.1.2 Markov's Principle

The Markov school uses the intuitionistic predicate arithmetic with an additional principle (known as Markov's principle):

$$\forall x.(A(x) \vee \neg A(x)) \rightarrow \neg\neg\exists x.A(x) \rightarrow \exists x.A(x) \tag{7.1}$$

where variables range over natural numbers. Note that this principle does not hold for Brouwer-Heyting-Kolmogorov's intuitionism.

Here is the justification of this principle in the CRM framework. Assume $\forall x.(A(x) \vee \neg A(x))$. Then there exists an effective procedure which for every x decides whether $A(x)$ or $\neg A(x)$. To establish (7.1), we need to write a program which would produce a witness of $\exists x.A(x)$. We can achieve that by writing a program which will try every natural number x and check $A(x)$ until it finds such an x that $A(x)$ is true. From $\neg\neg\exists x.A(x)$ we know that it can not be the case that such x would not be found. Therefore it is impossible that this algorithm does not terminate. Hence according to recursive constructivism it eventually stops.

Markov's principle is an important technical tool for proving termination of computations. Adding Markov's principle to a traditional constructive type theory could considerably extend the power of the latter in a pivotal class of verification problems.

7.1.3 Type Theory

We assume the type theory under consideration adheres to the *propositions-as-types principle*. As we have explained in Section 5.1.1 this principle means that a proposition is identified with the type of all its witnesses. A proposition is considered true if the corresponding type is inhabited and is considered false otherwise.

This makes terms *an element of a type* and *a witness of a proposition* synonyms. The elements of a type are actually λ–terms, i.e. programs that evaluates to a "canonical" element of this type.

We also assume that the type theory is extensional. That is, to prove that a term f is a function from A to B, it should be sufficient to show that for any $a \in A$ the application fa eventually evaluates to an element of the type B. This allows us to deal with recursive functions that we can prove will always terminate. We will use the `fix` operator to define recursive functions, where $\mathtt{fix}(f.p[f])$ is defined as $(\lambda x.p[xx])(\lambda x.p[xx])$, i.e. `fix` is the operator with the following property $\mathtt{fix}(f.p[f]) \mapsto p[\mathtt{fix}(f.p[f])]$. Although the general typing rule for `fix`

$$\frac{f : A \to A \vdash p[f] \in A \to A}{\vdash \mathtt{fix} f.p[f] \in A \to A}$$

is unsound, but for some particular p we can prove that $\mathtt{fix}(f.p[x])$ is a well-typed function.

Note that in an extensional type theory *a witness of a proposition* T is not the same as *a derivation of a proposition* T. In general, a witness (i.e. an element) of the type T may potentially range from full encoding of some derivation of T to a trivial constant.

Example 7.1.2. If V is an empty type, then *every* function has type $V \to W$, e.g. a function $\lambda x.foo$ is an element of $(A \wedge \neg A) \to \bot$ (although $\lambda x.foo$ does not encode any derivations of proposition $(A \wedge \neg A) \to \bot$).

Remark 7.1.3. Note that the question of whether a particular term is a witness of a particular proposition is in general undecidable.

We assume that the type theory has a membership type — "$t \in T$" which stands for the proposition "t is an element of type T". The only witness of a

membership proposition is a special constant $\bullet$.[2]

We assume that $t \in A$ implies A. The inverse should also be true:

Property 7.1.4. If we can prove $\Gamma \vdash A$ then there is a term t such that $\Gamma \vdash t \in A$.

Remark 7.1.5. The reason we want judgments of the form $\Gamma \vdash T$ and not just $\Gamma \vdash t \in T$ is that we are interested in type theories that can be used as a foundation for theorem provers. In a theorem prover situation we want user to be able to state and prove a judgment of the for $\Gamma \vdash T$ and have the system "extract" t from the resulting derivation instead of being required to figure out and provide t upfront.

In this chapter we present several formal derivations. These derivations were machine-checked in the MetaPRL system [HNK$^+$, Hic01]. The rules used in those derivations are summarized in Section A.1. The results of Sections 7.2, 7.3 and 7.4 are valid for any type theory containing this set of rules and satisfying Property 7.1.4. The results of the sections Sections 7.5 and 7.6 also require an intentional semantics. NuPRL is an example of a type theory that satisfies all these constraints.

However, most of our ideas can be easily applied to an even wider class of type theories. For example, typing rules are not really essential. Additionally, we do not need a membership type to express Markov's principle, we can use the `squash` operator instead (Section 7.4). Markov's principle can be used even in a purely propositional fragment of type theory without arithmetic and quantifiers.

---

[2] MetaPRL system [Hic01, HNK$^+$] uses the unit element () or "it" as a $\bullet$, NuPRL uses `Ax` and [Tho91] uses *Triv*.

7.2 Constructive Recursive Mathematics in a Type Theory with a Membership Type

Suppose one has proved that A implies $t \in T$ and $\neg A$ also implies *the same* $t \in T$. Then *classically* we can conclude that T is inhabited. Moreover the philosophy of recursive constructivism allows us to conclude that T is true *constructively*, because we can explicitly provide a constructive object t as an element of T. In other words, since the witness of T (which is just t) does not depend on the proof of $t \in T$, then T has a uniform witness regardless whether A is true or not (although the *proof* that t is a witness of T may depend on A).

This argument establishes that the following type theory rule is valid according to recursive constructivism:

$$\frac{\Gamma;\, x : A \,\vdash\, t \in T \quad \Gamma;\, y : \neg A \,\vdash\, t \in T \quad \Gamma \,\vdash\, A\,\mathrm{Type}}{\Gamma \,\vdash\, t \in T} \tag{7.2}$$

This rule formalizes exactly the philosophy of the recursive constructivism. In fact, as we are going to prove in Theorem 7.4.2, the above rule implies Markov's principle.

Remark 7.2.1. Note that in NuPRL-like type theories $t \in T$ is well-formed only when t is in fact an element of T. Therefore the rule stating that $\neg\neg(t \in T)$ implies $t \in T$ would be useless. On the other hand, in NuPRL type theory (7.2) is equivalent to

$$\frac{\Gamma \,\vdash\, (s \in T) \quad \Gamma \,\vdash\, (t \in T) \quad \Gamma \,\vdash\, \neg\neg(s = t \in T)}{\Gamma \,\vdash\, s = t \in T} \tag{7.3}$$

but the proof of (7.3) $\Rightarrow$ (7.2) is very NuPRL-specific.

7.3 Squashed Types and Squash-Stability

Now assume that our type theory contains a "`squash`" operator described in Section 5.2. In other words, for each type A we have a type $\lceil A \rceil$ ("squashed A") which is empty *if and only if* A is empty and contains a single element $\bullet$ when A is inhabited. Informally one can think of $\lceil A \rceil$ as a proposition that says that A is a *non-empty type*.

The (*SquashIntro*) rule of Section 5.2.2 allows us to prove that $A \vdash \lceil A \rceil$. However since $\lceil A \rceil$ does not provide a witness for A, $\lceil A \rceil$ does not always imply A; we can only derive $\lceil A \rceil \vdash \neg\neg A$. While $\lceil A \rceil$ does not provide a witness for A in general, in some cases we know what that witness would be when A is non-empty. For example, we know that if $t \in T$ is true, then $\bullet$ is the witness for the type $t \in T$. We will call such types *squash-stable*.

Definition 7.3.1. *A type T is squash-stable (in context Γ) when $\Gamma;\, x : T \vdash t \in T$ is provable for some t that does not have free occurrences of x.*

The above definition is equivalent to the one given in [Jac95, Section 3.7.3] using the `squash` operator:

Lemma 7.3.2. *T is squash-stable in a context Γ iff $\Gamma;\, v : \lceil T \rceil \vdash T$ is derivable.*

Proof. Suppose T is squash-stable. Then we have the following derivation of $\Gamma;\, v : \lceil T \rceil \vdash T$:

$$\cfrac{\cfrac{\Gamma;\, x : T \vdash t \in T}{\Gamma;\, v : \lceil T \rceil \vdash t \in T}(SquashElim)}{\Gamma;\, v : \lceil T \rceil \vdash T}$$

Now assume that $\Gamma;\, v : \lceil T \rceil \vdash T$. Then for some term t we have $\Gamma;\, v : \lceil T \rceil \vdash t \in T$. Note that term t may depend on v, *i.e.* $t = t[v]$. We can derive the

following:

$$
\dfrac{\dfrac{\overline{\Gamma;\, v : T \vdash T}}{\Gamma;\, v : T \vdash \bullet \in \lceil T \rceil}(\mathit{SquashMemIntro}) \qquad \Gamma;\, v : \lceil T \rceil \vdash t[v] \in T}{\Gamma;\, v : T \vdash t[\bullet] \in T}(\mathit{Let})
$$

Therefore T is squash-stable. □

We have already seen that $t \in T$ is a squash-stable type. The `squash` type itself is also squash-stable since $\bullet \in \lceil A \rceil$ whenever $\lceil A \rceil$ is true. Other examples of squash-stable types include the empty type ($\bot$), negations of arbitrary types ($\neg A$) as well as conjunctions of squash-stable types.

Remark 7.3.3. Note however that a disjunction of two squash-stable types is not necessarily squash-stable. For example, $\neg A \vee \neg\neg A$ would not be squash-stable unless A is decidable. Indeed, if A is undecidable, then there is no way to figure out which of the disjuncts is true even when we know that one of them must be true.

7.4 Classical Reasoning on Squashed Types

Squash operator gives us an alternative way of formulating constructive recursive mathematics in a type theory. Let us consider a problem similar to the one we have considered in Section 7.2.

Suppose we have constructively proved that $A \to B$ and $(\neg A) \to B$. It means that there is an algorithm that produces an element of B when A is true, and another algorithm that produces an element of B when A is false. Classically we know that B is true, because in each case we can produce an element of B.

But when A is undecidable, we might not have a uniform algorithm for producing an element of B, so B will not necessary be constructively true. In intuitionistic setting only $\neg\neg B$ would be provable in this case.

However if B is squash-stable, then there exists an element b, such that $b \in B$ whenever B is non-empty. In this case we know that B is not an empty type regardless of whether A is true. The constant algorithm that returns b does not depend on the truth of A. Therefore in constructive recursive mathematics we can conclude that B is *constructively* true, since we have an element b such that $b \in B$.

This reasoning establishes the following rule:

$$\frac{\Gamma; x : A \vdash B \quad \Gamma; y : \neg A \vdash B \quad \Gamma; v : \left[B\right] \vdash B \quad \Gamma \vdash A\,\mathrm{Type}}{\Gamma \vdash B} \tag{7.4}$$

This rule allows us to turn classical proofs of *squash-stable* statements into constructive ones. It is clear that rule (7.2) from Section 7.2 is a particular instance of (7.4). We will show that these two rules are in fact equivalent. We can also write a simpler version of the same rule:

$$\frac{\Gamma \vdash \neg\neg A \quad \Gamma \vdash A\,\mathrm{Type}}{\Gamma \vdash \left[A\right]} \tag{7.5}$$

This rules states that $\left[A\right] \Leftrightarrow \neg\neg A$ or informally, A is *a non-empty type* if and only if it is *not an empty type.*

Another way to formulate the same principle is to allow classical reasoning inside `squash` operator:

$$\frac{\Gamma \vdash A\,\mathrm{Type}}{\Gamma \vdash \left[A \vee \neg A\right]} \tag{7.6}$$

The following two theorems state that all the above rules are equivalent and that they imply Markov's principle. This shows that we can formulate Markov's

principle in a very simple language — we only need propositional language with the modal operator “`squash`”.

Theorem 7.4.1. *The rules* (7.2), (7.4), (7.5) *and* (7.6) *are equivalent.*

Proof. (7.4) $\Rightarrow$ (7.2). Take $B = (t \in T)$. We know that $t \in T$ is squash stable, therefore we can apply rule (7.4) to derive (7.2).

(7.2) $\Rightarrow$ (7.5).

$$
\dfrac{
\dfrac{\dfrac{}{\Gamma;\, x : A \vdash A}}{\Gamma;\, x : A \vdash \bullet \in [A]}
\quad
\dfrac{\dfrac{\Gamma \vdash \neg\neg A \quad \dfrac{}{y : \neg A;\, z : \neg\neg A \vdash \bot}}{\Gamma;\, y : \neg A \vdash \bot}(Cut)}{\Gamma;\, y : \neg A \vdash \bullet \in [A]}(E_{\bot}), (Cut)
\quad
\Gamma \vdash A\,\text{Type}
}{
\dfrac{\Gamma \vdash \bullet \in [A]}{\Gamma \vdash [A]}(E_{\in})
}(7.2)
$$

(7.5) $\Rightarrow$ (7.6).

It is easy to establish that $\neg\neg(A \vee \neg A)$ is an intuitionistic tautology. Therefore we have the following derivation:

$$
\dfrac{\dfrac{\Gamma \vdash A\,\text{Type}}{\Gamma \vdash \neg\neg(A \vee \neg A)} \quad \Gamma \vdash A\,\text{Type}}{\Gamma \vdash [A \vee \neg A]}(7.5)
$$

(7.6) ⇒ (7.4).

$$\dfrac{\dfrac{\Gamma \vdash A\,\text{Type}}{\Gamma \vdash \left[A \vee \neg A\right]}(7.6) \quad \dfrac{\dfrac{\dfrac{\Gamma;\, x : A \vdash B \quad \Gamma;\, y : \neg A \vdash B}{\Gamma;\, z : A \vee \neg A \vdash B}(E_\vee)}{\Gamma;\, z : A \vee \neg A \vdash \left[B\right]}(\mathit{SquashIntro})}{\Gamma;\, v : \left[A \vee \neg A\right] \vdash \left[B\right]}(\mathit{Unsquash}) \quad \Gamma;\, \left[B\right] \vdash B}{\Gamma \vdash B}(\mathit{Cut})$$

□

Theorem 7.4.2. *The rule* (7.5) *implies Markov's principle in a type theory:*

$$\forall x : \mathbb{N}.(A(x) \vee \neg A(x)) \rightarrow \neg\neg\exists x : \mathbb{N}.A(x) \rightarrow \exists x : \mathbb{N}.A(x)$$

Proof. We need to show that the following sequent is derivable:

$$d : \forall x : \mathbb{N}.(A(x) \vee \neg A(x));\, v : \neg\neg\exists x : \mathbb{N}.A(x) \vdash \exists x : \mathbb{N}.A(x)$$

The proof is a straightforward formalization of Markov's reasoning [Mar62]. We are given the element d of the type $\forall x : \mathbb{N}.(A(x) \vee \neg A(x))$. Hence d is an algorithm which decides whether $A(x)$ holds when given a natural number x. We need to construct a function f_d which would find an x such that $A(x)$. Let f_d be the function

$$\mathtt{fix}\Big(f.\lambda x.\mathit{decide}\big(d(x); a.\left\langle x, a\right\rangle; b.f(x+1)\big)\Big)$$

that is a function such that

$$f_d(x) = \begin{cases} \left\langle x, a\right\rangle, & \text{if } A(x) \text{ is true and } a \in A(x) \\ f_d(x+1), & \text{if } A(x) \text{ is false} \end{cases}$$

Whenever $A(n)$ holds for some n, that n becomes an upper bound for computing any $f(n-k)$ (for any $0 \leq k \leq n$). That is, one can prove that $\forall k \leq n. f_d(n-k) \in$

$\exists x : \mathbb{N}.A(x)$ by induction on k. Therefore $f_d(0) \in \exists x : \mathbb{N}.A(x)$. Then we have the following derivation:

$$\cfrac{\cfrac{d : \forall x : \mathbb{N}.(A(x) \vee \neg A(x));\ n : \mathbb{N};\ u : A(n) \ \vdash\ f_d(0) \in \exists x : \mathbb{N}.A(x)}{d : \forall x : \mathbb{N}.(A(x) \vee \neg A(x));\ \exists x : \mathbb{N}.A(x) \ \vdash\ f_d(0) \in \exists x : \mathbb{N}.A(x)}(E_\exists)}{d : \forall x : \mathbb{N}.(A(x) \vee \neg A(x));\ \left[\exists x : \mathbb{N}.A(x)\right] \ \vdash\ f_d(0) \in \exists x : \mathbb{N}.A(x)}(\mathit{SquashElim2})$$

Now we are left to show that $\neg\neg\exists x : \mathbb{N}.A(x)$ implies $\left[\exists x : \mathbb{N}.A(x)\right]$. This is true because of the rule (7.5). □

7.5 Semantical Consistency of Markov's Principle

Theorem 7.5.1. *The rule* (7.6) *(as well as its equivalents —* (7.2)*,* (7.4) *and* (7.5)*) is valid in S. Allen's semantics [All87a, All87b] if we consider it in a classical meta-theory.*

Proof. We need to show that in every model $\Gamma \vdash \left[A \vee \neg A\right]$ is true when A is a type. It is clear that whenever A is a well-formed type, $\left[A \vee \neg A\right]$ will also be a well-formed type. To prove that it is a true proposition we have to find a term in this type. Let us prove that in every model $\bullet$ is the witness of $\left[A \vee \neg A\right]$. Since we are in a classical meta-theory, in every model (where A is a type) and for every instantiation of variables introduced by Γ, A is either empty or not. If A is non-empty, then $A \vee \neg A$ is non-empty and so $\bullet \in \left[A \vee \neg A\right]$. If A is an empty type, then $\neg A$ is non-empty type and so, $\bullet$ is again in $\left[A \vee \neg A\right]$. Therefore $\bullet \in \left[A \vee \neg A\right]$ always holds. □

Remark 7.5.2. Note that even using a classical meta-theory we would not be able to establish validity of $\Gamma \vdash A \vee \neg A$. Indeed, as we saw in Remark 7.3.3 disjunctions are not necessarily squash-stable and there is no uniform witness for $A \vee \neg A$.

Corollary 7.5.3. *The rule* (7.6) *(and its equivalents) is consistent with the* NuPRL *type theory containing the theory of partial functions [CC01].*

On the other hand, the rule of excluded middle $\Gamma \vdash A \vee \neg A$ is known to be inconsistent with the theory of [CC01]. In particular, in that theory we can prove that there exists an undecidable proposition. That is, for some P the following is provable:

$$\neg(\forall n : \mathbb{N}.P(n) \vee \neg P(n)) \tag{7.7}$$

Therefore even using rule (7.6) we can not prove that

$$\big[\forall n : \mathbb{N}.P(n) \vee \neg P(n)\big]$$

(which would contradict (7.7)). But we can prove a weaker statement

$$\forall n : \mathbb{N}.\big[P(n) \vee \neg P(n)\big]$$

which does not contradict (7.7).

7.6 Squash Operator as a Modality

The `squash` operator can be regarded as an intuitionistic modality. It turns out that it behaves like the lax modality (denoted by $\bigcirc$) in the Propositional Lax Logic (PLL) which was developed for several different purposes independently (see [FM97] for an overview).

PLL is an extension of intuitionistic logic with the following rules (in Gentzen style):

$$\frac{\Gamma \vdash A}{\Gamma \vdash \big[A\big]} \qquad \frac{\Gamma;\, A \vdash \big[B\big]}{\Gamma;\, \big[A\big] \vdash \big[B\big]}$$

PLL$^+$ is PLL+$(\neg\big[\bot\big])$, i.e. PLL$^+$ has an additional rule:

$$\frac{\Gamma;\, A \vdash \bot}{\Gamma;\, \big[A\big] \vdash \bot}$$

PLL* is PLL$^+$+($\lceil A \rceil \leftrightarrow \neg\neg A$). We can write this axiom as the rule in Gentzen style:

$$\frac{\Gamma; \neg A \vdash \bot}{\Gamma \vdash \lceil A \rceil}$$

PLL$^+$ and PLL* are decidable and have natural categorical and Kripke models [FM97, AMdPR01]. They meet cut elimination property. PLL$^+$ has the subformula property. PLL* also has the subformula property if we define $\neg A$ to be a subformula of $\lceil A \rceil$.

Theorem 7.6.1. *Let A be a propositional formula with the* `squash` *modality. Let Γ be a set of hypothesis of the form $x : (p\ Type)$ for all propositional variables p in A. Then*

(i) PLL$^+$ $\vdash A$ iff $\Gamma \vdash A$ is derivable in the type theory without 7.6

(ii) PLL* $\vdash A$ iff $\Gamma \vdash A$ is derivable in the type theory with 7.6

Proof. From left to right this theorem can be proved by induction on derivation in PLL$^+$ (PLL*). The right to left direction needs a semantical reasoning. We will only outline the proof for PLL*.

Let A' be the formula A where all subformulas of the form $\lceil B \rceil$ are replaced by $\neg\neg B$. If $\Gamma \vdash A$ is derivable in the type theory with 7.6 then this sequent is valid in the standard semantics in classical meta-theory (Theorem 7.5.1). Since $\lceil B \rceil \leftrightarrow \neg\neg B$ is true in this semantics then $\Gamma \vdash A'$ is also true. A' is a modal-free formula. Therefore A' is a valid intuitionistic formula. Hence A' is derivable in the intuitionistic propositional logic. Since we have $\lceil B \rceil \leftrightarrow \neg\neg B$ in PLL*, we can derive A in PLL*. □

Remark 7.6.2. It is possible to consider the lax modality in PLL$^+$ as the diamond modality in the natural intuitionistic analog of S4 (in the style of [Wij90]) with an

additional rule $\Box A \leftrightarrow A$. Note that since in intuitionistic logics $\Box$ and $\Diamond$ are not interdefinable, $\Box A \leftrightarrow A$ does not imply $\Diamond A \leftrightarrow A$.

Example 7.6.3. We can prove some basic properties of `squash` in PLL$^+$:

$\big[A\big] \to \neg\neg A$

$\big[A\big] \leftrightarrow \big[\big[A\big]\big]$

$\big[A \wedge B\big] \leftrightarrow (\big[A\big] \wedge \big[B\big])$

$\big[A \to B\big] \to (\big[A\big] \to \big[B\big])$, but $(\big[A\big] \to \big[B\big]) \to \big[A \to B\big]$ is true only in PLL*

$\big[\neg A\big] \leftrightarrow \neg\big[A\big]$

$(\big[A\big] \vee \big[B\big]) \to \big[A \vee B\big]$, however $\big[A \vee B\big] \not\to \big[A\big] \vee \big[B\big]$ even in PLL*

We can express the notion of squash-stability in this logic as $sqst(A) = \big[A\big] \to A$.

Example 7.6.4. The following properties of squash-stability are derivable in PLL$^+$:

$sqst(\bot)$

$sqst(\neg A)$

$sqst(\big[A\big])$

$sqst(A) \wedge sqst(B) \to sqst(A \wedge B)$, but $sqst(A) \vee sqst(B) \not\to sqst(A \vee B)$

$sqst(B) \to sqst(A \to B)$

In PLL* we can also prove

$sqst(A) \to (\neg\neg A \to A)$

7.7 Related Work

The notion of *squash-stability* we use is very similar to the squash-stability defined in [Hic01, Section 14.2] and to the notion of *computational redundancy* [BCMS89, Section 3.4].

The `squash` operator we use is similar to the notion of proof irrelevance [Hof95a, Pfe01a]. Each object in a proof irrelevance type is considered to be equal to any other object of this type. In [Pfe01a] proof irrelevance was expressed in terms of a certain modality $\triangle$. If A is a type then $\triangle A$ is a type containing all elements of A considered equal. Using NuPRL notation we can write $\triangle A = A//True$, where $A//P$ is a quotient of a type A over relation P. We can prove the following chain:

$$A \to \triangle A \to \big[A\big] \to \neg\neg A$$

The main difference between $\big[A\big]$ and $\triangle A$ is that there is no uniform element for $\triangle A$. Therefore $\triangle A$ is not squash-stable and $\big[A\big]$ does not imply $\triangle A$. However it seems that modal logic of $\triangle$ modality is the same as logic of `squash` (i.e. PLL^+).

In [Jac95, Section 3.7] Paul Jackson discusses the issue of classical reasoning in the NuPRL type theory. He uses the $\big[\forall P \in \mathbf{Prop}.\,(P \vee \neg P)\big]$ axiom which is much stronger than our $\forall P \in \mathbf{Prop}.\,\big[P \vee \neg P\big]$ one and is inconsistent with Allen's semantics and with the theory of [CC01].

As far as we know Markov's principle in type theory was considered only by Erik Palmgren in [Pal95]. He proved that a fragment of *intensional* Martin-Löf type theory is closed under Markov's rule:

$$\frac{\Gamma \vdash \neg\neg\exists x : A.P[x]}{\Gamma \vdash \exists x : A.P[x]}$$

where $P[x]$ is an equality type (i.e. $P[x]$ is $t[x] = s[x] \in T$. It is easy to see that this formulation of Markov's rule is not valid for type theories with undecidable equality and, in particular, in extensional type theories.

Appendix A

A.1 Some Type Theory Rules

The judgments of the type theory are the sequents of the following form

$$x_1 : A_1; x_2 : A_2[x_1]; \ldots; x_n : A_n[x_1, \ldots, x_{n-1}] \vdash C[x_1, \ldots, x_n]$$

This sequent is true if we have a uniform witness $t[x_1, \ldots, x_n]$ such that for every $x_1, \ldots, x_n$ if $x_i \in A_i[x_1, \ldots, x_{i-1}]$ then $t[x_1, \ldots, x_n]$ is a member of $C[x_1, \ldots, x_n]$.

The inference rules are presented below[1]. For every type constructor we have a well-formedness rule (W), an introduction rule (I), an elimination rule (E) and a membership introduction rule (M).

A.1.1 Structural Rules

$$\frac{}{\Gamma; x : A; \Delta[x] \vdash A} \quad (ax)$$

$$\frac{\Gamma; \Delta \vdash A \qquad \Gamma; x : A; \Delta \vdash C}{\Gamma; \Delta \vdash C} \quad (\mathit{Cut})$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma; x : A \vdash C[x]}{\Gamma \vdash C[a]} \quad (\mathit{Let})$$

A.1.2 Membership Rule

$$\frac{\Gamma \vdash t \in A}{\Gamma \vdash (t \in A)\ \mathrm{Type}} \quad (W_\in)$$

$$\frac{\Gamma \vdash t \in A}{\Gamma \vdash \bullet \in (t \in A)} \quad (M_\in)$$

$$\frac{\Gamma \vdash A\ \mathrm{Type}}{\Gamma \vdash \bullet \in (A\ \mathrm{Type})} \quad (M_{\mathit{Type}})$$

[1] Some of this rules are redundant. For example most of introduction rules are derivable from their membership introduction counterparts. The (*Let*) rule is derivable from the (*Cut*) rule using function type.

$$\frac{}{\Gamma; x : A; \Delta[x] \vdash x \in A} \quad (I_\in)$$

$$\frac{\Gamma \vdash t \in A}{\Gamma \vdash A} \quad (E_\in)$$

A.1.3 Disjunction Rules

$$\frac{\Gamma \vdash A \,\mathrm{Type} \qquad \Gamma \vdash B \,\mathrm{Type}}{\Gamma \vdash A \vee B \,\mathrm{Type}} \quad (W_\vee)$$

$$\frac{\Gamma \vdash a \in A}{\Gamma \vdash \mathtt{inl}\, a \in A \vee B} \quad (M^1_\vee)$$

$$\frac{\Gamma \vdash b \in B}{\Gamma \vdash \mathtt{inr}\, b \in A \vee B} \quad (M^2_\vee)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad (I^1_\vee)$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A_1 \vee A_2} \quad (I^2_\vee)$$

$$\frac{\Gamma; x : A; \Delta[\mathtt{inl}\, x] \vdash C[\mathtt{inl}\, x] \qquad \Gamma; y : B; \Delta[\mathtt{inr}\, y] \vdash C[\mathtt{inr}\, y]}{\Gamma; z : A; \Delta[z] \vee B \vdash C[z]}$$

$$(E_\vee)$$

A.1.4 Universal Quantifier Rules

$$\frac{\Gamma; x : A \vdash B[x] \,\mathrm{Type}}{\Gamma \vdash \forall x : A.B[x] \,\mathrm{Type}} \quad (W_\forall)$$

$$\frac{\Gamma; x : A \vdash f x \in B[x]}{\Gamma \vdash f \in \forall x : A.B[x]} \quad (M_\forall)$$

$$\frac{\Gamma; x : A \vdash B[x]}{\Gamma \vdash \forall x : A.B[x]} \quad (I_\forall)$$

$$\frac{\Gamma; f : \forall x : A.B[x]; \Delta[f] \vdash a \in A}{\Gamma; f : \forall x : A.B[x]; \Delta[f] \vdash f a \in B[a]} \quad (E_\forall)$$

A.1.5 Existential Quantifier Rules

$$\frac{\Gamma; x : A \vdash B[x] \,\mathrm{Type}}{\Gamma \vdash \exists x : A.B[x] \,\mathrm{Type}} \quad (W_\exists)$$

$$\frac{\Gamma; x : A \vdash a \in A \qquad \Gamma; x : A \vdash b \in B[a]}{\Gamma \vdash \langle a, b \rangle \in \exists x : A.B[x]} \quad (M_\exists)$$

$$\frac{\Gamma; x : A \vdash a \in A \qquad \Gamma; x : A \vdash B[a]}{\Gamma \vdash \exists x : A.B[x]} \tag{$I_\exists$}$$

$$\frac{\Gamma; x : A; y : B[x]; \Delta[\langle x, y\rangle] \vdash C[\langle x, y\rangle]}{\Gamma; z : \exists x : A.B[x]; \Delta[z] \vdash C[z]} \tag{$E_\exists$}$$

A.1.6 Falsum Rules

$$\frac{}{\Gamma \vdash \bot\ \text{Type}} \tag{$W_\bot$}$$

$$\frac{}{\Gamma; x : \bot; \Delta[x] \vdash C} \tag{$E_\bot$}$$

A.1.7 Computation Rules

$$\frac{\Gamma \vdash b \in T \qquad \text{“}a \mapsto b\text{”}}{\Gamma \vdash a \in T} \tag{$Reduce$}$$

Usual reduction rules: $\lambda x.a[x]\ b \longrightarrow a[b]$, *etc*

A.1.8 Arithmetical Rules

Induction, *etc*

We assume the following definitions:

$A \to B = \forall x : A.B \quad A \wedge B = \exists x : A.B$, where x is not free in B

$\neg A = A \to \bot \qquad \mathtt{fix}(f.p[f]) = (\lambda x.p[xx])(\lambda x.p[xx])$

We can establish the property 7.1.4 in this fragment by a straightforward induction on the derivation.

A.2 NuPRL-4 Quotient Rules

NuPRL-4 [CAB+86] has the following rules for the quotient type:

$$\frac{\begin{array}{l} H \vdash A_1 = A_2 \in \mathbb{U} \\ H;\, x : A_1;\, y : A_1 \vdash E_1[u_1; v_1] = E_2[u_2; v_2] \in \mathbb{U} \\ H \vdash \text{“}E \text{ is an ER on } A\text{”} \end{array}}{H \vdash (u_1, v_1 : A_1//E_1) = (u_2, v_2 : A_2//E_2) \in \mathbb{U}}$$

(*quotientWeakEquality*)

$$\frac{\begin{array}{l} H \vdash (x, y : A//E) = (x, y : A//E) \in \mathbb{U} \\ H \vdash a \in A \end{array}}{H \vdash a \in x, y : A//E}$$

(*quotient-memberFormation*)

$$\frac{\begin{array}{l} H \vdash (x, y : A//E) = (x, y : A//E) \in \mathbb{U} \\ H \vdash a_1 = a_2 \in A \end{array}}{H \vdash a_1 = a_2 \in (x, y : A//E)}$$

(*quotient-memberWeakEquality*)

$$\frac{\begin{array}{l} H;\, u : (x, y : A//E);\, J;\, v : A;\, w : A \vdash E[x; y] = E[x; y] \in \mathbb{U} \\ H;\, u : (x, y : A//E);\, J \vdash T = T \in \mathbb{U} \\ H;\, u : (x, y : A//E);\, J;\, v : A;\, w : A;\, z : E[x; y] \vdash s[u] = t[u] \in T[u] \end{array}}{H;\, u : (x, y : A//E);\, J \vdash s = t \in T}$$

(*quotientElimination*)

$$\frac{\begin{array}{l} H;\, u : (x, y : A//E);\, J;\, v : A;\, w : A \vdash E[x; y] = E[x; y] \in \mathbb{U} \\ H;\, u : (x, y : A//E);\, J \vdash T = T \in \mathbb{U} \\ H;\, u : (x, y : A//E);\, v : A;\, w : A;\, z : E[x; y];\, J[u] \vdash s[u] = t[u] \in T[u] \end{array}}{H;\, u : (x, y : A//E);\, J \vdash s = t \in T}$$

(*quotientElimination-2*)

$$\frac{\begin{array}{l} H;\, u : (a = b \in (x, y : S//E));\, [v : E[x; y]];\, J \vdash t \in T \\ H;\, u : (a = b \in (x, y : S//E));\, J \vdash E[x; y] = E[x; y] \in \mathbb{U}\{j\} \end{array}}{H;\, u : (a = b \in (x, y : S//E));\, J \vdash t \in T} \quad (\textit{quotient-equalityElimination})$$

$$\frac{\begin{array}{ll} H \vdash x, y : A//E \in \mathbb{U} & H \vdash u, v : B//F \in \mathbb{U} \\ H \vdash A = B \in \mathbb{U} & H;\, w : (A = B \in \mathbb{U});\, r : A;\, s : A \vdash E[x; y] \;\Leftrightarrow\; F[u; v] \end{array}}{H \vdash (x, y : A//E) = (u, v : B//F) \in \mathbb{U}} \quad (\textit{quotientEquality})$$

$$\frac{\begin{array}{l} H \vdash (x, y : A//E) = (x, y : A//E) \in \mathbb{U} \\ H \vdash s = s \in A \quad H \vdash t = t \in A \quad H \vdash E[x; y] \end{array}}{H \vdash s = t \in (x, y : A//E)} \quad (\textit{quotient-memberEquality})$$

A.2.1 Comparison to the Proposed Rules

The (*IquotMemEqual*) rule (see Section 5.7) is similar to NuPRL-4 (*quotient-memberEquality*) rule. However, in the (*IquotMemEqual*) rule, the first assumption has the equality predicate `esquash`ed. Also, the next two assumptions only require x and y to be in $A/\!/E$, not necessarily in A, as NuPRL-4 does (which also allowed to get rid of well-typeness assumption and only leave it in (*IquotMemIntro*)). These two changes make the rule much stronger than it was in NuPRL-4.

The (*IquotElim*) rule may look unusual, but it is just a generalization (and simplification) of NuPRL-4 (*quotientElimination*) and (*quotientElimination-2*) rules. It does not need the well-typeness assumption for E since we are using the intensional quotient type. It is important to mention that while NuPRL-4 quotient

Table A.1: Concrete Syntax of Sequent Schemata

Description	Abstract syntax	Concrete syntax
Sequent turnstyle	$\vdash$	`>-`
First-order variable	a	`'a`
Second-order variable	$A_{\{\mathbf{H};\mathbf{J}\}}[x;y]$	`'A<H;J>['x;'y]`
Sequent context	$\mathbf{J}_{\{\mathbf{H}\}}[x]$	`'J<H>['x]`
Sequent example	$\mathbf{H};\, x : A;\, \mathbf{J}[x] \vdash C[x]$	`'H; x:'A; 'J['x] >- 'C['x]`
Rule	$\dfrac{S_1 \cdots S_n}{S}$	S_1 `-->` $\cdots$ `-->` S_n `-->` S

elimination rules are irreversible, the (*IquotElim*) rule is reversible. In fact, all the rules of Section 5.7.2 are reversible, except for the (*IquotMemIntro*).

The (*IquotEqualElim*) rule is similar to NuPRL-4 (*quotient-equalityElimination*) rule, but it does not have an extra well-typeness assumption for E that is only necessary for extensional quotient type. The (*IquotEqualIntro*) and (*IquotMem-Intro*) rules are the same as NuPRL-4 rules (*quotientWeakEquality*) and (*quotient-memberFormation*) and are only presented here for completeness.

A.3 Concrete Syntax of Sequent Schemata

Table A.1 describes the concrete syntax used by the MetaPRL system.

BIBLIOGRAPHY

[ABI+96] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16(3):321–353, June 1996.

[ACE+00] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The NuPRL open logical environment. In D. McAllester, editor, *Automated deduction – CADE-17 : 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes on Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.

[Acz86] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definition. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.

[All87a] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[All87b] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 215–224. IEEE, June 1987.

[AMdPR01] Natasha Alechina, Michael Mendler, Valeria de Paiva, and Eike Ritter. Categorical and Kripke semantics for constructive S4 modal logic. In Fribourg [Fri01], pages 292–307.

[Arm00] William Y. Arms. Automated digital libraries. How effectively can computers be used for the skilled tasks of professional librarianship? *D-Lib Magazine*, 6(7/8), July/August 2000. `http://www.dlib.org/dlib/july00/arms/07arms.html`.

[Asp00] David Aspinall. Proof General — A generic tool for proof development. In *Proceedings of TACAS*, volume 1785 of *Lecture Notes in Computer Science*, 2000. `http://zermelo.dcs.ed.ac.uk/home/da/papers/pgoutline/`.

[Bac84] Roland Backhouse. A note on subtypes in Martin Löf's theory of types. Technical Report CSM-70, University of Essex, November 1984.

[Bat79] J. L. Bates. *A Logic for Correct Program Development*. PhD thesis, Cornell University, 1979.

[BBC+96] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.

[BCH+00] Ken Birman, Robert Constable, Mark Hayden, Jason J. Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161. IEEE, 2000.

[BCMS89] Roland C. Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[Bee85] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[Bib87] W. Bibel. *Automated Theorem Proving*. Vieweg Verlag, Braunschweig, 2nd edition, 1987.

[BK91] David A. Basin and Matt Kaufmann. The Boyer-Moore prover and NuPRL: An experimental comparison. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 89–119. Cambridge University Press, 1991.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[BR88] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. Cambridge University Press, Cambridge, 1988.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Development System*. Prentice-Hall, NJ, 1986.

[Cal98] James Caldwell. *Decidability Extracted: Synthesizing "Correct-by-Construction" Decision Procedures from Constructive Proofs*. PhD thesis, Cornell University, 1998. Cornell TR98-1722.

[CC01] Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. In Wilfried Sieg, Richard Sommer, and Carolyn Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, Lecture Notes in Logic, pages 166–183. Association for Symbolic Logic, 2001.

[CDC80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[CMT02] Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors. *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.

[Con83] Robert L. Constable. Mathematics as programming. In *Proceedings of the Workshop on Programming and Logics, Lectures Notes in Computer Science 164*, pages 116–128. Springer-Verlag, 1983.

[Con98] Robert L. Constable. Types in logic, mathematics and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter X, pages 683–786. Elsevier Science B.V., 1998.

[Cou01] Pierre Courtieu. Normalized types. In Fribourg [Fri01], pages 554–569.

[CZ84] Robert L. Constable and D. R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, January 1984.

[Dyc92] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, volume 57(3), September 1992.

[FM97] Matt Fairtlough and Michael Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997.

[Fri01] L. Fribourg, editor. *Computer Science Logic, Proceedings of the 10th Annual Conference of the EACSL*, volume 2142 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. `http://link.springer-ny.com/link/service/series/0558/tocs/t2142.htm`.

[GM93] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Oder Logic*. Cambridge University Press, Cambridge, 1993.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[HAB+] Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. http://metaprl.org/theories.pdf.

[Har96] John Harrison. HOL Light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[Hay98] Mark Hayden. *The Ensemble System*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1998.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993. A revised and expanded verion of '87 paper.

[Hic97] Jason J. Hickey. NuPRL-Light: An implementation framework for higer-order logics. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes on Artificial Intelligence*, pages 395–399, Berlin, July 13–17 1997. Springer. CADE '97. An extended version of the paper can be found at http://www.cs.caltech.edu/~jyh/papers/cade14_nl/default.html.

[Hic99] Jason J. Hickey. Fault-tolerant distributed theorem proving. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 227–231, Berlin, July 7–10 1999. Springer. CADE '99.

[Hic01] Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[HL78] Gérard P. Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HN00] Jason J. Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in*

Higher Order Logics: 13th International Conference, TPHOLs 2000, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2000.

[HNK+] Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.

[Hof95a] Martin Hofmann. *Extensional concepts in intensional Type theory.* PhD thesis, University of Edinburgh, Laboratory for Foundations of Computer Science, July 1995.

[Hof95b] Martin Hofmann. A simple model for quotient types. In *Typed Lambda Calculus and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 216–234, 1995.

[How96] Douglas J. Howe. Semantic foundations for embedding HOL in NuPRL. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101. Springer-Verlag, Berlin, 1996.

[How98] Douglas J. Howe. A type annotation scheme for NuPRL. In *Theorem Proving in Higher-Order Logics*. Springer, 1998.

[Jac95] Paul B. Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, Ithaca, NY, January 1995.

[KM97] Matt Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[KN01] Alexei Kopylov and Aleksey Nogin. Markov's principle for propositional type theory. In Fribourg [Fri01], pages 570–584.

[KO99] Christoph Kreitz and Jens Otten. Connection-based theorem proving in classical and non-classical logics. *Journal for Universal Computer Science, Special Issue on Integration of Deductive Systems*, 5(3):88–112, 1999.

[Kop00] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. Department of Computer Science TR2000-1809, Cornell University, 2000.

[KS00] Christoph Kreitz and Stephan Schmitt. A uniform procedure for converting matrix proofs into sequent-style systems. *Journal of Information and Computation*, 162(1–2):226–254, 2000.

[LKvR+99] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating Systems Principles*, December 1999.

[LSBB92] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

[MA94] Conal L. Mannion and Stuart F. Allen. A notation for computer aided mathematics. Department of Computer Science TR94-1465, Cornell University, Ithaca, NY, November 1994.

[Mar54] A.A. Markov. On the continuity of constructive functions. *Uspekhi Matematicheskikh Nauk*, 9/3(61):226–230, 1954. In Russian.

[Mar62] A.A. Markov. On constructive mathematics. *Trudy Matematicheskogo Instituta imeni V.A. Steklova*, 67:8–14, 1962. In Russian. English Translation: A.M.S. Translations, series 2, vol.98, pp. 1-9. MR 27#3528.

[May97] S. Tobias Mayr. Generating primitive proofs from SupInf. Communicated to the NuPRL group at Cornell University, 1997.

[McC63] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. Amsterdam:North-Holland, 1963.

[McC94] William W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994. See also `http://www-unix.mcs.anl.gov/AR/otter/`.

[McC97] William W. McCune. 33 basic test problems: A practical evaluation of some paramodulation strategies. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 5, pages 71–114. MIT Press, 1997. `http://www-unix.mcs.anl.gov/~mccune/papers/33-basic-test-problems/`.

[Men88] P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[MN94] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer-Verlag, 1994.

[NH02] Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Carreño et al. [CMT02], pages 281–297. See also `http://nogin.org/papers/derived_rules.html`.

[Nog02a] Aleksey Nogin. Quotient types — a modular approach. Department of Computer Science TR2002-1869, Cornell University, April 2002. See also `http://nogin.org/papers/quotients.html`.

[Nog02b] Aleksey Nogin. Quotient types: A modular approach. In Carreño et al. [CMT02], pages 263–280. See also `http://nogin.org/papers/quotients.html`.

[NP83] Bengt Nordström and Kent Petersson. Types and specifications. In *IFIP'93*. Elsvier, 1983.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

[NSM01] Pavel Naumov, Mark-Olivar Stehr, and José Meseguer. The HOL/NuPRL proof translator: A practical approach to formal interoperability. In *The 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland*, Lecture Notes in Computer Science, pages 329–345. Springer-Verlag, September 2001.

[ORS] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. pages 748–752.

[Pal95] Erik Palmgren. The Friedman translation for Martin-Löf's type theory. *Mathematical Logic Quarterly*, 41:314–326, 1995.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1994.

[PD01] Frank Pfenning and Rowan Davies. Judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4), August 2001.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming*

Language Design and Implementation (PLDI), pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[Pfe89] Frank Pfenning. Elf: a language for logic definition and verified metaprogramming. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[Pfe01a] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS-2001)*, Boston, Massachusetts, June 2001.

[Pfe01b] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2. Elsevier Science Publishers, 2001.

[Pie91] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[PN90] L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge Computing Laboratory, 1990.

[Pot80] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.

[Rhi01] Morten Rhiger. *Higher-Order Program Generation.* PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, August 2001. `http://www.brics.dk/DS/01/4/`.

[Rhi02] Morten Rhiger. Compiling embedded programs to byte code. In Shriram Krishnamurthi and C.R. Ramakrishnan, editors, *Proceedings of the Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in Lecture Notes in Computer Science, pages 120–136, Portland, Oregon, January 2002. Springer-Verlag.

[SLKN01] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 421–426. Springer-Verlag, 2001.

[SORSC99] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[Tam97] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, April 1997.

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[Tro73] Anne Sjerp Troelstra. *Metamathematical Investigation of Intuitionistic Mathematics*, volume 344 of *Lecture Notes in Mathematics.* Springer-Verlag, 1973.

[TvD88] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction, Vol. I,II.* North-Holland, Amsterdam, 1988.

[Wei97] Christoph Weidenbach. SPASS — version 0.49. *Journal of Automated Reasoning*, 18(2):247–252, April 1997.

[Wij90] Duminda Wijesekera. Constructive modal logics I. *Annals of Pure and Applied Logic*, 50:271–301, 1990.

[WL99] Pierre Weis and Xavier Leroy. *Le langage Caml.* Dunod, Paris, 2nd edition, 1999. In French.