Metamathematical Extensibility in Type Theory

Todd Knoblock
Ph.D. Thesis

87-892
December 1987

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

METAMATHEMATICAL EXTENSIBILITY IN TYPE THEORY

A Thesis
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

Todd B. Knoblock

METAMATHEMATICAL EXTENSIBILITY IN

TYPE THEORY

Todd B. Knoblock, Ph.D.

Cornell University

An automated theorem prover is said to be *metamathematically extensible* if a metalanguage can be employed by the user to soundly extend the reasoning capabilities of the system. In this thesis, we present a framework for metamathematical extensibility for a system based upon a type-theoretic logic, the Nuprl system. Using this framework, the user can construct programs called *proof tactics* that may be used to provide new reasoning capabilities for the logic. These proof tactics can encode reasoning methods as simple as a derived rule of inference or as ambitious as a theorem prover.

The design of the framework ensures that all proof tactics are correct. A formal metalanguage called Metaprl is defined that represents the proof theory of Nuprl in a natural and computationally-oriented fashion. The logic of Metaprl is an extension of the constructive type theory of Nuprl. Type theories like Nuprl and Metaprl are distinguished by the uniform treatment of computations (programs) and logical propositions and by rich languages for expressing computations. In Metaprl, formal specifications for tactics may be written and formally correct tactics extracted from the proofs of the specification.

Three classes of tactics are defined: *complete tactics*, *partial tactics*, and *search tactics*. Complete tactics are analogous to derived axioms for the Nuprl logic. Partial tactics are analogous to derived rules of inference. Search tactics are analogous to the procedural tactics of LCF and the current Nuprl system. Examples from each class of tactics are presented.

There are a number of advantages to using a formal logic as a metalanguage for metamathematical extensibility. System-implemented *reflection principles* guarantee that the framework is a conservative extension of Nuprl. The expressiveness of the Metaprl logic allows the *validity* of tactics to be ensured. Often it is not

even necessary to execute tactics since they have been proved correct and all that is required is that a proof exists; the exact form of the proof is not needed. This can result in substantial computational savings.

Finally, the construction of a metalanguage for metamathematical extensibility for Nuprl is generalized to a hierarchy of metalanguages, each logic providing for metamathematical extensibility of the preceeding logic.

To Kathleen

Acknowledgements

I thank my advisor, Robert Constable, for his guidance and encouragement. His enthusiasm for science and his depth of knowledge were inspiring. I would like to acknowledge the interest and encouragement of the members of my committee: Prakash Panangaden, Richard Shore, and Alan Demers.

It has been my privilege to be associated with the "PRL group". I thank the group and especially Joseph Bates, Mark Bromley, Rance Cleaveland, Timothy Griffin, Robert Harper, Douglas Howe, and James Sasaki for the weekly seminars and daily arguments (discussions). Special thanks go to Timothy Griffin who provided valuable comments on a draft of this thesis and to Joseph Bates who originally sparked my interest in Nuprl.

I thank all my friends at Cornell who made my time in Ithaca pass quickly. I especially want to thank Anne Rogers, Michael Karasick, and Earl Zmijewski who provided shelter and logistical support in the final days.

I thank my parents, Janet, Peter, and Ellen, for their support throughout my studies. Finally, I thank my wife Kathleen for her love and patience.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Metamathematical Extensibility

An automated theorem prover is said to be *metamathematically extensible* if a metalanguage can be employed by the user to soundly extend the reasoning capabilities of the system. In this thesis, we present a framework for metamathematical extensibility for a system based upon a type-theoretic logic, the Nuprl system. Using this framework, the user can construct programs called *proof tactics* that may be used to provide new reasoning capabilities for the logic. These proof tactics can be used to encode reasoning methods as simple as a derived rule of inference or as ambitious as a theorem prover.

The Nuprl system is an interactive computer environment for developing formal proofs and verified programs. The proofs are written in a logic that is a version of constructive type theory. Correct programs can be *extracted* from proofs in this theory; given a proof of a sentence of the form,

$$\forall x \in A.\exists y \in B.P(x,y),$$

where A and B are types and P is a predicate, then a program, f, can be extracted from this proof which satisfies the condition that for any $x \in A$, $f(x) \in B$ and $P(x,f(x))$ is true (P acts as a formal specification for the program). This correspondence between proofs and programs can be exploited in constructing formal proofs by using a type-theoretic programming logic to specify and express tactics.

The Nuprl system is not an *automatic theorem prover*, but rather is intended to provide an environment in which mathematics can be interactively formalized and checked. The logic of Nuprl is a version of Martin-Löf's intuitionistic type theory [81], and a descendant of the Automath family of logics (de Bruijn [48]). The logic is distinguished in treating programs (computations) directly.

In any logic, writing a formal proof is a combination of insightful, interesting steps and tedious ones. Without some form of automation, the sheer detail required to construct a large collection of formal proofs would be overwhelming. The ideal is to allow the user to guide the system and supply just the main insights, the central ideas of a proof. McCarthy [86] said:

> We envisage the use of computer proof-checking in mathematics as follows: The mathematician already has formalizations of this branch of mathematics and the computer system has stored in it the theorems that have previously been proved. In addition, there are a number of techniques embodied in programs for generating proofs. The mathematician expresses his ideas of how a proof may be found by combining theses techniques into a program. The computer carries out the program which may prove the theorem, may generate information that will guide another try, may indicate an elementary misconception, or may be of no help whatsoever.

This is very much the view we have taken in designing Nuprl. The proof-constructing programs that McCarthy alludes to were called *proof tactics* by Gordon, Milner and Wadsworth in LCF [58]. Proof development systems, like Nuprl, should support some form of user-definable extensions to the reasoning capabilities of the system, what Davis and Schwartz [47] dubbed *metamathematical extensibility*.

In this thesis, we present a formal programming language, Metaprl, that is a metalanguage for Nuprl. We show how tactics can be defined in Metaprl, and how the implementation of the proof development system can support the use of tactics for metamathematical extensibility. Three distinct classes of tactics are defined with different properties: *complete tactics, partial tactics* and *search tactics*. Complete tactics are analogous to derived axioms for Nuprl, and partial tactics are analogous to derived rules of inference. These are only analogies since complete and partial tactics may perform more computation to verify applicability and to calculate subgoals than is ordinarily associated with derived axioms and rules. The third kind of tactics, search tactics, are analogous to derived rules of inference that do not have an explicit domain of goals to which they apply. Search tactics are similar to the LCF style of tactics in that the correctness is not proved, but verified by computation. Nevertheless, more can be proved about search tactics than about LCF tactics; in particular, the user can prove (in a machine verifiable form) that a search tactic does not diverge, and that it is a *valid* tactic in the sense defined by Gordon, Milner, and Wadsworth [58].

The logic of Metaprl is a constructive type theory that is very similar to the logic of Nuprl. Writing a collection of tactics in Metaprl is an activity closely related to the development of a theory in Nuprl. Therefore, for all the reasons that a formal programming language for metamathematical extensibility is desirable for Nuprl, it is equally desirable for Metaprl. Thus, the construction of one metalanguage, Metaprl, for the logic, Nuprl, is generalized to a hierarchy of logics, $L^0, L^1, L^2, \ldots$, each serving as a metalanguage for its predecessor.

1.2 Automatic Theorem Proving

Automated theorem proving systems have a relatively long history in computer science. Despite the sustained interest, the powerful, general purpose theorem provers that were envisioned in the 1950's and 60's have yet to be realized. There have been several extremely successful special purpose theorem provers, for example the geometry theorem prover of Wu [125], and the work on automatic proof of the existence of inclusive predicates of Mulmuley [94]. General purpose automatic theorem provers have not been as successful. The failures of the general purpose automatic theorem provers can be roughly categorized into two groups. Those that failed to be general, proving only a very limited class of goals, and those that failed to be automatic, requiring the user to guide the system towards a proof. It appears that it is not possible with the present technology to construct automatic theorem provers with even modest generality. Barring major improvements in technology, we are left to strive for *automated* theorem provers that require user guidance.

Existing automated theorem provers that require user guidance vary in how directly and by what mechanisms the user can guide the system towards a proof. In some, the user must supply *interpolating* lemmas that fall between the current achievements and the goal. In some, the user can *focus* the system on a subset of the current achievements that will be relevant to the proof of the goal, and thus control the combinatorial explosion in the search space. These mechanism for guidance are indirect. In still other automated theorem provers, the user can activate and focus special purpose reasoners. In a few systems, the user can imperatively guide the system to the proof of the goal. These are direct mechanisms of guidance.

It is not really surprising that research on automatic theorem proving has met with only limited success. Theorem proving is not a routine task, nor is it amenable to brute force attack. Some proofs contain incredibly clever and unintuitive steps or proof techniques. There are theoretical limitations, combinatorial limitations and unsolved problems in knowledge representation all of which must be overcome before automatic theorem proving would be possible.

Automatic theorem proving would be even more difficult for logics like Nuprl which include mechanisms for extracting programs from proofs. A successful automatic theorem prover for Nuprl would not only have to prove a wide class of theorems, but the proofs would have to be constructed in such a way as to result in efficient executable programs. Thus, it would also have to solve the problem of automatic programming in the context of the Nuprl programming language.

1.3 Interactive Proof Development

There have been several projects that have sought to solve the more tractable problem of *proof-checking*. In 1962, McCarthy [86] wrote about the possibility of using computers to check proofs and verify program correctness.

> Checking mathematical proofs is potentially one of the most interesting and useful applications of automatic computers. Computers can check not only the proofs of new mathematical theorems but also proofs that complex engineering systems and computer programs meet their specifications. Proofs to be checked by computer may be briefer and easier to write than the informal proofs acceptable to mathematicians. This is because the computer can be asked to do much more work to check each step than a human is willing to do, and this permits longer and fewer steps... The combination of proof-checking techniques with proof finding heuristics will permit mathematicians to try out ideas for proofs that are still quite vague and may speed up mathematical research.

Existing proof-checking systems include AUTOMATH (de Bruijn [48]) and PL/CV3 (Constable and Zlatin [40]), both of which were models for the Nuprl system.

When a student learns a mathematical theory, he learns more than definitions and theorems; he learns patterns and techniques of proof. Each theory, be it recursion theory or program verification, has its own suite of useful proof techniques and heuristic rules for when they are relevant. Automated system that support proof development on a machine should also support the encoding of proof methods. It is unrealistic to expect the implementors of such systems to include every conceivable proof strategy, semi-decision procedure, and expression simplification method that is relevant to any theory that might be formalized. There must be some mechanism that allows the user to incorporate new proof tactics into the system.

There are a number of possible mechanisms that would allow new proof tactics to be introduced. The user could be allowed to alter the implementation to accommodate a new proof tactic. This has the advantage that the tactic can be efficiently incorporated into the system. For some users, in particular the system implementors, this may work well. There are, however, two major disadvantages. First, Nuprl and most modern theorem proving environments are large and complicated programs. Nuprl is roughly 60,000 lines of LISP code. It requires a thorough knowledge of the system and the implementation to successfully implement a new rule of inference for Nuprl. The second major disadvantage is that with every addition to the system by the user, the entire system must be reassessed to determine if it still correctly implements the logic. Additionally, the implementation is too low-level a language to be appropriate for encoding proof tactics. A much more abstract language, far removed from manipulation of display forms, key strokes, and such, is called for.

A second way in which proof tactics might be introduced by the user would be to provide as part of the implementation a procedural metalanguage, a programming language in which to express tactics. This is the approach currently implemented in the Nuprl system (the author has reported on this approach in Constable, Knoblock, and Bates [36] and Constable et al. [41]). A modified version of the ML programming language from the LCF project (Gordon, Milner, and Wadsworth [58]) is employed as a procedural metalanguage. In addition to the usual programming language types, ML is endowed with types representing the terms, rules, and proofs of the Nuprl logic. The user writes programs to manipulate these data types in ML and the system implements facilities for invoking these programs to construct Nuprl proofs.

This approach does solve the problems associated with allowing the user to alter the system implementation directly. By prudently arranging the way in which the user's programs can manipulate the proof data type, it is possible to know that every tactic implemented in ML results in a correct Nuprl proof. The system monitors the construction of proofs so as to guarantee that each inference step is a valid instance of a rule in the logic. Thus proof tactics introduced by the user are guaranteed to be consistent extensions of the logic. The ML programming language is an elegant high-level language which presents the user with an appropriately abstract language for expressing tactics. Finally, although not as efficient as tactics that are part of the implementation, tactics expressed in ML are efficient.

There are, however, remaining drawbacks to using a procedural metalanguage for metamathematical extensibility. The formalization of a mathematical theory in Nuprl consists of three conceptually distinct activities: proving theorems, defining syntactic notations, and defining relevant proof tactics. The use of a procedural metalanguage requires a distinction between theorem proving and tactic writing. Nuprl contains a programming language, and yet the user is asked to express his proof tactics in a distinct programming language, ML. Another problem is that the only way for the system to ensure that the proof generated by a tactic is correct is to execute the tactic and verify the resulting proof. Even though the user may know (in fact, may be able to prove) that a given tactic will construct a correct proof of a given goal, the implementation must still execute the tactic and verify the result.

A third way in which metamathematical extensibility may be implemented is to use a formal programming language as a metalanguage for the logic. By *formal programming language* we mean a programming language and associated logic in which properties of programs expressed in the language may be proved. The difficulty in using ML or another procedural programming language as the metalanguage is not that properties of programs cannot be proved, of course they can be using any of the standard program verification techniques, but that the proofs cannot be checked by the machine. Thus the behavior of the system cannot be altered in a reliable way based upon the proved properties. If a formal programming language is used, then the system can check the proofs and adjust the system behavior based upon the known properties of tactics. Thus, for example, if the user proves that a tactic τ works on the set of goals, S, and it is known that goal g is in S, then the tactic does not have to be executed in order to generate a proof of g. The expression $\tau(g)$ is a proof of g. This can result in a substantial computational savings. In addition, there are the usual aesthetic and practical advantages to using verified programs.

1.4 Closely Related Work

The LCF project (Gordon, Milner, and Wadsworth [58]) contains powerful facilities for user-defined extensions to the reasoning engine of the system that do not rely on modifications to the implementation. LCF includes the programming language ML. ML has facilities for manipulating objects of the formal system of LCF; using these facilities, the user can write tactics to prove theorems in a high-level language. Complex tactics may be constructed from simpler tactics using tactic combining forms called *tacticals*. LCF is the model for the current procedural metalanguage interface of Nuprl. However, there are a number of differences between procedural tactics in LCF and procedural tactics in Nuprl stemming from a different treatment of proof in the two theories. See Constable, Knoblock, and Bates [36] for a comparison of the approaches.

There have been several recent theories that allow metatheoretic extensions via a mechanized formal metatheory. These include the automated theorem prover FOL of Aiello, Aiello and Weyhracuh [1,2,123], the theorem prover of Boyer and

Moore [19,18] which we call *BMT*, and the proposed system FS of Davis and Schwartz [47]. In addition to the LCF project, the research of Boyer and Moore, and of Davis and Schwartz have directly influenced this work.

The FOL system does allow the expression of formal tactics, however its *semantic attachment* mechanism and the fact that a proof of correctness of tactics is not required allow incorrect proof procedures. The EKL system of Ketonen and Weening [72,71] also admits tactics that can result in incorrect proofs.

In Boyer and Moore's theory the syntax and meaning (evaluation) of terms of the formal theory are axiomatized. The user can define *metafunctions* which are functions mapping terms to terms that are proved to preserve meaning. The close relation between the formal system's program structure and the implementation language's structure allows user defined metafunctions to be incorporated into the implementation mechanically, resulting in very efficient rewriting functions.

The BMT style of metafunction can only serve for term rewriting and not for general proof development. The complexity of the Nuprl term structure and evaluation procedure also preclude using the metafunction approach in Nuprl for the full theory. However, on restricted domains of terms, the approach is feasible. Howe [67] contains a study of reflected term rewriting for Nuprl using this approach.

In FS new rules of inference are expressed by the user in a decidable subtheory of FS called *LFS*. If the user proves that a formula of this subtheory forms a conservative extension of the theory when adjoined as a new rule, then the formula may be used as a rule of inference in a proof. Each time a new formula is admitted by the system as a rule of inference the underlying theory is changed, along with the reflected axiomatization of proofs, to include the new rule. Applications of these rules are carried out using an explicit reflection principle.

In contrast to the small subtheory, LFS, in which derived rules are defined in FS, we allow the full expressive power of the Nuprl logic to be used in writing tactics. The semantics of the proposed tactic mechanism is such that the underlying theory does not change with the addition of a new tactic. Like their system, theorems proved using metatheoretic extensions will generally be shorter than the corresponding proof that uses only primitive rules of inference. However, because we extract functions from proofs, it will be necessary in some cases to calculate primitive proofs in order to extract. This is always possible in our system because tactics are proved to be conservative extensions using constructive methods.

1.5 Theoretical Limitations

The goal of this research is to provide a framework for metamathematical extensibility for Nuprl and similar logics. The Nuprl language is both a logic and a programming language, and so appears to be a good language for expressing tactics. An obvious question is "can Nuprl serve as a framework for metamathematical extensibility for itself?". The question is too vague for a definitive negative answer. However, we can prove that some natural approaches are doomed to failure.

To show that these are general limitations and not particular to the Nuprl logic, we present the following results for a generic logic where we assume a minimum of structure. The arguments should apply to any logic with *reasonable expressibility* and the requisite structure. We begin by reviewing some standard metamathematical results.

Let T be a programming logic, such as type theory. We know from the work of Gödel and Löb that any *reasonably expressive* logic is capable of representing provability in T. In particular, there is a *Gödel numbering* of formulae, denoted $\ulcorner \cdot \urcorner$, and a predicate, Pr, definable in the language of T where Pr is a provability predicate. Pr is said to be a *provability predicate for T* if and only if the following properties hold. For any formulae A and B,

1. $\vdash^T A \Rightarrow \vdash^T Pr(\ulcorner A \urcorner)$
2. $\vdash^T Pr(\ulcorner A \rightarrow B \urcorner) \rightarrow Pr(\ulcorner A \urcorner) \rightarrow Pr(\ulcorner B \urcorner)$
3. $\vdash^T Pr(\ulcorner A \urcorner) \rightarrow Pr(\ulcorner Pr(\ulcorner A \urcorner) \urcorner)$

A provability predicate Pr is said to be *sound for T* if and only if

$$\vdash^T Pr(\ulcorner A \urcorner) \Rightarrow \vdash^T A.$$

In the following discussion we shall assume that the logic T is endowed with a Gödel numbering of the formulae and a (not necessarily sound) provability predicate, Pr.

Theorem 1.1 (Löb [79]) For any sentence A of T, if

$$\vdash^T Pr(\ulcorner A \urcorner) \rightarrow A,$$

then

$$\vdash^T A.$$

□

Let Con_T be the sentence $\neg Pr(\ulcorner false \urcorner)$. An immediate corollary of Löb's theorem is Gödel's second incompleteness theorem.

Theorem 1.2 If T is consistent, then

$$\nvdash Con_T.$$

Proof
Suppose $\vdash Con_T$. Then $\vdash Pr(\ulcorner false \urcorner) \rightarrow false$. Then, by Löb's theorem, $\vdash false$.

□

In order to serve as a framework for metamathematical extensibility, T must provide the facilities to write programs that manipulate and construct representations of proofs, and a *reflection principle* that connects representations of proofs within T to real proofs about the theory T. The former can be provided in any reasonable programming logic. Providing a general reflection principle for a logic is problematic.

Let R be the rule of inference

$$\frac{\vdash Pr(\ulcorner A \urcorner)}{\vdash A}.$$

This reflection principle would provide the necessary connection between the internal representation of proofs and real proofs. However, no consistent logic can admit such a rule.

Theorem 1.3 Let T be a programming logic with provability predicate Pr and in which R is an admissible rule. The logic T is inconsistent.

Proof
By Gödel's second incompleteness theorem, it suffices to show that $\vdash^T Con_T$. Using rule R, $\vdash Pr(\ulcorner false \urcorner) \rightarrow false$. Therefore, $\vdash Con_T$. Therefore, T is inconsistent.

□

In fact the rule R is stronger than necessary for a reflection principle for metamathematical extensibility. It allows proofs of $Pr(\ulcorner A \urcorner)$ posited on a set of assumptions to be a proof of A under the same set of assumptions. It would be sufficient if a proof of A could be deduced from a closed proof, a proof under no assumptions, of $Pr(\ulcorner A \urcorner)$. Let R' be the following rule of inference.

$$\frac{\vdash Pr(\ulcorner A \urcorner)\ (under\ no\ assumptions)}{\vdash A}$$

Rule R' would also provide the necessary connection between internal and external proofs. Unfortunately, admitting this rule in a logic also results in an inconsistent theory. To prove this we require a lemma that is the internal, formal analog to Löb's theorem (see Smoryński [114]).

Lemma 1.4 Let T be a logic with provability predicate Pr. For any formula G of T,

$$\vdash Pr(\ulcorner Pr(\ulcorner G \urcorner) \rightarrow G \urcorner) \rightarrow Pr(\ulcorner G \urcorner).$$

Proof
Let a formula G be given. Let $A := Pr(\ulcorner Pr(\ulcorner G \urcorner) \rightarrow G \urcorner) \rightarrow Pr(\ulcorner G \urcorner)$. By Löb's theorem, it suffices to prove $Pr(\ulcorner A \urcorner) \rightarrow A$.

1.	$\vdash Pr(\ulcorner A \urcorner)$	*assumption*
2.	$\vdash Pr(\ulcorner Pr(\ulcorner G \urcorner) \rightarrow G \urcorner)$	*assumption*
3.	$\vdash Pr(\ulcorner Pr(\ulcorner Pr(\ulcorner G \urcorner) \rightarrow G \urcorner) \urcorner)$	*provability property* 3 *on* 2
4.	$\vdash Pr(\ulcorner Pr(\ulcorner Pr(\ulcorner G \urcorner) \rightarrow G \urcorner) \urcorner) \rightarrow$ $Pr(\ulcorner Pr(\ulcorner G \urcorner) \urcorner)$	*provability property* 2 *on* 1
5.	$\vdash Pr(\ulcorner Pr(\ulcorner G \urcorner) \urcorner)$	*modus ponens* 3, 4
6.	$\vdash Pr(\ulcorner Pr(\ulcorner G \urcorner) \urcorner) \rightarrow Pr(\ulcorner G \urcorner)$	*provability property* 2 *on* 2
7.	$\vdash Pr(\ulcorner G \urcorner)$	*modus ponens* 5, 6

Therefore, $\vdash Pr(\ulcorner A \urcorner) \rightarrow A$. Therefore, by Löb's theorem, $\vdash A$. □

Suppose that T' is a logic that admits rule R' and has a provability predicate Pr. Note that Pr is a sound provability predicate. Given a formula, A, of T' and a closed proof, p, of $\vdash Pr(\ulcorner A \urcorner)$, then $\langle p, A \rangle$ is a proof of $\vdash A$, using rule R' to justify the last step. This reasoning can be formalized in T' yielding the following.

Lemma 1.5 Let T' be a logic that admits rule R' and has a provability predicate Pr. For every formula A of T',

$$\vdash^{T'} Pr(\ulcorner Pr(\ulcorner A \urcorner) \urcorner) \rightarrow Pr(\ulcorner A \urcorner).$$

□

We now show that admitting rule R' renders the logic inconsistent.

Theorem 1.6 Let T' be a logic that admits rule R' and has a provability predicate Pr. The logic T' is inconsistent.

Proof

1.	$\vdash Pr(\ulcorner Pr(\ulcorner false \urcorner) \urcorner) \rightarrow Pr(\ulcorner false \urcorner)$	*lemma* 1.5
2.	$\vdash Pr(\ulcorner Pr(\ulcorner Pr(\ulcorner false \urcorner) \urcorner) \rightarrow Pr(\ulcorner false \urcorner) \urcorner)$	*provability property* 3 *on* 1
3.	$\vdash Pr(\ulcorner Pr(\ulcorner Pr(\ulcorner false \urcorner) \urcorner) \rightarrow Pr(\ulcorner false \urcorner) \urcorner) \rightarrow$ $Pr(\ulcorner Pr(\ulcorner false \urcorner) \urcorner)$	*lemma* 1.4
4.	$\vdash Pr(\ulcorner Pr(\ulcorner false \urcorner) \urcorner)$	*modus ponens* 2, 3
5.	$\vdash Pr(\ulcorner false \urcorner)$	*modus ponens* 1, 4
6.	$\vdash false$	*rule* R'

□

Thus, we have shown that two obvious candidate rules, R and R', that would provide the required reflection between internal representations of proofs and real proofs in the theory render the theory inconsistent. These results do not prove that there cannot be a language supporting metamathematical extensibility for itself, only that these two direct methods fail. However, given the minimal requirements that have been placed on the formal systems and the provability predicates, they do suggest that all such approaches based upon general reflection principles will fail.

There are a number of ways in which to proceed. One would be to ignore the inconsistency brought about by the reflection principles. This is the approach taken in the FOL project. However, the designers of FOL were not concerned with such issues; their goals centered around the method of formalization of mathematical theories with the consistency of the underlying theory contingent upon the user. They in fact note that there are "embarrassing questions" which should not be presented to the prover (Weyhrauch [123], page 26). One of our main design criteria for a mathematically extensible language is that it preserves the consistency of the implementation.

A second approach would be to employ restricted versions of reflection principles. One such approach is described in an earlier paper by the author [74] based upon stratified internal evaluation functions. Although it contains some interesting aspects, it is deemed too complicated and cumbersome for practical use.

Another possible approach would be to view the addition of a proof tactic to the system not as a function operating within the theory but as resulting in a new, extended theory. The new theory should have the property that it is a conservative extension of the old; that is that anything provable in the new theory is also provable in the old theory. This is the approach proposed by Davis and Schwartz [47] and essentially the approach implemented by Boyer and Moore [18].

The approach taken in this thesis is to construct a distinct and explicit metalanguage in which tactics are expressed. Reflection principles can then be used to relate the levels without the problems of paradoxes associated with self-reflection.

1.6 Organization of the Thesis

The thesis is organized as follows. In chapter 2 we present a brief introduction to the Nuprl logic and system. In chapter 3, we define a formal proof metalanguage for Nuprl called Metaprl and outline a correctness proof for it. In chapter 4, we define three classes for formal tactics, complete, partial, and search tactics. We verify some properties of these classes and illustrate through examples how tactics would be constructed using Metaprl. In chapter 5, we describe how tactics would be employed in constructing proofs and what support for metamathematical extensibility must be supplied by the implementation. In chapter 6, we generalize the construction of a formal metalanguage for Nuprl to yield a hierarchy of formal metalanguages, each providing for metamathematical extensibility for its predecessor. In chapter 7, we conclude with a summary of results and some suggestions for redesigning the Nuprl logic and system to make them more amenable to metamathematical extensibility.

Chapter 2

Overview of Nuprl

2.1 Introduction

In this chapter we present an introduction to the Nuprl logic and system. A complete description of Nuprl is beyond the scope of what can be presented here. A number of important considerations and aspects of Nuprl will be, by necessity, omitted or given only cursory treatment. More information about Nuprl can be found in Constable et al. [41] and the works of Allen, Constable, Bates, Cleaveland, Knoblock, Mendler, Harper, Howe, Griffin, Sasaki, and Smith listed in the bibliography.

The chapter is organized as follows. We begin be giving an informal account of types as formalized by type theory. Next, we describe the terms and computation system underlying the elements of types. Next, we introduce the formal system of Nuprl followed by a description of the propositions–as–types principle that allows type theory to serve as a predicate logic. This is followed by a brief overview of the Nuprl system. We conclude the chapter by describing how Nuprl can serve as a functional programming language.

2.2 Nuprl Preliminaries

2.2.1 The Nuprl Types

The logic of Nuprl is a constructive type theory which is similar to one proposed by Martin-Löf [81]. Unlike most logics where the basic judgements are about the truth of a proposition, the basic judgements of Nuprl are about the inhabitation of a type. The base types of the logic are the integers, atoms and an empty type. Complex types including unions, products, function types, set types, and recursive types can be built from the base types using type constructors. The following is an informal description of the intended semantics of the types of the logic.

The integer type is denoted ***int***. The intended elements of this type are the integer numerals, $0, 1, -1, 2, -2, \ldots$. The atom type is denoted ***atom***. The intended elements are strings and are written "α", where α is a finite string of character symbols. The type ***void*** is the empty type. These three types, ***int***, ***atom***, and ***void***, are the *base types* of the logic.

Type *type constructors* of the logic include the *union type* which is denoted $A \mid B$ where A and B are types. This type represents the disjoint union of types A and B. The elements of a union type, $A \mid B$, are of the form $inl(a)$ and $inr(b)$ where a and b are terms and $a \in A$ and $b \in B$.

A *product type* is denoted $x{:}A \times B$ where x is an identifier and A is a type and under the assumption that $x \in A$, then B is a type. Note that x is bound to A and not to $A \times B$. The elements of a product type, $x{:}A \times B$, are pairs of terms, $\langle a, b \rangle$, where $a \in A$ and $b \in B[a/x]$[1]. This type is often called the *sigma type* in the literature and denoted $\Sigma x{:}A.B$.

A *set type* is denoted $\{x{:}A \mid B\}$ where x is an identifier, A is a type and under the assumption that $x \in A$, then B is a type. The elements of the set type $\{x{:}A \mid B\}$ are the terms $a \in A$ such that the type $B[x/a]$ is non-empty.

A *function type* is denoted $x{:}A \rightarrow B$ where x is an identifier, A is a type and under the assumption that $x \in A$, then B is a type. Note that x is bound to A and not to $A \rightarrow B$. The elements of the function type $x{:}A \rightarrow B$ are functions which are denoted $\lambda y.t$ where y is an identifier and t is term that satisfies the condition that for every term $a \in A$, $t[a/y] \in B$. This type is often called the *pi type* in the literature and written $\Pi x{:}A.B$.

A *list type* is denoted A ***list*** where A is a type. The elements of the list type A ***list*** are the term ***nil*** and the terms of the form $a.b$ where $a \in A$ and $b \in A$ ***list***. The dot operator, ".", should be read as *cons*.

A *recursive type* is denoted $rec(x.A)$ where x is an identifier and under the assumption that x is an element of an arbitrary type, then A is a type. The type $rec(x.A)$ is to be thought of as representing a certain solution to the equation

$$x = A.$$

The free occurrences of x in A are restricted so as to guarantee that a solution to the equation exists. These types represent recursive or inductively defined types. We call $A[rec(x.A)/x]$ the *expansion* of $rec(x.A)$. The inductive nature of these

[1] The notation $B[a/x]$ denotes the term formed by substituting a for all free occurrences of x in B, with alpha-conversion as necessary to avoid capture. The binding structure for terms is described below.

types is seen in the identification of $rec(x.A)$ and its expansion. Roughly, a term is an element of $rec(x.A)$ if and only if it is an element of the type formed by an iterated expansion of the type. Define a *finite unrolling* of $rec(x.A)$ as follows:

1. $rec(x.A)$ is a finite unrolling of $rec(x.A)$,

2. if R is a finite unrolling of $rec(x.A)$ and if T is a term with free variable z such that $R = T[rec(x.A)/z]$ then $T[A[rec(x.A)/x]/z]$ is a finite unrolling of $rec(x.A)$,

3. nothing else is a finite unrolling of $rec(x.A)$.

A term a is an element of $rec(x.A)$ if a is an element of a finite unrolling of $rec(x.A)$. In fact, there are recursive types that have elements of infinite unrollings.

An *equality type* is denoted $a = b \in A$ where a and b are terms and A is a type. The distinguished term *axiom* is in the equality type $a = b \in A$ if and only if $a =_A b$ where $=_A$ is the equality relation associated with type A. Although it may appear unnatural to describe equality as a type, it is consistent with the Nuprl treatment of all propositions as judgements about the inhabitation of types. Related to the equality type is a membership type, which is denoted $a \in A$ where a is a term and A is a type. In fact, the type $a \in A$ is just a notational abbreviation for the equality type $a = a \in A$.

The collections of types, called *universe types*, are denoted $U_1, U_2, U_3, \ldots$. Every Nuprl type is the member of some universe, and only types are elements of the universes. The universes form a cumulative hierarchy of types. Every base type is in U_1. Let $k \in \mathbf{N}$, $k > 0$ be given. All of the universes are closed under the type constructors; for example, if $A \in U_k$ and $B \in U_k$, then the union $A \mid B \in U_k$. Additionally, $U_k \in U_{k+1}$ and for every $A \in U_k$, $A \in U_{k+1}$. Judging that something is a type means judging that it inhabits some universe.

2.2.2 The Nuprl Terms

In Nuprl, unlike most logics, there is no distinction between terms and formulae; every expression, including those denoting types, are called terms. In addition to the terms representing the types and their elements that were summarized in the previous section, there are terms representing computations (programs).

The Nuprl terms are divided into two classes, *canonical* and *noncanonical*. Table 2.1 lists the Nuprl terms; those above the dotted line are canonical terms, and those bellow it are noncanonical terms. This table also defines when an identifier is bound in a subterm. All of the explicit identifiers in the terms are binding occurrences, and are bound in any subterms to the right of the occurrence that

Table 2.1: The Nuprl Terms

x	n	"α"	$void$
int	$atom$	$axiom$	nil
U_i	$inl(a)$	$inr(a)$	$A\ list$
$A \mid B$	$a < b$	$\langle a, b\rangle$	$a.b$
$\lambda x.\underset{x}{b}$	$x{:}A \times \underset{x}{B}$	$x{:}A \rightarrow \underset{x}{B}$	$\{x{:}A \mid \underset{x}{B}\}$
$a = b \in A$			

Canonical

....

Noncanonical

$\underset{\wedge}{a}+\underset{\wedge}{b}$	$any(a)$	$\underset{\wedge}{t}(\underset{\wedge}{a})$	$atom\_eq(\underset{\wedge}{a};\underset{\wedge}{b};s;t)$
$\underset{\wedge}{a}/\underset{\wedge}{b}$	$decide(\underset{\wedge}{a};x.\underset{x}{s};y.\underset{y}{t})$	$ind(\underset{\wedge}{a};x,y.\underset{\substack{x\\y}}{s};b;u,v.\underset{\substack{u\\v}}{t})$	$int\_eq(\underset{\wedge}{a};\underset{\wedge}{b};s;t)$
$\underset{\wedge}{a}*\underset{\wedge}{b}$	$less(\underset{\wedge}{a};\underset{\wedge}{b};s;t)$	$list\_ind(\underset{\wedge}{a};s;x,y,u.\underset{\substack{x\\y\\u}}{t})$	$spread(\underset{\wedge}{a};x,y.\underset{\substack{x\\y}}{t})$
$-\underset{\wedge}{a}$	$rec(x.\underset{x}{A})$	$rec\_ind(\underset{\wedge}{a};x,y.\underset{\substack{x\\y}}{t})$	$\underset{\wedge}{a}\ mod\ \underset{\wedge}{b}$
$\underset{\wedge}{a}-\underset{\wedge}{b}$			

x, y, u, v	range over identifiers	n	ranges over integer numerals
a, b, s, t, A, B	range over terms	i	rangers over positive natural numbers
α	ranges over character strings		

An identifier written below a subterm indicates that the identifier is bound in the subterm. Principle arguments are marked with "$\wedge$".

have the identifier below them. For example, in the term $ind(a; x, y.s; b; u, v.t)$, the identifiers represented by x, y, u, v are binding occurrences, and x and y are bound in the subterm represented by s, and u and v are bound in the subterm represented by t.

With this statement of the binding structure of the terms, we use the standard definitions for *free* and *bound* identifiers. A term without any free identifiers is said to be *closed.* Substitution in Nuprl terms may cause renaming of bound variables in order to avoid capture. Two terms are equal (as terms) if there is a renaming of the bound variables so that they are identical, that is they are *alpha-interconvertable* or *alpha-equal.*

The Nuprl identifiers are strings of the form i or $i@n$ where

$$i \in \{\_, a, b, c, \ldots, A, B, C, \ldots\}^* \setminus \{\epsilon\}$$

and $n \in \mathbb{N}$. The form $i@n$ occurs only through alpha-conversion to avoid capture during substitution. For example,

$$(\lambda f.f(x))[g(f)/x] = \lambda f@1.f@1(g(f)).$$

By convention, the underscore identifier, "_", is reserved and will not occur free in any terms.

To understand Nuprl as a programming logic, we must know how programs are executed, that is how terms are evaluated. Most of the noncanonical terms in table 2.1 have subterms marked with a caret. The carets mark the *principle arguments* of the terms. Tables 2.2 and 2.3 list the Nuprl redicies and corresponding contractum. If term t converts (or *computes* or *reduces*) to t' under the *head-first* (or *outermost*) reduction strategy, then we write

$$t \mapsto_h t',$$

and if t computes to t' under any reduction strategy, we write

$$t \mapsto t'.$$

The evaluation procedure for terms is the following.

1. If the term is canonical, then return the term.
2. If the term is a redex, return the result of evaluating the corresponding contractum.
3. If the term is noncanonical, replace the principle arguments with the result of evaluating them, and recursively evaluate the result.

Table 2.2: The Arithmetic Redicies

Redex	**Contractum**
$m + n$	$\mapsto_h$ The sum of m and n
$m - n$	$\mapsto_h$ The difference of m and n
$m * n$	$\mapsto_h$ The product of m and n
m/n	$\mapsto_h \begin{cases} 0 & \text{if } n = 0 \\ \text{The integer quotient of m and n} & \text{otherwise} \end{cases}$
$m \; mod \; n$	$\mapsto_h \begin{cases} 0 & \text{if } n = 0 \\ \text{The residual of } m \text{ and } n & \text{otherwise} \end{cases}$

where m, n range over integer numerals.

The term evaluation procedure can be summarized as "head compute the term until it is canonical". If a term does not contain any redicies, then we say it is *irreducible*. Note that canonical and irreducible are different. For example, the term

$$\langle(\lambda x.x)0, 1\rangle,$$

is canonical, but not irreducible.[2]

2.2.3 The Nuprl Logic

In this section we introduce the formal system that defines the Nuprl logic. The Nuprl logic is a *sequent calculus* (Gentzen [54]). Sequents form the basic unit for inference; it is sequents, not terms, that are proved. A *sequent* is a finite list of *declarations* $x_1{:}T_1, \ldots, x_k{:}T_k$ along with a term T where $x_1, \ldots, x_k$ are identifiers and $T_1, \ldots, T_k$ are terms. The notation for sequents is

$$x_1{:}T_1, \ldots, x_k{:}T_k \gg T.$$

[2] We could define a term as irreducible if it did not contain a redex in a legal tagging position of the term (see Constable et al. [41]). However, the legal tagging restrictions are unnatural and conjectured to be unnecessary. In this thesis, the extra generality of the definition is not exploited.

Table 2.3: The Non-Arithmetic Redicies

Redex	**Contractum**
$(\lambda x.b)(a)$	$\mapsto_h b[a/x]$
$spread(\langle a,b\rangle; x,y.t)$	$\mapsto_h t[a,b/x,y]$
$decide(inl(a); x.s; y.t)$	$\mapsto_h s[a/x]$
$decide(inr(b); x.s; y.t)$	$\mapsto_h t[b/y]$
$list\_ind(nil; s; x,y,u.t)$	$\mapsto_h s$
$list\_ind(a.b; s; x,y,u.t)$	$\mapsto_h t[a,b,list\_ind(b;s;x,y,u.t)/x,y,u]$
$atom\_eq(``\alpha"; ``\beta"; s; t)$	$\mapsto_h \begin{cases} s & \text{if } \alpha=\beta \\ t & \text{otherwise} \end{cases}$
$int\_eq(m;n;s;t)$	$\mapsto_h \begin{cases} s & \text{if } m=n \\ t & \text{otherwise} \end{cases}$
$less(m;n;s;t)$	$\mapsto_h \begin{cases} s & \text{if } m<n \\ t & \text{otherwise} \end{cases}$
$ind(m; x,y.s; b; u,v.t)$	$\mapsto_h \begin{cases} b \quad \text{if } m \text{ is } 0 \\ t[m, ind(m-1; x,y.s; b; u,v.t)/u,v] \\ \quad \text{if } m \text{ is positive} \\ s[m, ind(m+1; x,y.s; b; u,v.t)/x,y] \\ \quad \text{if } m \text{ is negative} \end{cases}$
$rec\_ind(c; x,y.t)$	$\mapsto_h t[c, \lambda z.rec\_ind(z; x,y.t)/x,y]$
$rec(x.a)$	$\mapsto_h a[rec(x.a)/x]$

x,y,u,v	range over identifiers	m,n	range over integer numerals
a,b	range over terms	c	ranges over canonical terms
α,β	range over character strings		

The part of a sequent to the left of the *turnstile*, $\gg$, is called the *declaration list* or *hypotheses* of the sequent, and the term to the right of the turnstile is called the *conclusion*. The sequent

$$x_1\!:\!T_1, \ldots, x_k\!:\!T_k \gg T$$

may be read as the assertion that if x_1 is of type T_1 and x_2 is of type T_2 and ... and x_k is of type T_k, then the type T is inhabited (non-empty). If a sequent is proved, we write it with a $\vdash$ in place of $\gg$, e.g.,

$$x_1\!:\!T_1, \ldots, x_k\!:\!T_k \vdash T.$$

To emphasize that a declaration list is empty in a sequent, $\gg T$, we write it as $\emptyset \gg T$ or $\emptyset \vdash T$. A sequent with an empty declaration list is said to be *closed*.

Proofs in Nuprl are developed in a refinement style where the goal is displayed with the subgoals corresponding to the rule beneath the goal. The presentation of the rules of inference of the logic mirrors this. For example, one of the rules for the product type is the following.

$$\begin{array}{l} H \gg \langle a, b\rangle \in x{:}A \times B \;\; \textit{by intro at } U_i \textit{ new } y \\ \quad \gg a \in A \\ \quad \gg b \in B[a/x] \\ \quad y{:}A \gg B[y/x] \in U_i \end{array}$$

In the presentation of this rule, H stands for an arbitrary declaration list and U_i stands for an arbitrary universe term. The goal of the rule is $H \gg \langle a, b\rangle \in x{:}A \times B$, the rule is designated by the phrase "*intro at* U_i *new* y", and the subgoals are

$$H \gg a \in A,$$

$$H \gg b \in B[a/x],$$

and

$$H, y{:}A \gg B[y/x] \in U_i.$$

Note that declarations in the subgoals that are inherited from the goal (H in this example) are elided in the presentation of the rule.

Due to the rich type structure of Nuprl it is not decidable whether a given notation denotes a type. Thus, one component of a derivation must be a proof that the syntactic notation of the goal does indeed denote a type, i.e., that the goal is *well-formed*. Rather than factoring the well-formedness proof out the proof of inhabitation, both occur simultaneously; embedded in the proof will be a proof that the goal is well-formed. At some point in a proof every syntactic notation for a type that occurs will be proved, either explictly or implictly, to inhabit a universe, thereby ensuring that it is well-formed. This is the reason for the third subgoal in the above rule. It is not necessary to have a explicit subgoal ensuring the well-formedness of A because it can be deduced from the first subgoal.

The "*at* U_i *new* y" phrase in the above rule of inference indicates the arguments or parameters to the rule. Nuprl has the *constructible subgoal property* meaning that from a goal and a rule (including arguments) alone, the subgoals can be constructed. In fact, in Nuprl the goal and rule functionally determine the subgoals. Thus, we can refer to the subgoals as the result of *refining* a goal by a rule. Besides the constructible subgoal property, the Nuprl logic has another important property. The Nuprl logic is a *refinement logic*, meaning that for any goal of a refinement, any proofs of the subgoals result in a proof of the goal. See Harper [63] for a detailed discussion of the definition and organization of type theories.

2.2.4 Extracting Programs from Proofs

Nuprl is a constructive type theory. We previously mentioned that the sequent

$$x_1{:}T_1,\ldots,x_k{:}T_k \gg T$$

may be read as the assertion that if x_1 is of type T_1 and x_2 is of type T_2 and ... and x_k is of type T_k, then the type T is inhabited. In fact, a stronger reading can be given, viz., if x_1 is an *object* of type T_1 and x_2 is an *object* type T_2 and ... and x_k is of type T_k, then an *object* of type T can be constructed. The rules of inference of the Nuprl logic specify how the inhabiting object of type T can be *extracted* from a proof of the sequent. The extract is a term but can be thought of as a program guaranteed to have the type of the conclusion of the proof. We shall see below that the Nuprl type theory is a very rich language. We can give specifications for programs using the types, and the extracts are then programs that have been proved to meet the specification.

The presentation of the rules of inference of the logic include indications for how the extract object is to be constructed from the extracts of the subgoals. For example, the following is a rule for the function type.

$$\begin{array}{l} H \gg x{:}A \to B \textit{ by intro at } U_i \textit{ new } y \; [\textit{ext } \lambda y.b] \\ \quad y{:}A \gg B[y/x] \; [\textit{ext } b] \\ \quad \gg A \in U_i \end{array}$$

The bracketed parts of the rule presentation marked "ext" specify how the extract is to be constructed. If b is the extract from the subproof for the first subgoal, then $\lambda y.b$ is the extract for the goal. Appendix A contains the version of the Nuprl rules that are used in this thesis. Many of the rules, the previous rule for product for example, have the constant extraction *axiom*. Only the rules with non-trivial extractions have the extraction explictly listed in the rule presentation.

The extract of a sequent can be automatically computed from a proof of the sequent. The user can control what object is extracted implicitly by knowing how each rule of inference in the proof affects the term extracted. When a particular extract is desired, the user can also control the extraction explicitly using the explicit intro rule.

$$\begin{array}{l} H \gg T \text{ by explicit intro } t \ [ext\ t] \\ \quad \gg t \in T \end{array}$$

In a few instances, we will want to assert that a sequent $H \gg T$ has a proof that results in a particular extract t. We will use the following notation for this.

$$H \vdash T \ [ext\ t]$$

For more information about extracting programs from proofs, see Bates [11] and Sasaki [104].

2.2.5 The Correctness of Nuprl

The semantics of Nuprl-like type theories is the subject of two recent studies, Allen [5] and Mendler [89]. While neither directly studied the Nuprl logic, the logics studied are very similar and the results will apply *mutatis mutandis* to the Nuprl logic. In this section, we give a statement of the correctness of Nuprl.

The usual correctness criteria for logics do no apply directly to the Nuprl logic since it is a type theory. The correctness of Nuprl involves the correctness of the rules of inference, the computation system, and the extraction mechanism. In order to state the correctness of Nuprl, we first identify the *canonical types*, and the *appropriate* terms that can inhabit each canonical type.

Let x and y be an identifiers and A, B, a, and b be terms. The *canonical types* are the terms of the form atom, int, void, $A \mid B$, $x{:}A \times B$, $x{:}A \rightarrow B$, $a = b \in A$, U_i for i a positive natural number, and $\{x{:}C \mid A\}$ where C is a canonical type.

For a given canonical type, C, we can define the *appropriate terms for* C. The atom constants "α", where α is a non-empty finite string of characters are the appropriate terms for *atom*. The integer numerals $0, 1, -1, \ldots$ are the appropriate terms for *int*. The terms of the form $inl(a)$ and $inr(a)$ are the appropriate terms for $A \mid B$. Terms of the form $\langle a, b\rangle$ are the appropriate terms for $x{:}A \times B$. Terms of the form $\lambda y.a$ are the appropriate terms for $x{:}A \rightarrow B$. The term *axiom* is the only appropriate term for $a = b \in A$. The appropriate terms for $\{x{:}C \mid A\}$ are the appropriate terms for the canonical type C. Terms of the form *atom*, *int*, *void*, $A \mid B$, $x{:}A \times B$, $x{:}A \to B$, $a = b \in A$, $\{x{:}A \mid B\}$, and U_j are the appropriate terms for U_i, where $j < i$.

Using these definitions, we can give a statement of the correctness of Nuprl. There are stronger semantic correctness conditions, but the following will suffice in this context.

Theorem 2.1 **(Correctness of Nuprl)**
If

$$\emptyset \vdash t \in T$$

or

$$\emptyset \vdash T\ [ext\ t],$$

then $t \mapsto_h \hat{t}$ and $T \mapsto_h \widehat{T}$ where $\widehat{T}$ is a canonical type and $\hat{t}$ is a canonical term of appropriate type for $\widehat{T}$. □

2.3 Logic in Type Theory

A constructive predicate calculus can be embedded in the Nuprl type theory using the *propositions-as-types* principle (see Curry [44], Scott [108], Howard [66], and de Bruijn [48]). Under this principle each formula, F, of the predicate calculus corresponds to a type, T, of the Nuprl type theory. The formula F is valid if and only if T is inhabited. The correspondence between formulae and types is defined inductively on the structure of formulae. The proposition *false* corresponds to the type *void*. The proposition *true* corresponds to the type *int* because *int* is trivially inhabited.[3]

If A and B are types (thought of as representing propositions), then A & B is represented by the product $\_{:}A \times B$ where by convention $\_$ is not free in B. The proposition A & B is valid if and only if A is valid and B is valid. Under the propositions-as-types principle A is valid and B is valid if and only if $a \in A$ and $b \in B$ for some a and b. Finally, this is equivalent to $\langle a, b\rangle \in \_{:}A \times B$. To summarize, the proposition A & B is valid if and only if the type $\_{:}A \times B$ is inhabited.

The proposition $A \vee B$ is represented by the union type, $A \mid B$. The proposition $A \vee B$ is valid (constructively) if and only A is valid or B is valid and an indication of which is valid is supplied. The type $A \mid B$ is inhabited by $inl(a)$ if $a \in A$ or by $inr(b)$ if $b \in B$. The implication $A \Rightarrow B$ is represented by the function type

[3] What could be more *true* than the integers?

Table 2.4: The Propositions-as-Types Principle

Proposition	**Corresponding Type**
false	*void*
true	*int*
$\neg A$	$\_:A \rightarrow void$
$A \vee B$	$A \mid B$
$A \,\&\, B$	$\_:A \times B$
$A \Rightarrow B$	$\_:A \rightarrow B$
$\exists x{:}A.B$	$x{:}A \times B$
$\forall x{:}A.B$	$x{:}A \rightarrow B$

$\_:A \rightarrow B$. The proposition A implies B is valid if and only if the validity of A implies the validity of B, whereas an element of $\_:A \rightarrow B$ is a function which maps inhabiting objects of type A to inhabiting objects of type B. The proposition $\neg A$ is interpreted as the proposition $A \Rightarrow false$, i.e., as the type $\_:A \rightarrow void$.

The proposition $\exists x{:}A.B$ corresponds to the type $x{:}A \times B$. Note that A in the proposition $\exists x{:}A.B$ is thought of as a type. The proposition $\exists x{:}A.B$ is valid if there exists $a \in A$ such that $B[a/x]$ is valid. Under the propositions-as-types principle, this is true if and only if there exists $a \in A$ such that $B[a/x]$ is inhabited, which is equivalent to the type $x{:}A \times B$ being inhabited. Finally, the proposition $\forall x{:}A.B$ is represented as the type $x{:}A \rightarrow B$. The proposition $\forall x{:}A.B$ is valid if and only if for every $a \in A$, $B[a/x]$ is valid, whereas the type $x{:}A \rightarrow B$ is inhabited if there is a mapping which maps every $a \in A$ to inhabiting objects of $B[a/x]$. Table 2.4 summarizes the embedding of propositions in type theory.

Note that the correctness of Nuprl implies the consistency of the embedded predicate calculus. Suppose that $\vdash false$ were proved. Then by the correctness of Nuprl, a term t can be extracted from the proof and t computes to an appropriate canonical term for *void*. But, there are no appropriate terms of type *void*. Therefore, $\nvdash false$.

2.4 The Nuprl System

In this section, we present an overview of the Nuprl system for proof development. The major components of the system include a command processor, a proof editor, an extractor for extracting programs from proofs, an evaluator for executing programs, a library mechanism for maintaining theories, a procedural metalanguage

```
Edit Theorem
```
x:*int*,
y:*int*,
c:$x < y \vee \neg(x < y)$
$\gg \exists m{:}int.(x \leq m \;\&\; y \leq m \;\&$
$(m = x \in int \vee m = y \in int))$
by < *refinement* >

Figure 2.1: Proof Editor Display of an Unrefined Proof Node

Edit Theorem
x:*int*,
y:*int*,
c:$x < y \vee \neg(x < y)$
$\gg \exists m{:}int.(x \leq m \;\&\; y \leq m \;\&$
$(m = x \in int \vee m = y \in int))$
by < *refinement* >

Edit Rule
elim c new a, a, w

Figure 2.2: The Refinement Rule is Entered

interface, and a syntactic definition facility.

The Nuprl system uses a window system that allows the various components of the system have displays on the screen simultaneously. The command processor includes commands to manipulate the library and to create and edit objects: proofs, syntactic definitions, extraction terms, and procedural tactics. The command processor has additional commands to invoke the extractor and evaluator.

Proof editing is performed with a special editor, a *refinement editor*. This editor presents proofs as trees with just one node of a proof displayed at a time. The user can navigate through a proof using the mouse or key board commands. Incomplete proofs can be extended by *refining* an incomplete node by a rule of inference. Figures 2.1–2.3 show three steps in editing a sample proof. In figure 2.1 the editor is positioned at the leaf of an incomplete proof. The theorem being proved is the specification for an integer maximum function:

Edit Theorem

x:*int*,
y:*int*,
c:$x < y \vee \neg(x < y)$
$\gg \exists m$:*int*.$(x \leq m$ & $y \leq m$ &
  $(m = x \in$ *int* $\vee$ $m = y \in$ *int*$))$
  by elim c new a, a, w

  a:$x < y$
  w:$(c =$ *inl*$(a) \in x < y \vee \neg(x < y))$
  $\gg \exists m$:*int*.$(x \leq m$ & $y \leq m$ &
    $(m = x \in$ *int* $\vee$ $m = y \in$ *int*$))$

  a:$\neg(x < y)$
  w:$(c =$ *inr*$(a) \in x < y \vee \neg(x < y))$
  $\gg \exists m$:*int*.$(x \leq m$ & $y \leq m$ &
    $(m = x \in$ *int* $\vee$ $m = y \in$ *int*$))$

Figure 2.3: The Result of Refining the Proof Node

$$\forall x{:}int.\forall y{:}int.\exists m{:}int.(x \le m \;\&\; y \le m \;\&\; (m = x \in int \lor m = y \in int)).$$

In figure 2.2, the user is has just entered an indication of what refinement rule is to be used. The rule name "elim" and the form of the indicated declaration, labeled by c, must be combined to determine which rule is intended; it is the seventh union rule in appendix A in this instance.

Figure 2.3 shows the result after the system has generated the subgoals resulting from refining the sequent by the rule. Note that the extraction clauses listed in the rules of inference are not displayed in the proof, and are used only in extracting a term from a complete proof.

The present Nuprl system provides a rudimentary mechanism for user-defined syntactic definitions, called *defs*. Using this mechanism the user can define syntactic abbreviations, macros, for commonly used notations. For example, the notations used in embedding the predicate calculus in type theory can be defined as defs.

Since we are proposing extensions in this thesis to the Nuprl system to provide for metamathematical extensibility, in most respects we have been faithful to the system as implemented. The currently implemented def mechanism has a number of major flaws, however, in both conception and implementation which make it inefficient and prevent certain kinds of notations from being defined. Many of the shortcomings of the current def mechanism are easily corrected. Since our work on metamathematical extensibility relies heavily upon syntactic notations, we have not felt bound in defining notations to stay within the constraints of the current def mechanism. More advanced mechanisms for syntactic abbreviations in Nuprl-like systems are being studied by Griffin [60] and Allen [3].

In this thesis we will define syntactic notations as in the following example.

Notation 2.1

$list\_sum(l) := list\_ind(l; 0; f, r, h.f + h)$
$\quad list\_sum{:}int\ list \rightarrow int$

The notation being defined is *list_sum*. The expression $list\_sum(x)$, for example, stands for $list\_ind(x; 0; f, r, h.f + h)$. Bound variables in the right hand side are renamed if necessary to avoid capture when the notation is instantiated. The second line of the above definition should be read as a comment; the intended "type" of this notation is to take an integer list argument and result in an integer value. Nuprl terms are actually untyped, and thus notations may not have a unique type, or any type at all. For example, *list_sum* also admits the type $\mathbf{N}\ list \rightarrow \mathbf{N}$. Table 2.5 contains a list of some common notations used in Nuprl. We shall also use the notations for embedding predicate calculus in type theory described in the previous section.

The notation A *cand* B represents the short-circuit conjunction of propositions A and B (all conjunctions are really of this form). The notations $A \times B$ and $A \rightarrow B$ represent product and function types where B is independent of the value of A. The notations $x > y$, $x \leq y$, $x \geq y$, and $x \neq y$ define the usual integer relations. The notations N and N^+ represent the natural numbers and positive natural numbers respectively. The notations x, y and $\langle x, y, z\rangle$ represent pairs and triples of values. The notations $fst(x)$ and $snd(x)$ are the first and second projections for pairs. The notations $isl(x)$ and $isr(x)$ represent predicates on type unions which indicate whether the value is in the left or right type of the union. The notations $outl(x)$ and $outr(x)$ are the corresponding projections. The notation $[a; b; c; \cdots]$ provides a nice notation for lists. The notation $l@g$ represents the appending of lists l and g. The notation $f \circ g$ represents function composition. The notation $f^{\circ k}$ represents iterated self-composition. The term represented by the notation $\| x \|$ is called the *squash* of x. It is used to hide the computational content, the inhabiting object, of x. The notation $A \upharpoonright P$ represents the restriction of type A by the predicate P. The notation $\{x{:}A \times y{:}B \mid P\}$ allows projections to be avoided in the predicate part of the set by labeling the components of the pair.

The Nuprl logic was designed as a logic to be implemented . The one node at a time view of proofs provided by the refinement editor and the system-generated subgoals based upon the arguments to rules make it convenient to edit proofs on a machine. However, writing and presenting Nuprl proofs on paper requires a different approach. A node by node proof display would be very inefficient on paper. We thus adopt an alternative syntax for displaying proofs on paper. Figure 2.4 contains a sample display of a proof fragment. In this display, the subgoals of a refinement are displayed below the goal at one additional level of indentation. For almost all refinements, the declaration lists of the subgoals are extensions of the declaration list of the goal. In these cases, the declarations inherited from the goal are not redisplayed in the subgoals; the declarations for a subgoal are any declarations for the goal and any declarations explicit in the subgoal. In the rare cases where the declarations of a subgoal are not an extension of the declarations of the goal, then the entire declaration list will be given for the subgoal.

As with programming in assembly language and linear algebra at the "*A-i-j*" level, it is possible but painful and uninteresting to prove theorems in Nuprl strictly at the primitive inference level. One quickly surpasses the level of abstraction. Raising the level of abstraction of proofs is, of course, a major theme of this work. Towards this end we employ meta-theorems to justify more abstract representa-

Table 2.5: Common Notations Used in Nuprl

$A\ cand\ B$	$:=$	$\_{:}A \times B$
$A \times B$	$:=$	$\_{:}A \times B$
$A \rightarrow B$	$:=$	$\_{:}A \rightarrow B$
$x > y$	$:=$	$y < x$
$x \leq y$	$:=$	$x < y \vee (x = y \in int)$
$x \geq y$	$:=$	$y \leq x$
$x \neq y$	$:=$	$\neg(x = y \in int)$
$\mathbf{N}$	$:=$	$\{n{:}int \mid 0 \leq n\}$
$\mathbf{N}^+$	$:=$	$\{n{:}int \mid 0 < n\}$
x, y	$:=$	$\langle x, y\rangle$
$\langle x, y, z\rangle$	$:=$	$\langle x, \langle y, z\rangle\rangle$
$fst(x)$	$:=$	$spread(x; a, b.a)$
$snd(x)$	$:=$	$spread(x; a, b.b)$
$isl(x)$	$:=$	$decide(x; a.true; b.false)$
$isr(x)$	$:=$	$decide(x; a.false; b.true)$
$outl(x)$	$:=$	$decide(x; a.a; b.void)$
$outr(x)$	$:=$	$decide(x; a.void; b.b)$
$[a; b; c; \cdots]$	$:=$	$a.b.c. \cdots .nil$
$l@g$	$:=$	$list\_ind(l; g; f, r, h.(f.h))$
$f{\circ}g$	$:=$	$\lambda x.f(g(x))$
$f^{\circ k}$	$:=$	$\begin{cases} f & \text{if } k = 1 \\ f{\circ}f^{\circ k-1} & \text{if } k > 1 \end{cases}$
$\lVert x \rVert$	$:=$	$\{\_{:}trivial \mid x\}$
$A \upharpoonright P$	$:=$	$\{x{:}A \mid P(x)\}$
$\{x{:}A \times y{:}B \mid P\}$	$:=$	$\{z{:}x{:}A \times B \mid P[fst(z), snd(z)/x, y]\}$

```
≫ ∀x:int.∀y:int.∃m:int.(x ≤ m & y ≤ m & (m = x ∈ int ∨ m = y ∈ int))
by intro at U1 new x
| x:int ≫ ∀y:int.∃m:int.(x ≤ m & y ≤ m & (m = x ∈ int∨m = y ∈ int))
| by intro at U1 new y
| | y:int ≫ ∃m:int.(x ≤ m & y ≤ m & (m = x ∈ int ∨ m = y ∈ int))
| | by seq x < y ∨ x ≥ y new c
| | | ≫ x < y ∨ ¬(x < y) by arith
| | | c:x < y ∨ ¬(x < y) ≫ ∃m:int.(x ≤ m & y ≤ m &
| | | (m = x ∈ int ∨ m = y ∈ int)) by elim c new a, a, w
| | | | a:x < y, w:(c = inl(a) ∈ x < y ∨ ¬(x < y))
| | | | ≫ ∃m:int.(x ≤ m & y ≤ m &
| | | | (m = x ∈ int ∨ m = y ∈ int)) by intro x at U1 new z
| | | | | ≫ x ∈ int by intro
| | | | | ≫ (x ≤ x & y ≤ x & (x = x ∈ int ∨ x = y ∈ int))
| | | | |   ⋮
| | | | | z:int ≫ (z ≤ m & y ≤ m &
| | | | | (m = z ∈ int ∨ m = y ∈ int)) ∈ U1
| | | | |   ⋮
| | | | a:¬(x < y), w:(c = inr(a) ∈ x < y ∨ ¬(x < y))
| | | | ≫ ∃m:int.(x ≤ m & y ≤ m &
| | | | (m = x ∈ int ∨ m = y ∈ int)) by intro y at U1new z
| | | | | ≫ y ∈ int by intro
| | | | | ≫ (x ≤ y & y ≤ y & (y = x ∈ int ∨ y = y ∈ int))
| | | | |   ⋮
| | | | | z:int ≫ (x ≤ m & z ≤ m &
| | | | | (m = x ∈ int ∨ m = z ∈ int)) ∈ U1
| | | | |   ⋮
| | y:int ≫ ∃m:int.(x ≤ m & y ≤ m &
| |   (m = x ∈ int ∨ m = y ∈ int)) ∈ U1
| |   ⋮
| x:int ≫ ∀y:int.∃m:int.(x ≤ m & y ≤ m &
|   (m = x ∈ int ∨ m = y ∈ int)) ∈ U1
|   ⋮
```

Figure 2.4: A Sample Display of a Proof Fragment

tions of proofs. In this section, derived rules of inference are used to facilitate the presentation of the following material. In chapter 4 we will explore writing tactics that implement derived rules for use during proof development with the Nuprl system.

Definition 2.2 A *derived rule of inference* is a list of sequents of the form

$$\begin{array}{l} H \gg G \\ \quad H_1 \gg G_1 \\ \qquad \vdots \\ \quad H_n \gg G_n \end{array}$$

such that from arbitrary proofs of $H_1 \gg G_1, \ldots, H_n \gg G_n$ a proof of $H \gg G$ can be constructed.

The following derived rule asserts that extra hypotheses can be ignored. The proof is trivial.

Proposition 2.3 The following is a derived rule of inference.

$$\begin{array}{l} H, x{:}T, H' \gg G \ \textit{by thinning } x \\ \quad H, H'' \gg G \end{array}$$

where H'' is the largest sublist of H' not containing x free and having all variables declared to the left, and all free variables of G are declared in H or H''. □

The next rule states that declared variables can be uniformly renamed.

Proposition 2.4 Let y be a variable that is not declared in declaration lists H and H'. The following is a derived rule of inference.

$$\begin{array}{l} H, x{:}T, H' \gg G \ \textit{by rename } x \textit{ to } y \\ \quad H, y{:}T, H'[y/x] \gg G[y/x] \end{array}$$

□

The next example derived rule provides a form of case analysis.

Proposition 2.5 The following is a derived rule of inference.

$$\begin{array}{l} H \gg G \ \textit{by cases } A \vee B \textit{ new } y \\ \quad \gg A \vee B \\ \quad y{:}A \gg G \\ \quad y{:}B \gg G \end{array}$$

$\gg \forall x{:}int.\forall y{:}int.\exists m{:}int.(x \le m \; \& \; y \le m \; \& \; (m = x \in int \vee m = y \in int))$
[intro at U_1 (twice)]
| $x{:}int, y{:}int \gg \exists m{:}int.(x \le m \; \& \; y \le m \; \& \; (m = x \in int \vee m = y \in int))$
| *[rule 2.5: cases $x < y \vee \neg(x < y)$]*
| | $\gg x < y \vee \neg(x < y)$ *by arith*
| | $x < y \gg \exists m{:}int.(x \le m \; \& \; y \le m \; \& \; (m = x \in int \vee m = y \in int))$
| | *by intro x at U_1 new z*
| | | $\gg x \in int$ *by intro*
| | | $\gg (x \le x \; \& \; y \le x \; \& \; (x = x \in int \vee x = y \in int))$ *[trivial]*
| | | $z{:}int \gg (z \le m \; \& \; y \le m \; \&$
| | | $(m = z \in int \vee m = y \in int)) \in U_1$ *[trivial]*
| | $\neg(x < y) \gg \exists m{:}int.(x \le m \; \& \; y \le m \; \& \; (m = x \in int \vee m = y \in int))$
| | *by intro y at U_1 new z*
| | | $\gg y \in int$ *by intro*
| | | $\gg (x \le y \; \& \; y \le y \; \& \; (y = x \in int \vee y = y \in int))$ *[trivial]*
| | | $z{:}int \gg (x \le m \; \& \; z \le m \; \&$
| | | $(m = x \in int \vee m = z \in int)) \in U_1$ *[trivial]*
| $x{:}int \gg \forall y{:}int.\exists m{:}int.(x \le m \; \& \; y \le m \; \&$
| $(m = x \in int \vee m = y \in int)) \in U_1$ *[trivial]*
| $x{:}int, y{:}int \gg \exists m{:}int.(x \le m \; \& \; y \le m \; \&$
| $(m = x \in int \vee m = y \in int)) \in U_1$ *[trivial]*

Figure 2.5: An Abstract Display of a Sample Proof

Proof
Let x and z be an identifiers not declared in H.

$H \gg G$ *by seq $A \vee B$ new x*
| $\gg A \vee B$ *[assumption]*
| $x{:}A \vee B \gg G$ *by elim x new y, y, z*
| | $y{:}A, z{:}(x = inl(y) \in A \vee B) \gg G$ *[thinning x]*
| | | $H, y{:}A \gg G$ *[assumption]*
| | $y{:}B, z{:}(x = inr(y) \in A \vee B) \gg G$ *[thinning x]*
| | | $H, y{:}B \gg G$ *[assumption]*

□

Even from the small proof fragment in figure 2.4, it is apparent that formal Nuprl proofs that are complete in every detail can be difficult to read. A major theme of this thesis is that the system can be used to conduct theorem proving at a reasonably abstract level. For the purpose of presenting proofs on paper in this thesis, we have adopted a number of conventions.

1. The arguments in rules allow the system to generate the subgoals from the goal. When the subgoals are present, as in a complete proof, the arguments are redundant. In most cases, we will abbreviate or omit the arguments in rules.

2. We will often omit the identifier of a declaration when it does not occur free in the hypothesis list or conclusion.

3. Abstract refinements and instances of derived rules will be indicated by enclosing a meta-rule in brackets in place of a refinement rule. For example, "[*trivial*]" will be used to mean that the subgoal has a simple proof, and "[*Rule 2.2*: a, b]" means that the derived rule proved in proposition 2.2 (or lemma 2.2 as appropriate) is to be used with a, b as arguments to refine the goal.

4. Occasionally, a subgoal with a trivial proof will be omitted entirely.

Note that these are only conventions for the display of proofs on paper. However, all can be mimicked in the present system and using the formal metalanguage proposed in this thesis. Figure 2.5 contains a sample proof employing these conventions.

2.5 Programming in Nuprl

The Nuprl term and computation system can be seen as a programming language, but like predicate calculus in Nuprl, it is necessary to code the standard programming constructs in the Nuprl language. Even then there are a number of noteworthy differences between standard programming languages and Nuprl programs.

Perhaps the most important of these is the way in which programs are constructed. In Nuprl, programs are not written in isolation, but extracted from proofs of types; the types serve as program specifications. There are two paradigms for this. One is to use the implicit forms of the Nuprl rules (those that have non-trivial extracts) and then extract the program from the complete proof of the type. The other is to explictly specify the program early in the proof of the type using the explicit introduction rule, and then prove that the program meets the specification. Generally, the former is preferable since it results in proofs that are more compact and abstract. It is also possible to program using a hybred of these paradigms where the specification type is refined using the implicit rules for part of the proof, and then pieces of the computation are explicitly introduced.

A second important difference between Nuprl programs and standard programs is that Nuprl programs always terminate when applied to arguments of the correct type. This stems from the fact that all Nuprl functions inhabiting a function type are total.[4]

Nuprl programs are functional and lazy; thus not all algorithms can be efficiently expressed directly in the language. Although this entails some overhead, there are a number of extraction and evaluation techniques that can ameliorate it. For example, the evaluation of the term $(\lambda x.g(x)(x))(\phi)$ first reduces to $g(\phi)(\phi)$. Although this appears to double the amount of computation necessary to evaluate ϕ, the implemented evaluation mechanism of Nuprl maintains the identity between the two instances of ϕ and thus only needs to compute it once. In general, arguments are computed at most once. Second, as noted by Goad [56,55], the fact that programs are supplied with proofs of their correctness means that there is more information than usual available that can be used in program optimization. Sasaki [104] is a study of how functional programs in a type theoretic language can, in some cases, be optimized to have imperative parts by analyzing the proof to determine how values are employed by the program. Finially, we note that in our experience with the procedural metalanguage, ML, the imperative aspects of the language, such as assignment, are almost never used in writing tactics.

Some programming language concepts have direct analogs in the Nuprl language. Programs are functions and sequencing is composition. Many of the arithmetic operators generally available in programming languages are built into the Nuprl language as are the list operations. Many other programming language concepts may be used in Nuprl via simple codings. The trivial one-element type can be defined as follows.

Notation 2.2

$trivial := \{x{:}int \mid x = 0 \in int\}$
  $trivial \in U_1$

The unique element of this type is 0. The two element type of boolean elements can be defined using a union type.

[4] Partial functions in constructive type theory are currently being researched by Constable and Smith [39].

Notation 2.3

$bool := trivial \mid trivial$
  $bool \in U_1$
$tt :\doteq inl(0)$
  $tt \in bool$
$f\!f := inr(0)$
  $f\!f \in bool$

A conditional branch on boolean values can be written using a decide term.

Notation 2.4

$if_{bool}\ b\ then\ s\ else\ t\ :=\ decide(b; \_s; \_t)$

With the propositions-as-types principle in mind, we refer to any element of U_1 as a *proposition* and any function with codomain U_1 as a *predicate*. In keeping with the propositions-as-types principle, a predicate results in an empty type to denote *false*, and a non-empty type to denote *true*.

It is not possible, in general, to use a predicate as the test for a conditional branch. The difficulty is that the Nuprl language is rich enough to express predicates which cannot be mechanically decided. For example, the predicate

$$\lambda f{:}int \rightarrow int.\exists x{:}int.(f(x) = 0 \in int),$$

is not decidable; a decision procedure for this would imply a solution to Hilbert's tenth problem. We can, however, identify a class of predicates which may serve as the tests for conditionals.

A proposition, $a \in U_1$, is said to be *decidable* if $\vdash a \vee \neg a$. Since Nuprl is constructive, this is not provable for all propositions. Any boolean combination of decidable propositions is decidable.

Proposition 2.6 If $a, b \in U_1$ are decidable propositions, then so are

$$\neg a,$$
$$a \vee b,$$
$$a \,\&\, b,$$
$$a \Rightarrow b.$$

□

A predicate $P{:}A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_k \rightarrow U_1$ is said to be *decidable* (over $A_1, A_2, \ldots, A_k$) if

$$\vdash \forall a_1{:}A_1.\forall a_2{:}A_2.\cdots\forall a_k{:}A_k.(P(a_1,\ldots,a_k) \vee \neg P(a_1,\ldots,a_k)).$$

Boolean combinations of decidable predicates are also decidable.

Proposition 2.7 Let A be given. If $P, Q \in A \rightarrow U_1$, are decidable predicates over A, then so are

$$\begin{array}{l} \lambda a.\neg P(a), \\ \lambda a.P(a) \vee Q(a), \\ \lambda a.P(a) \;\&\; Q(a), \\ \lambda a.P(a) \Rightarrow Q(a). \end{array}$$

□

Equality does not form a decidable predicate for all Nuprl types; however, it is decidable for *int* and *atom*.

Proposition 2.8 The following are decidable predicates.

$$\begin{array}{l} \lambda x.\lambda y.(x = y \in int) \\ \lambda x.\lambda y.(x = y \in atom) \\ \lambda x.\lambda y.(x < y) \end{array}$$

□

Let A be given and let $P \in A \rightarrow U_1$ be a decidable predicate over A. Because P is decidable, there is a term that *decides* it:

$$\exists d \in (\forall a{:}A.(P(a) \vee \neg P(a))),$$

that is

$$\exists d \in (a{:}A \rightarrow (P(a) \mid \neg P(a))).$$

Thus, the intended meaning of

$$if\ P(a)\ then\ s\ else\ t,$$

is captured by the Nuprl term

$$decide(d(a);\_s;\_t).$$

The boolean *if* given above can be viewed as a simplification of this more general *if* where the predicate and decision term have been combined. Unlike the case of the boolean *if*, we cannot give a notational definition of the general *if* because the decision term, d, is not explicit in the *if* statement. We shall, however, use the informal notation

$$if\ P(a)\ then\ s\ else\ t,$$

when the predicate P is known to be or obviously is decidable.

Recursion and iteration, as they occur in programming, are coded in Nuprl using *rec_ind*, *list_ind*, and *ind* terms.

The following notation provides a simple form of *destructuring* or *pattern-matching let binding*. It provides a convenient and compact notation for writing programs.

Notation 2.5

$let\ x = c\ in\ G := G[c/x]$
$let\ a = c\ and\ b = d\ in\ G :=$
$\quad let\ a = c\ in\ let\ b = d\ in\ G$
$let\ \langle a, b\rangle = c\ in\ G :=$
$\quad let\ a = fst(c)\ and\ b = snd(c)\ in\ G$
$let\ inl(a) = c\ in\ G :=$
$\quad let\ a = outl(c)\ in\ G$
$let\ inr(a) = c\ in\ G :=$
$\quad let\ a = outr(c)\ in\ G$

The final programming language concept that we examine is a *failure* or *escape* mechanism. It is intended to provide a way for functions to indicate *failure* on some arguments.

Notation 2.6

$A? := A \mid atom$
$a?b := decide(a; x.inl(x); \_.b)$
$failwith\ t := inr(t)$

Let A be a type. A potentially failing function is represented by a function with a codomain $A?$. A successful value is indicate by a term of the form $inl(a)$ for $a \in A$, and a failure is indicated by $failwith\ t$ for $t \in atom$. Let $a, b \in A?$ be given. Then $a?b \in A?$ and the expression $a?b$ is equal to a if a does not indicate failure, and b otherwise.

2.5.1 Derived rules for a Generalized Type Union

In this section we present a n-ary generalization of union type. This generalization will be used in proving a result in the next chapter. The reader may wish to defer reading this section until then.

For the remainder of this discussion, fix $n \in N, n \geq 2$ and let $\{T_i \mid 1 \leq i \leq n\}$ be an indexed family of terms. The type being defined is n-ary union $T_1 \mid \cdots \mid T_n$. The canonical terms for the type are a generalization of the *inl*, *inr* terms.

Notation 2.7

$$in_{n,j}(t) := \begin{cases} inl(t) & \text{if } j = 1 \\ inr^{\circ j-1} \circ inl(t) & \text{if } 1 < j < n \\ inr^{\circ n}(t) & \text{if } j = n \end{cases}$$

The notation $decide_n$ represents a noncanonical form for this type.

Notation 2.8

$$decide_n(x; b.t_1, \ldots, t_n) := \begin{cases} decide(x; b.t_1; b.t_2) & \text{if } n = 2 \\ decide(x; b.t_1; y.decide_{n-1}(y; b.t_2, \ldots, t_n)) & \text{if } n > 2 \end{cases}$$

The following series of propositions demonstrate that for these definitions, we can provide meta-rules that are analogs to the usual rules of inference for the union type.

Proposition 2.9 The following is a derived rule of inference.

$$\begin{array}{l} H \gg T_1 \mid \cdots \mid T_n \in U_i \text{ by intro} \\ \quad \gg T_1 \in U_i \\ \quad\quad \vdots \\ \quad \gg T_n \in U_i \end{array}$$

□

Proposition 2.10 The following is a derived rule of inference.

$$\begin{array}{l} H \gg in_{n,j}(b) \in T_1 \mid \cdots \mid T_n \text{ by intro at } U_i \\ \quad \gg b \in T_j \\ \quad \gg T_1 \in U_i \\ \quad\quad \vdots \\ \quad \gg T_n \in U_i \end{array}$$

Proof

The proof is by induction on n. The base case follows directly from the union introduction rule. Assume the rule holds for $n, n > 2$. Let j be given, $1 \leq j \leq n$. If $j = 1$ then since $in_{n,1} = inl$, we have

$$\begin{array}{l} H \gg in_{n,j}(b) \in T_1 \mid \cdots \mid T_n \text{ by intro at } U_i \\ \mid \gg b \in T_1 \text{ [assumption]} \\ \mid \gg T_2 \mid \cdots \mid T_n \in U_i \text{ [trivial from assumptions]} \end{array}$$

If $j > 1$ then $in_{n,j} = inr \circ in_{n-1,j-1}$ and

$H \gg in_{n,j}(b) \in T_1 \mid \cdots \mid T_n$ *by intro at* U_i
$\mid \gg in_{n-1,j-1}(b) \in T_2 \mid \cdots \mid T_n$ *[induction hypothesis]*
$\mid \gg T_1 \in U_1$ *[assumption]*

□

Proposition 2.11 The following is a derived rule of inference.

$$
\begin{array}{l}
H, x{:}T_1 \mid \cdots \mid T_n, H' \gg G \textit{ by elim } x \textit{ new } b, w \\
\quad b{:}T_1, w{:}(x = in_{n,1}(b) \in T_1 \mid \cdots \mid T_n) \gg G[in_{n,1}(b)/x] \\
\qquad \vdots \\
\quad b{:}T_n, w{:}(x = in_{n,n}(b) \in T_1 \mid \cdots \mid T_n) \gg G[in_{n,n}(b)/x]
\end{array}
$$

Proof

By induction on n. The base case, $n = 2$, is trivial. Assume the rule holds for n, $n > 2$. Let y and u be a new identifiers not occurring in $x{:}T_1 \mid \cdots \mid T_{n+1} \gg G$.

$H, x{:}T_1 \mid \cdots \mid T_{n+1}, H' \gg G$ *by elim* x *new* b, y, u
$\mid b{:}T_1, u{:}(x = in_{n+1,1}(b) \in T_1 \mid \cdots \mid T_{n+1})$
$\mid \gg G[in_{n+1,1}(b)/x]$ *[assumption]*
$\mid y{:}T_2 \mid \cdots \mid T_{n+1}, u{:}(x = inr(y) \in T_1 \mid \cdots \mid T_{n+1})$
$\mid \mid \gg G[inr(y)/x]$ *[induction hypothesis on y]*
$\mid \mid \mid b{:}T_2, u{:}(y = in_{n,1}(b) \in T_2 \mid \cdots \mid T_{n+1}) \gg G[inr(y)/x][in_{n,1}(b)/y]$
$\vdots$
$\mid \mid \mid b{:}T_{n+1}, u{:}(y = in_{n,n}(b) \in T_2 \mid \cdots \mid T_{n+1}) \gg G[inr(y)/x][in_{n,n}(b)/y]$

By definition, for every $j \in \mathbf{N}$, $1 \le j \le n$, $inr \circ in_{n,j} = in_{n+1,j+1}$. So,

$$G[inr(y)/x][in_{n,j}(b)/y] = G[in_{n+1,j+1}(b)/x].$$

Thus, for each j,

$b{:}T_{j+1}, u{:}(y = in_{n,j}(b) \in T_2 \mid \cdots \mid T_{n+1}) \gg G[in_{n+1,j+1}(b)/x]$
| *by seq* $x = in_{n+1,j+1}(b) \in T_1 \mid \cdots \mid T_{n+1}$ *new* w
| $\gg x = in_{n+1,j+1}(b) \in T_1 \mid \cdots \mid T_{n+1}$ [*def of* $in_{n+1,j+1}$]
| ·| $\gg x = inr(in_{n,j}(b)) \in T_1 \mid \cdots \mid T_{n+1}$
| | *by subst* $in_{n,j}(b) = y \in T_2 \mid \cdots \mid T_{n+1}$
| | *over* $z.x = inr(z) \in T_1 \mid \cdots \mid T_{n+1}$ *at* U_1
| | | $\gg in_{n,j}(b) = y \in T_2 \mid \cdots \mid T_{n+1}$ *by equality*
| | | $\gg x = inr(y) \in T_1 \mid \cdots \mid T_n$ *by hyp*
| | | $z{:}T_2 \mid \cdots \mid T_{n+1} \gg (x = inr(z) \in T_1 \mid \cdots \mid T_{n+1}) \in U_i$
| | | | [*trivial from assumptions*]
| $w{:}(x = in_{n+1,j+1}(b) \in T_1 \mid \cdots \mid T_n) \gg G$ [*thinning and assumption*]

□

Proposition 2.12 The following is a derived rule of inference.

$$
\begin{array}{l}
\gg decide_n(e; b.t_1; \cdots; t_n) \in G[e/z] \textit{ by intro over } z.G \\
\quad \textit{using } T_1 \mid \cdots \mid T_n \textit{ new } b \\
\quad \gg e \in T_1 \mid \cdots \mid T_n \\
\quad b{:}T_1, e = in_{n,1}(b) \in T_1 \mid \cdots \mid T_n \gg t_1 \in G[in_{n,1}(b)/z] \\
\qquad \vdots \\
\quad b{:}T_n, e = in_{n,n}(b) \in T_1 \mid \cdots \mid T_n \gg t_n \in G[in_{n,n}(b)/z]
\end{array}
$$

Proof

Sequence in $y{:}T_1 \mid \cdots \mid T_n \times (y = e \in T_1 \mid \cdots \mid T_n)$ then eliminate it using the $decide_n$ term. The rule then follows from the previous proposition.

□

Chapter 3

A Formal Metalanguage

3.1 Introduction

In this chapter we define a logic that is a formal metalanguage for Nuprl. The logic of Nuprl is formal and has a rigorous definition of proof. Although rigorous, the definition is usually stated in an informal metalanguage: a combination of English and mathematics. Most logics could be used as a formal metalanguage for Nuprl via some form of Gödel numbering of the Nuprl proof syntax. To be useful as a tactic language, however, the metalanguage must be able to naturally and directly represent Nuprl proofs and computations on them. This chapter defines a formal logic, *Metaprl*, that directly encodes the Nuprl proof theory. In addition Metaprl, like Nuprl itself, is a programming logic, and thus suitable for expressing the algorithms involved in tactic writing.

Metaprl is not intended as a general metalanguage for Nuprl suitable for studying Nuprl model theory or proof-normalization arguments. The requirements for the logic are modest: that it represent just enough of the proof theory of Nuprl so that proof tactics are expressible.

The chapter is organized as follows. First, we present the logic of Metaprl. Next, we show that under a reasonable assumption, Metaprl is a conservative extension of Nuprl. From this it is deduced that Metaprl is consistent. Finally, we outline the argument that Metaprl faithfully and adequately represents Nuprl proofs.

3.2 Formalizing the Nuprl Proof Theory

The first step towards a formal metalanguage for Nuprl is to give rigorous definitions of the syntactic concepts of identifier, term, rule, refinement and proof for

Nuprl. These concepts, taken together, comprise the Nuprl proof theory.

Metaprl is defined by employing the Nuprl type theory as a logic and a programming logic, and adjoining types and predicates that represent the Nuprl proof theory. The language of Metaprl, like that of Nuprl, is defined implicitly by the rules of the logic.

By definition every rule of Nuprl is a rule of Metaprl. Additional rules are introduced that axiomatize the Nuprl proof theory. Appendix A contains the version of the Nuprl logic that is formalized by Metaprl. Appendix B contains the additional rules that define Metaprl.

To facilitate the development of Metaprl, various notational or syntactic abbreviations are introduced. Although it may appear more direct to define these as primitives of Metaprl, it will facilitate later development if the number of new primitive concepts in Metaprl is minimized. Thus, whenever it is convenient to define a new concept in terms of existing concepts, we will do so.

3.2.1 Identifiers

The first concept of the Nuprl proof theory to be formalized in Metaprl is identifiers. Nuprl identifiers are either of the form β or $\beta@n$ where

$$\beta \in \{a, b, c, \ldots, A, B, C, \ldots, \_\}^*,$$

and n is a non-zero natural number. These two forms can be coalesced to simplify formalization by accepting identifiers of the form $\beta@0$ and equating an identifier of the from β with $\beta@0$.

The type *ident* is included in Metaprl to represent Nuprl identifiers. The canonical elements of *ident* are of the form

$$\beta@n$$

where β should be thought of as representing a character string and c as representing a natural number. The noncanonical term for *ident* is $\mathit{id\_parts}(v)$ which computes to a pair consisting of a character string and a natural number. The rules of inference for the type *ident* are listed in appendix B.

3.2.2 Syntactic Terms

The next concept of the Nuprl proof theory to be formalized in Metaprl is syntactic terms. The Metaprl formalization of syntactic terms is at the level of abstract as opposed to concrete syntax. That is, at the level of the parse-tree structure. This is a useful abstraction for a formal metalanguage since, aside from *defs*[1], no real information is lost in the mapping from concrete to abstract syntax. It is a great simplification because details of the Nuprl term parsing method do not need to be formalized, details that are irrelevant to writing proof tactics.

The type $term^0$ is included in Metaprl to represent Nuprl terms. Table 3.1 contains names of the Nuprl terms. The canonical terms of the type $term^0$ are the Nuprl term kinds. For example

$$application\_term^0(b) \in term^0,$$

where

$$b \in term^0 \times term^0.$$

There is one canonical term in $term^0$ for each Nuprl term kind. The canonical terms for $term^0$ are listed in table 3.2.

The noncanonical term for this type is $term\_ind^0$ which represents structural induction on terms. The syntax for this term is

$$term\_ind^0(t; h, b.d_1; \cdots; d_{38})$$

where d_i is the induction clause for i^{th} canonical term for $term^0$, one clause per term kind. The implicit term induction rule is given in figure 3.1. The rules of inference for $term^0$ are listed in appendix B.

Auxiliary Term Functions

To be useful as a tactic metalanguage, Metaprl must include functions representing the common predicates and calculations on terms. In particular, it should include functions for substitution, α-equality, computation, and various discriminators and destructors on terms. Such functions could be included as primitive concepts in Metaprl, but it is preferable to keep the base logic of Metaprl simple, and define such functions as notational definitions.

The following notation allow discriminations depending upon the kind of a term. The result is the name of the term as described in table 3.1.

Notation 3.1

$term\_kind^0(t) := term\_ind^0(t; h, b.\text{"}addition\text{"}; \text{"}any\text{"}; \text{"}application\text{"}; \cdots)$
  $term\_kind^0 : term^0 \rightarrow atom$

[1] *Defs* are the primitive macro facility currently implemented for Nuprl.

Table 3.1: The Kinds of Nuprl Terms

1.	Addition	$a + b$
2.	Any	$any(a)$
3.	Application	$t(a)$
4.	Atom	"α"
5.	Atom type	$atom$
6.	Atom-eq	$atom\_eq(a; b; s; t)$
7.	Axiom	$axiom$
8.	Cons	$a.b$
9.	Decide	$decide(a; x.s; y.t)$
10.	Division	a/b
11.	Equality	$a = b \in A$
12.	Function type	$x{:}A \rightarrow B$
13.	In-left	$inl(a)$
14.	In-right	$inr(a)$
15.	Induction	$ind(a; x, y.s; b; u, v.t)$
16.	Int-eq	$int\_eq(a; b; s; t)$
17.	Integer	n
18.	Integer type	int
19.	Lambda	$\lambda x.b$
20.	Less	$less(a; b; s; t)$
21.	Less-than	$a < b$
22.	List induction	$list\_ind(a; s; x, y, u.t)$
23.	List type	$A\ list$
24.	Modulo	$a\ mod\ b$
25.	Multiplication	$a * b$
26.	Negation	$-a$
27.	Nil	nil
28.	Pair	$\langle a, b\rangle$
29.	Product type	$x{:}A \times B$
30.	Rec type	$rec(x.A)$
31.	Rec induction	$rec\_ind(a; x, y.t)$
32.	Set type	$\{x{:}A \mid B\}$
33.	Spread	$spread(a; x, y.t)$
34.	Subtraction	$a - b$
35.	Union type	$A \mid B$
36.	Universe	U_k
37.	Variable	x
38.	Void	$void$

Table 3.2: Canonical Terms for $term^0$

1.	$addition\_term^0(b)$	20.	$less\_term^0(b)$
2.	$any\_term^0(b)$	21.	$less\_than\_term^0(b)$
3.	$application\_term^0(b)$	22.	$list\_induction\_term^0(b)$
4.	$atom\_term^0(b)$	23.	$list\_type\_term^0(b)$
5.	$atom\_type\_term^0$	24.	$modulo\_term^0(b)$
6.	$atom\_eq\_term^0(b)$	25.	$multiplication\_term^0(b)$
7.	$axiom\_term^0$	26.	$negation\_term^0(b)$
8.	$cons\_term^0(b)$	27.	nil_term^0
9.	$decide\_term^0(b)$	28.	$pair\_term^0(b)$
10.	$division\_term^0(b)$	29.	$product\_type\_term^0(b)$
11.	$equality\_term^0(b)$	30.	$rec\_term^0(b)$
12.	$function\_type\_term^0(b)$	31.	$rec\_ind\_term^0(b)$
13.	$in\_left\_term^0(b)$	32.	$set\_type\_term^0(b)$
14.	$in\_right\_term^0(b)$	33.	$spread\_term^0(b)$
15.	$induction\_term^0(b)$	34.	$subtraction\_term^0(b)$
16.	$int\_eq\_term^0(b)$	35.	$union\_type\_term^0(b)$
17.	$integer\_term^0(b)$	36.	$universe\_term^0(b)$
18.	$integer\_type\_term^0$	37.	$variable\_term^0(b)$
19.	$lambda\_term^0(b)$	38.	$void\_term^0$

Also useful are predicates on terms that are true if the term has a particular kind, for example, $is\_addition^0(t)$ is defined to be true if and only if the term kind of t is "addition".

Notation 3.2

$is\_addition^0(t) := (term\_kind^0(t) = \text{“}addition\text{”} \in atom)$
$\quad is\_addition^0 : term^0 \rightarrow U_1$

Analogous predicates are defined for all the term kinds. The following notation computes the subparts of terms.

Notation 3.3

$destruct\_term^0(t) := rec\_ind(t; h, b.b; b; b; \cdots)$

Also useful are term destructors that select out particular parts of terms. An example is the following.

$$
\begin{array}{l}
H \gg term\_ind^0(t; h, b.d_1; \cdots; d_{38}) \in G[t/v] \\
\quad by \;\; intro \; over \; v.G \; new \; q, h', b' \\
\quad\; \gg t \in term^0 \\
\quad q{:}term^0 \rightarrow U_1, h'{:}v{:}term' \rightarrow G, b'{:}term' \times term' \\
\qquad \gg d_1[h', b'/h, b] \in G[addition\_term^0(b')/v] \\
\quad q{:}term^0 \rightarrow U_1, h'{:}v{:}term' \rightarrow G, b'{:}term' \\
\qquad \gg d_2[h', b'/h, b] \in G[any\_term^0(b')/v] \\
\quad q{:}te\cdots{}^0 \rightarrow U_1, h'{:}v{:}term' \rightarrow G, b'{:}term' \times term' \\
\qquad \cdots [h', b'/h, b] \in G[application\_term^0(b')/v] \\
\quad q{:}te\cdots \rightarrow U_1, h'{:}v{:}term' \rightarrow G, b'{:}term', \\
\qquad \gg \cdots', b'/h, b] \in G[atom\_term^0(b')/v] \\
\qquad \gg a \cdots \in G[atom\_type\_term^0/v] \\
\qquad\quad \vdots \\
where \;\; term' := \{t{:}term \mid q(t)\}
\end{array}
$$

Figure 3.1: The Implicit Induction Rule for Terms

Notation 3.4

$fst\_of\_equality^0(t) := term\_ind^0(t; h, b. \cdots; fst(b); \cdots)$
$\quad fst\_of\_equality^0{:}term^0 \rightarrow term^0$

This notation will select the first subterm in an equality term, for example,

$$fst\_of\_equality^0(\ulcorner a = b \in A \urcorner) \mapsto a.$$

Table 3.3 summarizes some of these destructors by showing what they compute to on sample arguments.

The notation is $substitute^0(t, t', x)$ computes the result of substituting the term t' for identifier x in term t The notation $alpha\_equal^0(t, t')$ is a predicate on pairs of terms that is true if terms t and t' are α-equal. The notation $instance^0(t, v, t')$ is a predicate that is true if and only if term t is an instance of $v.t'$ where v is an identifier and t' is a term. The definitions for these notations are lengthy but straight-forward from the Metaprl primitive terms. The following notation is used to represent substitution.

Notation 3.5

$t[t'/x] := substitute^0(t, t', x)$

Table 3.3: Summary of Sample Term Destructors

$$
\begin{array}{ll}
fst\_of\_equality^0(\ulcorner a = b \in A \urcorner) & \mapsto a \\
snd\_of\_equality^0(\ulcorner a = b \in A \urcorner) & \mapsto b \\
type\_of\_equality^0(\ulcorner a = b \in A \urcorner) & \mapsto A \\
 & \\
identifer\_of\_product^0(\ulcorner x{:}A \times B \urcorner) & \mapsto x \\
left\_of\_product^0(\ulcorner x{:}A \times B \urcorner) & \mapsto A \\
right\_of\_product^0(\ulcorner x{:}A \times B \urcorner) & \mapsto B
\end{array}
$$

Recall that in Nuprl, a membership term $t \in T$ is an abbreviation for the equality term $t = t \in T$. Because many Nuprl refinement rules apply to membership and not general equalities, the following notation is introduced to allow uniform treatment of membership terms.

Notation 3.6

$$
\begin{array}{l}
membership\_term^0(t, T) := equality\_term^0(t, t, T) \\
\quad membership\_term^0 \in term^0 \to term^0 \to term^0 \\
is\_membership^0(t) := is\_equality^0(t) \ cand \\
\quad (fst\_of\_equality^0(t) = snd\_of\_equality^0(t) \in term^0) \\
is\_membership^0 \in term^0 \to U_1
\end{array}
$$

3.2.3 Sequents

Sequents are formalized in Metaprl without defining a new primitive type. A sequent is a list of bindings (pairs of identifiers and terms, along with a flag indicating whether the hypothesis is hidden) and a conclusion term.[2]

The notation $sequent^0$ is introduced to represent that type along with the destructors hyp^0 and $concl^0$ for the list of bindings and conclusion respectively. The following notations are required.

[2]Note that sequents are not required to have distinct declared variables, or to be closed. It is a property of proofs that all sequents occurring in a proof have distinct declared variables and have all free variables in the declarations and conclusion declared to the left of the occurrence.

Notation 3.7

$decl^0 := ident \times term^0 \times bool$
  $decl^0 \in U_1$
$sequent^0 := decl^0\ list \times term^0$
  $sequent^0 \in U_1$
$hyp^0 := fst$
  $hyp \in sequent^0 \to decl^0\ list$
$concl^0 := snd$
  $concl^0 \in sequent^0 \to term^0$
$id\_of\_decl^0(d) := fst(d)$
  $id\_of\_decl^0 \in decl^0 \to ident$
$type\_of\_decl^0(d) := fst(snd\ d)$
  $type\_of\_decl^0 \in decl^0 \to term^0$
$flag\_of\_decl^0(d) := snd(snd\ d)$
  $flag\_of\_decl^0 \in decl^0 \to bool$
$is\_hidden^0(d) := (flag\_of\_decl^0(d) = tt \in bool)$
  $is\_hidden^0 \in decl^0 \to U_1$

3.2.4 New Identifiers

Many refinement rules require that the identifiers introduced by the rule are not already declared in the in the sequent. The notation new^0 expresses the following predicate.

$$new^0(x, s) = new\_in\_hyp^0(x, hyp^0\ s)$$

where

$$new\_in\_hyp^0(x, l) = \begin{cases} true & \text{if } null\ l \\ x \neq id\_of\_decl^0(hd\ l)\ \& & \\ \quad new\_in\_hyp^0(x, tl\ l) & \text{otherwise} \end{cases}$$

Notation 3.8

$new^0(x, s) := list\_ind((hyp^0\ s); true;$
  $f, r, h.((id\_of\_decl^0(f) \neq x \in ident)\ \&\ h)$

Additionally, we will require an expression that represents the result of "extending" the list of hypotheses with a new binding. That is an expression that represents the following function.

$$extend\_hyp^0(l, d) = \begin{cases} [d] & \text{if } null\ l \\ (hd\ l).extend\_hyp^0(tl\ l, d) & \text{otherwise} \end{cases}$$

In addition to defining a term for $extend\_hyp^0$, we define terms for the two constant cases of the hidden flag.

Notation 3.9

$extend\_hyp^0(h, d) := list\_ind(h; [d]; f, r, h.(f.h))$
$extend^0(h, b) := extend\_hyp^0(h, (fst\ b,\ snd\ b,\ f\!f))$
$extend\_hidden^0(h, b) := extend\_hyp^0(h, (fst\ b,\ snd\ b,\ tt))$

The following notations provide a nice notation for extending declaration lists.

Notation 3.10

$h + d := extend^0(h, d)$
$h + x{:}y := h + \langle x, y\rangle$
$h + [d] := extend\_hidden^0(h, d)$

We also require notations for the following functions.

$distinct^0{:}ident\ list \rightarrow U_1$
: $distinct^0(l)$ is true if and only if all the elements of list l are distinct.

$fresh^0{:}ident \times decl^0\ list \rightarrow U_1$
: $fresh^0(x, h)$ is true if and only if x is not declared in declaration list h.

$declared^0{:}ident \times decl^0\ list \rightarrow U_1$
: $declared(x, h)$ is $\neg fresh^0(x, h)$.

$declaration^0{:}x{:}ident \rightarrow h{:}\{h{:}decl^0 list \mid declared^0(x, h)\} \rightarrow term^0$
: $declaration^0(x, h)$ is the declared term labeled by the identifier x in declaration list h.

3.2.5 Antiquotation

Although any Nuprl term can be represented in Metaprl using the previous forms, it is an unwieldy notation for all but the most simple terms. To alleviate this, we define a system of *antiquotation* similar to antiquotation in LCF and backquote in LISP that provides a more compact and readable notation for the representation of terms.

$$\gg universe\_intro\_function^0(a) \in rule^0 \ by \ intro$$
$$\gg a \in ident \times term^0$$

Figure 3.2: A Sample Introduction Rule for *rule⁰*.

Notation 3.11

$$\begin{array}{ll}
\text{“}\wedge v\text{”} & := v \\
\ulcorner t\ a \urcorner^0 & := application_term^0(\ulcorner t \urcorner^0, \ulcorner a \urcorner^0) \\
\ulcorner int \urcorner^0 & := int_term^0 \\
\ulcorner \lambda v.b \urcorner^0 & := lambda_term^0(\text{“}v\text{”}, \ulcorner b \urcorner^0) \\
\ulcorner x{:}a \times b \urcorner^0 & := product_term^0(\text{“}x\text{”}, \ulcorner a \urcorner^0, \ulcorner b \urcorner^0) \\
\ulcorner U_i \urcorner^0 & := universe_term^0(i) \\
 & \vdots \\
\ulcorner \wedge x \urcorner^0 & := x \\
\ulcorner t \in T \urcorner^0 & := membership_term^0(\ulcorner t \urcorner^0, \ulcorner T \urcorner^0)
\end{array}$$

The notation of antiquotation provides a familiar syntax for constant, and *nearly* constant terms. For example, if $x \in ident$ and $f(x) \in term^0$, then

$$\ulcorner \wedge x{:}int \times \wedge f(x) \urcorner = product\_type\_term^0(x, integer\_type\_term^0, f(x)).$$

3.2.6 Rules

The type $rule^0$ in Metaprl represents Nuprl refinement rules. In a formal representation of proofs, it is necessary for each rule to have a name. When editing Nuprl proofs, generic rule names such as "intro" and "elim" are employed; the context in which the rule is to be employed is used to make the intended rule unambiguous. In the metalanguage where a proof rule may exist outside the context of a proof, it is necessary to give unique names to each of the rules. In addition to representing a specific refinement rule, the elements of type $rule^0$ represent the arguments ("on", "over", "new", etc.) parts of rules.[3] There is a canonical term in $rule^0$ for each of the Nuprl refinements rules. A sample introduction rule for $rule^0$ is given in figure 3.2.

[3]The canonical terms of type $rule^0$ differ from the procedural metalanguage's representation of rules in that no semantic requirements are placed upon the arguments other than that they be of correct type. All of the semantic checks on arguments are combined with the check that a given rule is applicable to a given sequent.

1. $\mathit{applies}(s, \mathit{product\_equality}^0(d)) \mapsto_h$
   $\mathit{let}\ c, h = s\ \mathit{in}$
   $\mathit{is\_membership}^0(c)\ \mathit{cand}$
   $\mathit{is\_product}^0(\mathit{element\_of\_member}^0(c))\ \&$
   $\mathit{is\_universe}^0(\mathit{type\_of\_member}^0(c))\ \&$
   $\mathit{fresh}^0(d, h)$

2. $\mathit{subgoals}^0(s, \mathit{product\_equality}^0(d)) \mapsto_h$
   $\mathit{let}\ c, h = s\ \mathit{and}\ \ulcorner \wedge x{:}\wedge A \times \wedge B \in \wedge Ui \urcorner = c\ \mathit{in}$
   $[\langle h, \ulcorner \wedge A \in \wedge Ui \urcorner\rangle; \langle h + x{:}A, \ulcorner \wedge B[\wedge d / \wedge x] \in \wedge Ui \urcorner\rangle]$

Figure 3.3: Sample Formalization of a Nuprl Refinement Rule

The noncanonical term for rule^0, $\mathit{rule\_case}^0$, allows discrimination on the rule kind and the arguments of the rule to be computed. The rules of inference for rule^0 are given in appendix B.

3.2.7 Refinement

The next step in formalizing the Nuprl proof theory is to formalize refinement by the rules of inference of the Nuprl logic. This is a relation between sequents, rules, and lists of sequents, interpreted as a goal, rule, and subgoals respectively. In a refinement logic, such as Nuprl, this relation is functional: A goal and rule determine the subgoals (assuming the rule applies to the sequent). This functionality can be exploited by specifying a function $\mathit{subgoals}^0$ which has the abstract type

$$\{s{:}\mathit{sequent}^0 \times r{:}\mathit{rule}^0 \mid \mathit{applies}^0(s, r)\} \rightarrow \mathit{sequent}^0\ \mathit{list}$$

where $\mathit{applies}^0$ is a predicate which specifies when a rule is applicable to a sequent. Rather than formalizing $\mathit{subgoals}^0$ and $\mathit{applies}^0$ as functions, they are defined as noncanonical elements of the types $\mathit{sequent}^0\ \mathit{list}$ and U_1 respectively. This is analogous to other function-like primitives ($\mathit{decide}, \mathit{spread}, \mathit{ind}$, etc.), and allows reasoning about the computation performed by $\mathit{subgoals}^0$. Figure 3.3 contains the clauses for $\mathit{applies}^0$ and $\mathit{subgoals}^0$ for a sample rule, the rule numbered product 2 in appendix A.

The definition of the Nuprl logic in **Implementing Mathematics with the Nuprl Proof Development System** [41] contains two inference rules for the application of known theorems: the *lemma* and the the *def* rules (see figure 3.4). There are three reasons to omit these rules from a logic endowed with a formal metalanguage. First, these rules were incorporated as *inference* rules to provide a

$$\begin{array}{l} H \gg T \; by \; lemma \; theorem \; new \; x \; [ext \; t[term\_of(theorem)/x]] \\ \quad x{:}C \gg T \; [ext \; t] \end{array}$$

where C is the conclusion of the complete theorem *theorem*.

$$\begin{array}{l} H \gg T \; by \; def \; theorem \; new \; x \; [ext \; t[axiom/x]] \\ \quad x{:}term\_of(theorem) = ext\text{-}term \in C \gg T \; [ext \; t] \end{array}$$

where C is the conclusion of the complete theorem, *theorem*, and *ext-term* is the term extracted from that theorem.

Figure 3.4: The Lemma Rules of Nuprl.

connection between proofs under construction and known proofs. However, even without rules stating these connections, the language of a formal metalanguage can express the use of lemmas. That is, the action of employing a lemma can be stated as a Metaprl rule. A more pragmatic reason for not including the lemma rules, is that formalization of the these rules would force mutual recursive definitions of $term^0$, $rule^0$ and $proof^0$. Without the lemma rules, simpler definitions of the types for terms, rules and proofs can be given. Finally, an adequate treatment of the lemma rules would require a formalization of the Nuprl library structure. We have tried to avoid doing this because the present library structure is primitive and likely to be replaced in future version of the system. For these reasons, we have omitted the lemma and def rules from the version of Nuprl present in appendix A. It should become clear to the reader as we proceed how the effect of these rules can be produced using Metaprl.

Most of the remaining inference rules of the Nuprl logic follow the pattern of rules given in figure 3.3 fairly directly. They have a simply expressed applicability predicate and connect the goals and subgoals via a simple syntactic calculation. The exceptions are the direct computation rules and the decision procedures, *equality* and *arith*.

The formalization of the direct computation rules requires some further machinery. The axiomatization of the the rules requires encoding the *legal-tagging* restriction. The rules for the $applies^0$ type encode a check that the tagged term with the tags erased matches the term to be reduced. The subgoal calculation incorporates the (syntactic) term reduction algorithm, where the reduction algorithm is expressed as a function from $term^0$ to $term^0$. We have not included a formal treatment of legal tagging and direct computation because it is conjectured that the legal-tagging restriction on direct computation is unnecessary and will be omitted in future versions of the Nuprl logic. In any event, it should be clear that legal tagging and direct computation can be formalized with the following caveat.

In **Implementing Mathematics with the Nuprl Proof Development System** [41], an expression of the form $[*; t]$ is considered a legal tagging of term t. Computing with a tag of this form means that reduction is to take place until there are no further redices. Since this reduction sequence need not terminate, admitting such taggings as part of the inference structure of the logic would make recognizing proofs not recursive. Therefore, taggings of this form are considered meta-rules implemented by the system, and not part of the definition of the Nuprl logic. Like the lemma rules, this is an instance where a meta-rule was incorporated into the implementation of the logic as a convenience.

The two Nuprl inference rules presented as decision procedures, *arith* and *equality*, differ from the other inference rules of the Nuprl logic in that they do not have compactly expressible applicability predicates; the algorithms encoding the decision procedures are the test of applicability. Thus, in the formalization of the rules, the $\mathit{applies}^0$ predicate has to encode the decision procedure. This is accomplished by simulating the decision procedure on the Metaprl encoding of the sequent.

The computation clauses for the equality rule, for example, encode the decision procedure:

$$\mathit{applies}^0(\bar{s}, \mathit{equality}) \mapsto_h \mathit{true}$$

if $\bar{s}$ is irreducible and the equality decision procedure simulated on $\bar{s}$ accepts, and

$$\mathit{applies}^0(\bar{s}, \mathit{equality}) \mapsto_h \mathit{false}$$

if $\bar{s}$ is irreducible and the equality decision procedure simulated on $\bar{s}$ rejects.

Note that if $\bar{s}$ is reducible, then $\mathit{applies}(\bar{s}, \mathit{equality})$ is not a redex. This is in contrast to most computation rules where the principle arguments of the term only need be canonical. The subgoal computation clause for the equality rule is simply

$$\mathit{subgoals}^0(\bar{s}, \mathit{equality}) \mapsto_h \mathit{nil}.$$

The formalization of the arith rule in Metaprl is treated similarly to the equality rule, with the slight additional complication that well-formedness subgoals are generated by arith on some sequents.

$$
\begin{aligned}
H &\gg \mathit{applies}^0(s,r) \vee \neg \mathit{applies}^0(s,r) \text{ by } \mathit{decidable}\\
&\gg s \in \mathit{sequent}^0\\
&\gg r \in \mathit{rule}^0
\end{aligned}
$$

Figure 3.5: Applies Represents a Decidable Predicate

This treatment of decision procedures in not completely adequate. All of the other inference rules of Nuprl are encoded in Metaprl in an abstract way. The decision procedures are completely concrete — it can only be determined for irreducible sequents whether a decision procedure will apply. We cannot, for example, directly prove that arith will apply to any sequent of the form

$$0 = 1 \in \mathit{int} \gg t,$$

where t is an arbitrary term. In chapter 7, we discuss an alternative treatment for decision-procedure like rules of inference.

Finally, there is a rule that states that $\mathit{applies}^0$ is a decidable predicate (figure 3.5). The extraction from this rule is a procedure called $\mathit{decide\_applies}^0$ that computes the decision.

3.2.8 Proofs

The final Nuprl concept to be formalized in Metaprl is that of proof.[4] Nuprl proofs are trees of refinements; the nodes are of type $\mathit{sequent}^0 \times \mathit{rule}^0$ and satisfy the following condition. Given an arbitrary node $\langle s, r\rangle$ of a proof tree, $\mathit{subgoals}^0(s,r) = g$ where g is the list of sequents of the children of the given node.

The expression $\mathit{goals}(c)$ for $c \in \mathit{proof}^0\ \mathit{list}$ represents the goal sequents for each proof in list c.

$$\mathit{goals}(c) = \begin{cases} \mathit{nil} & \text{if } \mathit{null}\ c\\ (\mathit{fst}\ (\mathit{hd}\ c)).\mathit{goals}(\mathit{tl}\ c) & \text{otherwise} \end{cases}$$

Notation 3.12

$\mathit{goals}(c) := \mathit{list\_ind}(c; \mathit{nil}; f, r, h.((\mathit{fst}\ f).h))$
  $\mathit{goals} \in \mathit{proof}^0\ \mathit{list} \rightarrow \mathit{sequent}^0\ \mathit{list}$

[4]Note that it is complete proofs that are formalized in Metaprl in contrast to partial proofs which are the main data type of the current procedural metalanguage.

$$
\begin{array}{l}
H \gg \langle s,r,c\rangle \in proof^0 \text{ by intro} \\
\quad \gg s \in sequent^0 \\
\quad \gg r \in rule^0 \\
\quad \gg c \in proof^0 \text{ list} \\
\quad \gg applies^0(s,r) \\
\quad \gg subgoals^0(s,r) = goals(c) \in sequent^0 \text{ list}
\end{array}
$$

Figure 3.6: The Introduction Rule for $proof^0$.

Figure 3.6 contains the explicit introduction rule for proofs. The noncanonical term, $proof\_ind^0$, for the type $proof^0$ represents structural induction on proof trees.

This concludes the definition of the logic of Metaprl. In the next two sections, we discuss the correctness of Metaprl. First, we look at its consistency. Next, we argue that it faithfully and adequately represents Nuprl proofs.

3.3 The Consistency of Metaprl

In this section we present an outline of a proof that Metaprl is consistent. We require the following definitions. Let H and H' be formal systems with

$$L(H) \subset L(H').$$

The formal system H' is said to be a *conservative extension of* H if and only if for every formula t of H, if

$$\vdash^{H'} t,$$

then

$$\vdash^{H} t.$$

Let *Metaprl*$^-$ be the formal system formed by omitting from Metaprl all references to the decisions procedures, *arith* and *equality*, from the $rule^0$, $subgoals^0$, and $applies^0$ types. We prove that *Metaprl*$^-$ is a conservative extension of Nuprl. A corollary of this is that *Metaprl*$^-$ is consistent. For if *void* were provable in *Metaprl*$^-$, then $\vdash^0$ *void*. But Nuprl is consistent, and therefore *Metaprl*$^-$ is consistent. We will use "$\vdash^{1'}$" and "$\gg^{1'}$" as the turnstiles for *Metaprl*$^-$.

The proof that *Metaprl*$^-$ is a conservative extension of Nuprl can be extend to the full logic of Metaprl under the plausible assumptions that the decision procedures can be *encoded* in the Nuprl logic.

The proof that *Metaprl*$^-$ is a conservative extension proceeds as follows. Let T be the set of terms of Nuprl and T' be the set of terms of *Metaprl*$^-$. We define a mapping $\phi{:}T' \rightarrow T$ satisfying

$$\forall t \in T.\phi(t) = t,$$

i.e., ϕ is the identity mapping when restricted to the terms of Nuprl. The mapping ϕ *embeds* the terms of *Metaprl*$^-$ in the terms of Nuprl. The mapping ϕ is then extended to sequents in the obvious way. We then prove that if

$$\begin{array}{l} \gg^{1'} \sigma \\ \quad \gg^{1'} \sigma_1 \\ \quad\quad \vdots \\ \quad \gg^{1'} \sigma_k \end{array}$$

is a refinement rule of *Metaprl*$^-$, then

$$\begin{array}{l} \gg^{0} \phi(\sigma) \\ \quad \gg^{0} \phi(\sigma_1) \\ \quad\quad \vdots \\ \quad \gg^{0} \phi(\sigma_k) \end{array}$$

is a derived rule of Nuprl. Furthermore, for any reduction

$$t \mapsto_h t',$$

of *Metaprl*$^-$,

$$\phi(t) \mapsto \phi(t')$$

in Nuprl. Thus any direct computation on a term t in *Metaprl*$^-$ can be simulated by direct computation on $\phi(t)$ in Nuprl. Whence, for any Nuprl term $t \in T$, if

$$\vdash^{1'} t$$

then

$$\vdash^{0} \phi(t).$$

But, $\phi(t) = t$ because $t \in T$; therefore,

$$\vdash^{0} t.$$

The definition of the mapping $\phi{:}T' \rightarrow T$ is by structural induction on the terms of *Metaprl*$^-$. Table 3.4 gives the definition of ϕ for the terms of *Metaprl*$^-$

Table 3.4: Translation of Standard Terms of Metaprl

1.	$\phi(a+b)$	$:= \phi(a)+\phi(b)$
2.	$\phi(any(a))$	$:= any(\phi(a))$
3.	$\phi(t(a))$	$:= \phi(t)(\phi(a))$
4.	$\phi(``\alpha")$	$:= ``\alpha"$
5.	$\phi(atom)$	$:= atom$
6.	$\phi(atom\_eq(a;b;s;t))$	$:= atom\_eq(\phi(a);\phi(b);\phi(s);\phi(t))$
7.	$\phi(axiom)$	$:= axiom$
8.	$\phi(a.b)$	$:= \phi(a).\phi(b)$
9.	$\phi(decide(a;x.s;y.t))$	$:= decide(\phi(a);x.\phi(s);y.\phi(t))$
10.	$\phi(a/b)$	$:= \phi(a)/\phi(b)$
11.	$\phi(a=b\in A)$	$:= \phi(a)=\phi(b)\in\phi(A)$
12.	$\phi(x{:}A\rightarrow B)$	$:= x{:}\phi(A)\rightarrow\phi(B)$
13.	$\phi(inl(a))$	$:= inl(\phi(a))$
14.	$\phi(inr(a))$	$:= inr(\phi(a))$
15.	$\phi(ind(a;x,y.s;b;u,v.t))$	$:= ind(\phi(a);x,y.\phi(s);\phi(b);u,v.\phi(t))$
16.	$\phi(int\_eq(a;b;s;t))$	$:= int\_eq(\phi(a);\phi(b);\phi(s);\phi(t))$
17.	$\phi(n)$	$:= n$ (n *a canonical integer*)
18.	$\phi(int)$	$:= int$
19.	$\phi(\lambda x.b)$	$:= \lambda x.\phi(b)$
20.	$\phi(less(a;b;s;t))$	$:= less(\phi(a);\phi(b);\phi(s);\phi(t))$
21.	$\phi(a<b)$	$:= \phi(a)<\phi(b)$
22.	$\phi(list\_ind(a;s;x,y,u.t))$	$:= list\_ind(\phi(a);\phi(s);x,y,u.\phi(t))$
23.	$\phi(A\ list)$	$:= \phi(A)\ list$
24.	$\phi(a\ mod\ b)$	$:= \phi(a)\ mod\ \phi(b)$
25.	$\phi(a*b)$	$:= \phi(a)*\phi(b)$
26.	$\phi(-a)$	$:= -\phi(a)$
27.	$\phi(nil)$	$:= nil$
28.	$\phi(\langle a,b\rangle)$	$:= \langle\phi(a),\phi(b)\rangle$
29.	$\phi(x{:}A\times B)$	$:= x{:}\phi(A)\times\phi(B)$
30.	$\phi(rec(x.A))$	$:= rec(x.\phi(A))$
31.	$\phi(rec\_ind(a;x,y.t))$	$:= rec\_ind(\phi(a);x,y.\phi(t))$
32.	$\phi(\{x{:}A\mid B\})$	$:= \{x{:}\phi(A)\mid\phi(B)\}$
33.	$\phi(spread(a;x,y.t))$	$:= spread(\phi(a);x,y.\phi(t))$
34.	$\phi(a-b)$	$:= \phi(a)-\phi(b)$
35.	$\phi(A\mid B)$	$:= \phi(A)\mid\phi(B)$
36.	$\phi(U_k)$	$:= U_k$
37.	$\phi(x)$	$:= x$ (x *a variable*)
38.	$\phi(void)$	$:= void$

inherited from Nuprl. The following defines ϕ on the "new" terms of Metaprl introduced to code the Nuprl proof theory.

Strings of characters are coded as natural numbers. Let f be a bijective mapping of type $\{\_, a, b, c, \ldots, A, B, C, \ldots\}^* \setminus \{\epsilon\} \to \mathbf{N}$.

$$\begin{array}{ll} \phi(\text{`}\beta\text{'}) & := f(\beta) \\ \phi(char\_string) & := \mathbf{N} \\ \phi(char\_string\_eq(a;b;t;t')) & := int\_eq(\phi(a);\phi(b);\phi(t);\phi(t')) \end{array}$$

The *ident* type in Metaprl represents identifiers with components that are character strings and natural numbers. The translation of identifiers from Metaprl to Nuprl mirrors this.

$$\begin{array}{ll} \phi(ident) & := \phi(char\_string) \times \mathbf{N} \\ \phi(a@b) & := \langle \phi(a), \phi(b) \rangle \\ \phi(id\_parts(x)) & := \phi(x) \end{array}$$

The type $term^0$ is translated to a union of types in Nuprl, one component in the union for each of the term kinds. Define the following notations for Nuprl.

$$\begin{array}{rll} term\_body & := term \times term & (*\ Addition\ *) \\ & \mid term & (*\ Any\ *) \\ & \mid term \times term & (*\ Application\ *) \\ & \vdots & \\ term & := rec(term.term\_body) & \end{array}$$

Then define

$$\phi(term^0) := term.$$

The canonical terms of $term^0$ are coded as injections in this type. Define

$$\begin{array}{ll} addition\_term(b) & := in_{38,1}(b) \\ any\_term(b) & := in_{38,2}(b) \\ application\_term(b) & := in_{38,3}(b) \\ & \vdots \end{array}$$

Then,

$$\begin{array}{ll} \phi(addition\_term^0(b)) & := addition\_term(\phi(b)) \\ \phi(any\_term^0(b)) & := any\_term(\phi(b)) \\ \phi(application\_term^0(b)) & := application\_term(\phi(b)) \\ & \vdots \end{array}$$

The $term\_ind^0$ term is coded by a rec_ind term. Define

$$term\_ind(t; h, b.d_1; \cdots; d_{38}) := rec\_ind(t; h, c.decide_{38}(c; b.d_1; \cdots; d_{38})).$$

Then,

$$\phi(term\_ind^0(t; h, b.d_1; \cdots; d_{38})) := term\_ind(\phi(t); h, b.\phi(d_1); \cdots; \phi(d_{38})).$$

In the following discussion, we will use the notations defined for Metaprl in Nuprl. For example, the notation

$$substitute(b, x, y)$$

in Nuprl stands for

$$\phi(substitute^0(b, x, y)).$$

The translation of the type $rule^0$ and the canonical and noncanonical terms for this type is similar to, but simpler than, the translation of the terms of $term^0$ because $rule^0$ is not a recursive type. The details are left as an exercise.

The subgoals calculation and applies predicate are coded using *rule_case*. Define,

$$
\begin{array}{ll}
applies(s, r) := rule\_case(r; & \\
\quad let\ c, h = s\ and\ A, x = d\ in & \\
\quad\quad is\_universe(c)\ \&\ fresh(x, h) & (*product\ 1*) \\
\quad let\ c, h = s\ in & \\
\quad\quad is\_membership(c)\ cand & \\
\quad\quad\quad is\_product(element\_of\_member(c))\ \& & \\
\quad\quad\quad is\_universe(type\_of\_member(c))\ \& & \\
\quad\quad\quad fresh(d, h) & (*product\ 2*) \\
\quad\quad \vdots & \\
\quad), &
\end{array}
$$

and

$$
\begin{array}{ll}
subgoals(s, r) := rule\_case(r; & \\
\quad let\ c, h = s\ and\ A, x = d\ in & \\
\quad\quad [\langle h, \ulcorner \wedge A \in \wedge c \urcorner \rangle; & \\
\quad\quad \langle h + x{:}A, c \rangle] & (*product\ 1*) \\
\quad let\ c, h = s\ and\ \ulcorner \wedge x{:}\wedge A \times \wedge B \in \wedge Ui \urcorner = c\ in & \\
\quad\quad [\langle h, \ulcorner \wedge A \in \wedge Ui \urcorner \rangle; & \\
\quad\quad \langle h + x{:}A, \ulcorner \wedge B[\wedge d / \wedge x] \in \wedge Ui \urcorner \rangle] & (*product\ 2*) \\
\quad\quad\quad \vdots & \\
\quad). &
\end{array}
$$

Then,

$$\phi(applies^0(s,r)) := applies(\phi(s),\phi(r)),$$

$$\phi(subgoals^0(s,r)) := subgoals(\phi(s),\phi(r)).$$

Finally, we define the mapping ϕ on the terms in Metaprl for proofs. Unlike the other types of Metaprl, $proof^0$ does not have a simple connection to a type in Nuprl. Because of the syntactic restrictions on recursive types that guarantee that the body of the recursive type is monotone, it is necessary to construct the embedding of $proof^0$ in two steps. First, we define the type proof tree that represents trees with nodes representing sequents and rules. Next, we use a predicate, *legal*, and the Nuprl set type to cut the proof tree type down to only those trees where the goal, rule and children are legal refinement steps.[5] Define the following notations for Nuprl.

$$\begin{array}{ll}
proof_tree & := rec(proof_tree.sequent \times rule \times (proof_tree\ list)) \\
goals(c) & := list_ind(c; nil; a,b,h.(fst\ a).h) \\
all_in_list(p,l) & := list_ind(l; true; a,b,h.p(a)\ \&\ h) \\
legalnode(p) & := applies(fst(p),snd(p))\ cand \\
 & \quad (subgoals(fst(p),snd(p)) = goals(third(p)) \in sequent\ list) \\
legal & := \lambda p.rec_ind(p; h,x.legalnode(x)\ \&\ all_in_list(h,third(x)) \\
proof & := \{p{:}proof_tree \mid legal(p)\}
\end{array}$$

Then define,

$$\phi(proof^0) := proof.$$

This concludes the definition of the mapping ϕ. It is immediate from the definition that

$$\forall t \in T.\phi(t) = t.$$

The remaining task is to verify that for every rule of inference of Metaprl, the embedding of ϕ of the rule is derivable in Nuprl.

A simple examination will verify that all of the rules of inference in Metaprl associated with the types *char_string* and *ident* have been correctly embedded in Nuprl. That is, that the images under ϕ of the rules are derived rules in Nuprl. It also follows that the computation rule for *ident* is a valid computation on the embedded terms.

The rules associated with the type $term^0$ in Metaprl are only slightly more complicated. The formation rules are obviously derivable in Nuprl, as are the introduction rules. The noncanonical rules require verification.

[5] With the "recursive types with auxiliary functions" (see Mendler [88]) a more direct translation of type $proof^0$ can be given.

Proposition 3.1 The explicit term induction rule (embedded in Nuprl) is derived. The following is a derived rule of inference.

$$
\begin{array}{l}
H \gg term\_ind(t; h, b.d_1; \cdots; d_{38}) \in G[t/v] \\
\quad by \;\; intro \; over \; v.G \; new \; q, h', b' \\
\quad \gg t \in term \\
\quad q{:}term \to U_1, h'{:}v{:}term' \to G, b'{:}term' \\
\qquad \gg d_1[h', b'/h, b] \in G[any\_term(b')/v] \\
\quad q{:}term \to U_1, h'{:}v{:}term' \to G, b'{:}term' \times term' \\
\qquad \gg d_2[h', b'/h, b] \in G[addition\_term(b')/v] \\
\quad q{:}term \to U_1, h'{:}v{:}term' \to G, b'{:}term' \times term' \\
\qquad \gg d_3[h', b'/h, b] \in G[application\_term(b')/v] \\
\quad q{:}term \to U_1, h'{:}v{:}term' \to G, b'{:}term', \\
\qquad \gg d_4[h', b'/h, b] \in G[atom\_term(b')/v] \\
\qquad \gg d_5 \in G[atom\_type\_term/v] \\
\qquad \vdots \\
where \; term' := \{t{:}term \mid q(t)\}
\end{array}
$$

Proof
Let c be a new variable.

$$
\begin{array}{l}
t{:}term \gg term\_ind(t; h, b.d_1; \cdots; d_{38}) \in G[t/v] \\
\mid by \; intro \; over \; v.G \; using \; term \; at \; U_1 \; new \; q, h', c \\
\mid \gg t \in term \\
\mid q{:}term \to U_1, h'{:}v{:}term' \to G, \; c{:}term\_body' \\
\mid \;\mid \gg decide_{38}(c; b.d_1; \cdots; d_{38})[h'/h] \in G[c/v] \\
\mid \;\mid \;\mid by \; rule \; 2.12 \; over \; v.G \; using \; term\_body' \\
\mid \;\mid \;\mid \gg c \in term\_body' \; [trivial] \\
\mid \;\mid \;\mid b'{:}term' \times term', c = addition\_term(b') \in term\_body' \\
\mid \;\mid \;\mid \;\mid \gg d_1[h', b'/h, b] \in G[addition\_term(b')/v] \; [assumption] \\
\mid \;\mid \;\mid \;\mid \;\mid \;\; \vdots \\
\mid \gg term \in U_1 \; by \; [trivial]
\end{array}
$$

$$
\begin{array}{l}
where \; term' := \{t{:}term \mid q(t)\} \\
and \; term\_body' := term\_body[term'/term]
\end{array}
$$

□

Similarly, the implicit term induction rule is derived.

Proposition 3.2 The following is a derived rule of inference.

$$
\begin{array}{l}
H \gg G \textit{ by intro by elim } t \textit{ new } q, h, v \\
\quad [\textit{ext term\_ind}(t; h, b.d_1; \cdots; d_{38})] \\
\quad q{:}\textit{term} \rightarrow U_1, h{:}v{:}\textit{term}' \rightarrow G, b{:}\textit{term}' \\
\qquad \gg \in G[\textit{any\_term}(b')/t] \; [\textit{ext } d_1] \\
\quad q{:}\textit{term} \rightarrow U_1, h{:}v{:}\textit{term}' \rightarrow G, b{:}\textit{term}' \times \textit{term}' \\
\qquad \gg G[\textit{addition\_term}(b')/t] \; [\textit{ext } d_2] \\
\quad q{:}\textit{term} \rightarrow U_1, h{:}v{:}\textit{term}' \rightarrow G, b{:}\textit{term}' \times \textit{term}' \\
\qquad \gg G[\textit{application\_term}(b')/t] \; [\textit{ext } d_3] \\
\quad q{:}\textit{term} \rightarrow U_1, h{:}v{:}\textit{term}' \rightarrow G, b{:}\textit{term}', \\
\qquad \gg G[\textit{atom\_term}(b')/t] \; [\textit{ext } d_4] \\
\qquad \gg G[\textit{atom\_type\_term}/t] \; [\textit{ext } d_5] \\
\qquad \vdots \\
\textit{where } \; \textit{term}' := \{t{:}\textit{term} \mid q(t)\}.
\end{array}
$$

Proof
Let c be a new variable.

$$
\begin{array}{l}
t{:}\textit{term} \gg G \textit{ by elim } t \textit{ at } U_1 \textit{ new } q, h, c \\
\mid q{:}\textit{term} \rightarrow U_1, h{:}t{:}\textit{term}' \rightarrow G, c{:}\textit{term\_body}' \\
\mid \; \mid \gg G[c/t] \textit{ by rule } 2.11 \; c \textit{ new } b \\
\mid \; \mid b{:}\textit{term} \times \textit{term}, c = \textit{addition\_term}(b) \in \textit{term\_body}' \\
\mid \; \mid \; \mid \gg G[\textit{addition\_term}(b)/t] \; [\textit{assumption}] \\
\mid \; \mid \; \mid \quad \vdots \\
\mid \gg \textit{term} \in U_1 \textit{ by rule } [\textit{trivial}]
\end{array}
$$

$$
\begin{array}{l}
\textit{where } \textit{term}' := \{t{:}\textit{term} \mid q(t)\} \\
\textit{and } \textit{term\_body}' := \textit{term\_body}[\textit{term}'/\textit{term}]
\end{array}
$$

□

Next, note that the redicies for $term^0$ in Metaprl are valid computations in Nuprl.

Proposition 3.3 The term

$$\textit{term\_ind}(\textit{addition\_term}(a); h, b.d_1; \cdots; d_{38})$$

computes to

$$d_1[\lambda z.\textit{term\_ind}(z; h, b.d_1; \cdots; d_{38}), a/h, b]$$

in Nuprl.

Proof

$$\begin{array}{l} term\_ind(additon\_term(a); h, b.d_1; \cdots; d_{38}) \mapsto_h \\ \quad decide_{38}(addition\_term(a); b.d_1; \cdots; d_{38})[\lambda z.term\_ind(z; h, b.d_1; \cdots; d_{38})/h] \\ \qquad \mapsto_h d_1[\lambda z.term\_ind(z; h, b.d_1; \cdots; d_{38}), a/h, b] \end{array}$$

□

The remaining redicies of $term^0$ are similarly derivable in Nuprl. It is easy to verify that the rules of inference in Metaprl for $rule^0$ are also derivable in Nuprl, following the same pattern as the rules of $term^0$. It is also easy to check that the rules for $applies^0$ are $subgoals^0$ are derived rules in Nuprl.

The embedding of the $proof^0$ type and related terms is somewhat more complicated than the embedding of the other terms of Metaprl. Thus, it is less apparent that the translations of the rules are derived rules Nuprl. We will verify each of the $proof^0$ rules in turn. First, we require some simple facts about the Nuprl terms involved in the embedding.

Lemma 3.4 The following is a derived rule of inference.

$$H \gg^0 proof\_tree \in U_i$$

□

Lemma 3.5 The following is a derived rule of inference.

$$H, y{:}sequent \times rule \times (proof\_tree\ list), H' \gg^0 legal\_node(y) \in U_i$$

□

Lemma 3.6 The following is a derived rule of inference.

$$H, k{:}A \rightarrow U_1, a{:}A\ list, H' \gg^0 all\_in\_list(k, a) \in U_i$$

□

Lemma 3.7 The following is a derived rule of inference.

$$H, p{:}proof\_tree, H' \gg^0 legal(p) \in U_i$$

□

The first $proof^0$ rule that we verify is correctly embedded in Nuprl is the formation rule.

Proposition 3.8 The following is a derived rule of inference.

$$H \gg^0 proof \in U_i$$

□

The next fact is used in establishing that the intro rule for $proof^0$ is derived in Nuprl.

Lemma 3.9 The following is a derived rule of inference.

$$H, c{:}proof\ list, H' \gg^0 \ \| all\_in\_list(legal, c) \|$$

□

Proposition 3.10 The following is a derived rule of inference.

$$\begin{array}{l} H \gg^0 \langle s, r, c \rangle \in proof \\ \quad \gg s \in sequent \\ \quad \gg r \in rule \\ \quad \gg c \in proof\ list \\ \quad \gg applies(s, r) \\ \quad \gg subgoals(s, r) = goals(c) \in sequent\ list \end{array}$$

Proof

$$\begin{array}{l}
H \gg \langle s, r, c\rangle \in \mathit{proof}\ \mathit{by}\ \mathit{seq}\ s \in \mathit{sequent}\\
\mid s \in \mathit{sequent}\ [\mathit{assumption}]\\
\mid s \in \mathit{sequent} \gg \langle s, r, c\rangle \in \mathit{proof}\\
\mid \mathit{by}\ \mathit{seq}\ r \in \mathit{rule}\\
\mid\ \mid \gg r \in \mathit{rule}\ [\mathit{assumption}]\\
\mid\ \mid r \in \mathit{rule} \gg \langle s, r, c\rangle \in \mathit{proof}\ \mathit{by}\ \mathit{seq}\ c \in \mathit{proof}\ \mathit{list}\\
\mid\ \mid\ \mid \gg c \in \mathit{proof}\ \mathit{list}\ [\mathit{assumption}]\\
\mid\ \mid\ \mid c \in \mathit{proof}\ \mathit{list} \gg \langle s, r, c\rangle \in \mathit{proof}\ \mathit{by}\ \mathit{seq}\ \mathit{applies}(s, r)\\
\mid\ \mid\ \mid\ \mid \gg \mathit{applies}(s, r)\ [\mathit{assumption}]\\
\mid\ \mid\ \mid\ \mid \mathit{applies}(s, r) \gg \langle s, r, c\rangle \in \mathit{proof}\\
\mid\ \mid\ \mid\ \mid \mathit{by}\ \mathit{seq}\ \mathit{subgoals}(s, r) = \mathit{goals}(c) \in \mathit{sequent}\ \mathit{list}\\
\mid\ \mid\ \mid\ \mid\ \mid \gg \mathit{subgoals}(s, r) = \mathit{goals}(c) \in \mathit{sequent}\ \mathit{list}\ [\mathit{assumption}]\\
\mid\ \mid\ \mid\ \mid\ \mid \mathit{subgoals}(s, r) = \mathit{goals}(c) \in \mathit{sequent}\ \mathit{list}\\
\mid\ \mid\ \mid\ \mid\ \mid \gg \langle s, r, c\rangle \in \mathit{proof}\ \mathit{by}\ \mathit{seq}\ \ \|\mathit{all_in_list}(\mathit{legal}, c)\|\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid \gg\ \|\mathit{all_in_list}(\mathit{legal}, c)\|\ [\mathit{Rule}\ 3.9]\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid \|\mathit{all_in_list}(\mathit{legal}, c)\| \gg \langle s, r, c\rangle \in \mathit{proof}\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid \mathit{by}\ \mathit{intro}\ \mathit{at}\ U_1\ \mathit{new}\ y\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid \mathit{all_in_list}(\mathit{legal}, c) \gg \langle s, r, c\rangle \in \mathit{proof_tree}\ [\mathit{trivial}]\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid \gg \mathit{legal}(\langle s, r, c\rangle)\ \mathit{by}\ \mathit{compute}\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid \gg \mathit{applies}(s, r)\ \mathit{cand}\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid (\mathit{subgoals}(s, r) = \mathit{goals}(c) \in \mathit{sequent}\ \mathit{list}))\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \&\ \mathit{all_in_list}(\mathit{legal}, c)\ [\mathit{trivial}]\\
\mid\ \mid\ \mid\ \mid\ \mid\ \mid\ \mid y{:}\mathit{proof_tree} \gg \mathit{legal}(y) \in U_1\ [\mathit{Rule}\ 3.7]
\end{array}$$

□

The following lemmas are used to justify the embedding of the noncanonical rules of proof^0.

Lemma 3.11 The following is a derived rule of inference.

$$H, q{:}\mathit{proof\_tree} \rightarrow U_1, H' \gg q \in \mathit{proof} \rightarrow U_1$$

□

Lemma 3.12 The following is a derived rule of inference.

$$H, h{:}x{:}\mathit{proof\_tree} \upharpoonright q \rightarrow (\|\mathit{legal}(x)\| \rightarrow G), H' \gg^0 x{:}\mathit{proof} \upharpoonright q \rightarrow G$$

□

Lemma 3.13 The following is a derived rule of inference.

$$
\begin{array}{l}
H, z{:}sequent \times rule \times (proof\_tree \restriction q\ list), legal(z), H' \\
\quad \gg^0 \{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand \\
\quad\quad subgoals(s,r) = goals(c) \in sequent\ list\}
\end{array}
$$

□

The next lemma combines the previous three facts.

Lemma 3.14 The following is a derived rule of inference.

$$
\begin{array}{l}
H, q{:}proof\_tree \to U_1, h{:}x{:}proof\_tree \restriction q \to (\| legal(x) \| \to G), \\
\quad z{:}sequent \times rule \times (proof\_tree \restriction q\ list), legal(z), H' \\
\quad\quad \gg q{:}(proof \to U_1) \times h{:}(p{:}proof \restriction q \to G) \times \\
\quad\quad\quad \{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand \\
\quad\quad\quad\quad subgoals(s,r) = goals(c) \in sequent\ list\}
\end{array}
$$

□

We next prove that the $proof^0$ implicit induction rule is derived in Nuprl under the translation ϕ.

Proposition 3.15 The following is a derived rule of inference.

$$
\begin{array}{l}
H, p{:}proof, H' \gg^0 G\ by\ elim\ p\ new\ q, h, z \\
\quad q{:}proof \to U_1, h{:}p{:}proof \restriction q \to G, \\
\quad\quad z{:}\{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand \\
\quad\quad\quad subgoals(s,r) = goals(c) \in sequent\ list\} \gg G[z/p]
\end{array}
$$

Proof

$$
\begin{array}{l}
H, p{:}proof, H' \gg G\ by\ elim\ p\ at\ U_1\ new\ x, y \\
\mid x{:}proof\_tree \gg legal(x) \in U_1\ [Rule\ 3.7] \\
\mid x{:}proof\_tree, [legal(x)], y{:}(p = x \in proof\_tree) \\
\mid \gg G[x/p]\ by\ seq\ \| legal(x) \| \to G[x/p] \\
\mid \mid \gg \| legal(x) \| \to G[x/p]\ by\ elim\ x\ at\ U_1\ new\ q', h', z' \\
\mid \mid \mid q'{:}proof\_tree \to U_1, h'{:}x{:}proof\_tree \restriction q' \to (\| legal(x) \| \to G[x/p]), \\
\mid \mid \mid z'{:}sequent \times rule \times (proof\_tree \restriction q'\ list) \\
\mid \mid \mid \gg \| legal(z') \| \to G[z'/p]\ by\ intro\ at\ U_1 \\
\mid \mid \mid \mid \| legal(x) \| \gg G[z'/p]
\end{array}
$$

| | | | by seq $q{:}(proof \rightarrow U_1) \times h{:}(p{:}proof \restriction q \rightarrow G) \times$
| | | | $z{:}\{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand$
| | | | $subgoals(s,r) = goals(c) \in sequent\ list\} \rightarrow G[z/p]$
| | | | | $\gg q{:}(proof \rightarrow U_1) \times h{:}(p{:}proof \restriction q \rightarrow G) \times$
| | | | | $z{:}\{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand$
| | | | | $subgoals(s,r) = goals(c) \in sequent\ list\} \rightarrow G[z/p]$
| | | | | by intro
| | | | | | $q{:}(proof \rightarrow U_1) \times h{:}(p{:}proof \restriction q \rightarrow G) \times$
| | | | | | $z{:}\{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand$
| | | | | | $subgoals(s,r) = goals(c) \in sequent\ list\} \gg G[z/p]$
| | | | | | [elim twice and thinning]
| | | | | | | $q{:}(proof \rightarrow U_1), h{:}(p{:}proof \restriction q \rightarrow G),$
| | | | | | | $z{:}\{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand$
| | | | | | | $subgoals(s,r) = goals(c) \in sequent\ list\}$
| | | | | | | $\gg G[z/p]$ [thinning and assumption]
| | | | | $q{:}(proof \rightarrow U_1) \times h{:}(p{:}proof \restriction q \rightarrow G) \times$
| | | | | $z{:}\{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand$
| | | | | $subgoals(s,r) = goals(c) \in sequent\ list\} \rightarrow G[z/p]$
| | | | | $\gg G[z'/p]$ by elim using $\langle q', h', z' \rangle$
| | | | | | $\gg \langle q', h', z' \rangle \in$
| | | | | | $h{:}(p{:}proof \restriction q \rightarrow G) \times$
| | | | | | $z{:}\{s{:}sequent \times r{:}rule \times c{:}proof \restriction q\ list \mid applies(s,r)\ cand$
| | | | | | $subgoals(s,r) = goals(c) \in sequent\ list\}$
| | | | | | [trivial using 3.14]
| | | | | | $G[z'/p] \gg G[z'/p]$ by hyp
| | | | $\gg \| legal(z') \| \in U_1$ [Rule 3.7]
| | | $\gg proof\_tree \in U_1$ [Rule 3.4]
| | $\| legal(x) \| \rightarrow G[x/p] \gg G[x/p]$ [trivial]

□

The explicit term induction rule is proved to be derived in a similar fashion. We have established the following theorem.

Theorem 3.16 For every refinement rule of *Metaprl*$^{-}$,

$$
\begin{array}{l}
\gg^{1'} \sigma \\
\quad \gg^{1'} \sigma_1 \\
\quad \quad \vdots \\
\quad \gg^{1'} \sigma_k,
\end{array}
$$

the corresponding rule of Nuprl,

$$\begin{array}{l} \gg^0 \phi(\sigma) \\ \quad \gg^0 \phi(\sigma_1) \\ \quad\quad \vdots \\ \quad \gg^0 \phi(\sigma_k) \end{array}$$

is a derived rule. □

Corollary 3.17 *Metaprl*$^-$ is a conservative extension of Nuprl. □

Corollary 3.18 *Metaprl*$^-$ is consistent. □

The proof that *Metaprl*$^-$ is a conservative extension of Nuprl extends to the full Metaprl logic under the plausible conjecture that the terms for the applicability of the decision procedures can be defined in the Nuprl logic. More specifically, that a term *applies_equality* can be expressed in Nuprl with the property that for any irreducible term $\bar{s} \in$ *sequent*,

$$applies\_equality(\bar{s}) \mapsto \begin{cases} true & \textit{if the equality decision procedure simulated on } \bar{s} \textit{ accepts} \\ false & \textit{otherwise,} \end{cases}$$

and the analogous term, *applies_arith*, can be defined for arith.

The proof we presented that *Metaprl*$^-$ is consistent is proof-theoretic; proofs in *Metaprl*$^-$ were translated to proofs in Nuprl, and we know that Nuprl is consistent. A semantic account could also be given using the techniques of Allen [5] and Mendler [88].

3.4 Adequacy and Faithfulness

In the previous section, we saw that *Metaprl*$^-$ is consistent, and under the plausible assumption that the decision procedures of Nuprl can be encoded in the Nuprl logic, that Metaprl itself is consistent. What remains is to verify that Metaprl correctly encodes Nuprl proofs. That is, that the type *proof*^{0} really "corresponds" to the set of Nuprl proofs.

To thoroughly treat this issue, a long digression into the semantics of type theory would be necessary. Such a digression is beyond the scope of this section. However, we can outline the concepts and steps necessary to establish the adequacy and faithfulness of Metaprl.

Roughly stated, Metaprl is *adequate (for Nuprl proofs)* if for any Nuprl proof, π, there is a corresponding term in Metaprl, $\bar{\pi}$, such that $\bar{\pi} \in proof^0$. Metaprl is said to be *faithful (to Nuprl proofs)* if for every Metaprl term, $p \in proof^0$, there is a Nuprl proof π such that

$$\bar{\pi} = p \in proof^0.$$

We begin with two facts about the computation system of Metaprl. Recall from chapter 2 that we can give a statement of the correctness of Nuprl that says that all terms compute to canonical elements with a form appropriate to the type. An analogous statement holds for Metaprl.

Theorem 3.19 (Correctness of Metaprl) If

$$\emptyset \vdash^1 t \in T$$

or

$$\emptyset \vdash^1 T\ [ext\ t],$$

then $t \mapsto_h \hat{t}$ and $T \mapsto_h \widehat{T}$ where $\widehat{T}$ is a canonical type and $\hat{t}$ is a canonical term of appropriate type for $\widehat{T}$. □

In fact, the types *ident*, $term^0$, $rule^0$, and $proof^0$ are very well-behaved in the sense that all elements of these types compute to irreducible terms, not just canonical terms.

Theorem 3.20 If

$$\emptyset \vdash^1 t \in T$$

or

$$\emptyset \vdash^1 T\ ext\ t,$$

and if $T \mapsto_h \widehat{T}$ where $\widehat{T}$ is one of *ident*, $term^0$, $rule^0$, or $proof^0$, then $t \mapsto \bar{t}$ where $\bar{t}$ is an irreducible term of type $\widehat{T}$. □

The next lemma states that Metaprl correctly represents Nuprl terms.

Lemma 3.21 There is a bijective mapping, θ, from the set of (syntactic) terms of Nuprl to the set of irreducible elements of $term^0$. □

The mapping θ extends to sequents in the obvious way. The next lemma states that θ corresponds to the antiquotation notation introduced earlier in the chapter.

Lemma 3.22 Let σ be a Nuprl sequent. Then

$$\theta(\sigma) = \ulcorner\sigma\urcorner.$$

□

The corresponding fact holds for rules, where Nuprl rules are the actual designation of a rule and its arguments, and not the abbreviated names like "intro".

Lemma 3.23 There is a bijective mapping, ψ, between the set of Nuprl rules and the set of irreducible elements of rule^0. □

The next two lemmas state that $\mathit{applies}^0$ and $\mathit{subgoals}^0$ correctly represent the valid application of a rule and the calculation of subgoals from a sequent and a rule.

Lemma 3.24 Let σ, $\sigma_1, \ldots, \sigma_k$ be Nuprl sequents, and let ρ be a Nuprl rule. If

$$\begin{array}{l} \gg^0 \sigma \ \mathit{by}\ \rho \\ \quad \gg^0 \sigma_1 \\ \quad \vdots \\ \quad \gg^0 \sigma_k \end{array}$$

is a valid instance of a Nuprl refinement rule, then

$$\mathit{applies}^0(\theta(\sigma), \psi(\rho)),$$

and

$$\mathit{subgoals}^0(\theta(\sigma), \psi(\rho)) = [\theta(\sigma_1); \cdots; \theta(\sigma_k)] \in \mathit{sequent}^0\ \mathit{list}.$$

□

Lemma 3.25 Let $\bar{s}, \bar{s_1}, \ldots, \bar{s_k}$ be irreducible elements of $\mathit{sequent}^0$ and let $\bar{r}$ be an irreducible element of rule^0. Then if $\mathit{applies}^0(\bar{s}, \bar{r})$ and

$$\mathit{subgoals}^0(\bar{s}, \bar{r}) = [\bar{s_1}; \cdots; \bar{s_k}] \in \mathit{sequent}^0\ \mathit{list},$$

then

$$\begin{array}{l} \gg^0 \theta^{-1}(\bar{s})\ \mathit{by}\ \psi^{-1}(\bar{r}) \\ \quad \gg^0 \theta^{-1}(\bar{s_1}) \\ \quad \vdots \\ \quad \gg^0 \theta^{-1}(\bar{s_k}) \end{array}$$

is a valid instance of a Nuprl refinement rule. □

The following theorem combines the previous lemmas into a result about proofs.

Theorem 3.26 There is a bijective mapping, ζ, from the set of Nuprl proofs to the set of irreducible elements of proof^{0}. $\square$

The adequacy and faithfulness of Metaprl is a corollary of this theorem and theorem 3.20.

Theorem 3.27 (Adequacy) Let π be a Nuprl proof of sequent σ. Then,

$$\zeta(\pi) \in \mathit{proof\_of}^{0}(\ulcorner\sigma\urcorner).$$

$\square$

Theorem 3.28 (Faithfulness) Let σ be a Nuprl sequent. Let $p \in \mathit{proof\_of}^{0}(\ulcorner\sigma\urcorner)$ be given. By theorem 3.20, $p \mapsto \bar{p}$ for $\bar{p}$ an irreducible term of proof^{0}. Then,

$$\zeta^{-1}(\bar{p})$$

is a Nuprl proof, and

$$\bar{p} = p \in \mathit{proof\_of}^{0}(\ulcorner\sigma\urcorner).$$

$\square$

Chapter 4

Writing Programs That Construct Proofs

4.1 Introduction

In this chapter we explore techniques for writing programs that construct Nuprl proofs. These programs correspond to functions in Metaprl with a codomain that may be consider to be *proof*.[1] We examine three classes of tactics: *complete tactics, partial tactics* and *search tactics.*

A complete tactic is a mapping from a particular subset of sequents to proofs of the sequent. Since functions in Metaprl are total, complete tactics are similar to derived axioms; for the domain of sequents to which they apply, they provide proofs. Partial tactics are similar to derived rules of inference in that they may result in further proof obligations. The final class of tactics, search tactics, represent potentially failing derived rules. They are similar to LCF style procedural tactics and can be combined using tacticals to produce complex tactics that search for proofs.

4.2 Complete Tactics

Complete tactics produce complete (finished) proofs for a given subset of sequents. Part of the definition of a complete tactic is the explicit specification of the domain to which the tactic is to apply. The requirement that the domain be specified means that complete tactics will be most useful when a class of provable sequents may be circumscribed and a uniform proof given for the class. This includes many

[1] In this chapter, the zero superscript on terms in Metaprl has been uniformly omitted.

subsets of sequents that may proved by a derived axiom of the inference rules, or specified by a decision procedure.

Notation 4.1

$$\begin{array}{l} proof\_of(s) := \{p{:}proof \mid goal(p) = s \in sequent\} \\ \quad proof\_of \in sequent \rightarrow U_1 \\ c\_tactic(P) := s{:}\{s{:}sequent \mid P(s)\} \rightarrow proof\_of(s) \\ \quad c\_tactic \in (sequent \rightarrow U_1) \rightarrow U_1 \end{array}$$

If $\tau \in c\_tactic(P)$ then P is called the *applicability predicate* of τ. Complete tactics are complete in the sense that if the tactic is applicable to a sequent (the predicate is true for that sequent) then the tactic computes a complete proof of the sequent. The other two types of tactics, partial and search tactics, compute representations of potentially incomplete proofs, i.e., proof trees with unproved leaves.

The construction of a complete tactic is illustrated by considering the following problem. Let us call an arbitrary product of the type *int* (e.g., *int*, $int \times int$, $(int \times int) \times (int \times int)$) an *int-or-prod term*. The problem is to construct a proof of any well-formedness goal of the form

$$H \gg T \in U_i,$$

where H is an arbitrary declaration list, T is an int-or-prod term, and U_i is an arbitrary universe. A sequent of this form is called a *prod-formation* goal. Two Nuprl rules are relevant to constructing proofs of prod-formation goals. Integer formation

$$H \gg int \in U_i \text{ } by\ intro$$

and product formation

$$\begin{array}{l} H \gg x{:}A \times B \in U_i \text{ } by\ intro\ new\ y \\ \quad \gg A \in U_i \\ \quad y{:}A \gg B \in U_i. \end{array}$$

Refinement by the product formation rule preserves the property of being a prod-formation goal, so the proof of a prod-formation goal, $H \vdash T \in U_i$, can be constructed inductively based upon the structure of T. Figure 4.1 contains the an abstract computation that produces a proof for prod-formation goals.

The domain of applicability of this tactic is expressed by the following two predicates.

$$
\begin{array}{l}
prod\_tac(\ulcorner H \gg T \in U_i \urcorner) := \\
\quad if\ is\_int(T)\ then \\
\quad\quad \langle \ulcorner H \gg T \in U_i \urcorner, int\_formation, nil \rangle \\
\quad else \\
\quad\quad \langle \ulcorner H \gg T \in U_i \urcorner, product\_formation(y), \\
\quad\quad\quad [prod\_tac(\ulcorner H \gg left\_of\_product(T) \in U_i \urcorner); \\
\quad\quad\quad prod\_tac(\ulcorner H' \gg right\_of\_product(T) \in U_i \urcorner)] \rangle \\
\\
where\ \ y = new\_id(\ulcorner H \gg T \in U_i \urcorner) \\
and\ H' = extend(H, y, left\_of\_product(T))
\end{array}
$$

Figure 4.1: Computing Proofs for Prod-Formation Goals

$$
\begin{array}{l}
is\_int\_or\_prod(t) := \\
\quad if\ is\_int(t)\ then \\
\quad\quad true \\
\quad if\ is\_product(t)\ then \\
\quad\quad is\_int\_or\_prod(left\_of\_product(t))\ \& \\
\quad\quad is\_int\_or\_prod(right\_of\_product(t)) \\
\quad else \\
\quad\quad false
\end{array}
$$

and

$$
\begin{array}{l}
is\_prod\_formation(s) := \\
\quad is\_membership(concl(s))\ cand \\
\quad\quad (is\_int\_or\_prod(fst\_of\_equality(concl(s)))\ \& \\
\quad\quad is\_universe(type\_of\_equality(concl(s)))).
\end{array}
$$

The predicate *is_int_or_prod* can be formalized in Metaprl as

$$
\begin{array}{l}
is\_int\_or\_prod := \\
\quad \lambda t.term\_ind(t; h, b. \\
\quad\quad false;\ (*\ addition\ term\ *) \\
\quad\quad\quad \vdots \\
\quad\quad true;\ (*\ integer\ type\ *) \\
\quad\quad\quad \vdots \\
\quad\quad h(fst(snd(b)))\ \&\ h(snd(snd(b)));\ (*\ product\ type\ *) \\
\quad\quad\quad \vdots \\
\quad)
\end{array}
$$

where all the elided clauses contain "*false*". In the case for the product type in the *term_ind*, the bound variable b represents the triple of the binding variable, the left type and the right type of the product. The predicate *is_prod_formation* is formalized directly in Metaprl.

As is the usual practice with the Nuprl system, these predicates are defined by proving the sequents

$$\vdash \mathit{term} \rightarrow U_1$$

and

$$\vdash \mathit{sequent} \rightarrow U_1,$$

and extracting the computations from the proofs rather than explicitly writing the programs.

In solving subproblems, the tactic is applied to extended declaration lists (H' rather than H in figure 4.1). Thus the proof of the tactic requires that a more general result first be established:

$$\forall t{:}\mathit{term}.(h{:}\{h{:}\mathit{decl\ list} \mid \mathit{is\_prod\_formation}(h,t)\} \rightarrow \mathit{proof\_of}(h,t)).$$

A direct proof of this fact can be constructed by structural induction on t and providing expressions involving the induction hypothesis representing proofs of the subgoals. The reason that expressions for the subproofs would be given rather than extracted from refinements is that the consequent of the lemma is a set type, and the intro rule for the set type requires an explicit extraction value. The difficulty with this approach is that it requires that the major part of the computation of the tactic to be given explicitly early in the proof.

Much more elegant and abstract proofs of conclusion of the form $\mathit{proof\_of}(s)$ can be constructed by employing some derived rules of inference that formally reflect the fact that Nuprl is a refinement logic. Recall that for every refinement rule, if the rule applies to a given goal, then any proofs of the subgoals result in a valid proof of the goal.

Lemma 4.1

$$\begin{aligned}\vdash\ & \forall s{:}\mathit{sequent}.\forall r{:}\mathit{rule}.\\ & \quad \mathit{applies}(s,r) \rightarrow\\ & \qquad (\mathit{proofs\_of}(\mathit{subgoals}(s,r)) \rightarrow \mathit{proof\_of}(s))\end{aligned}$$

Proof

Given $s \in \mathit{sequent}$, $r \in \mathit{rule}$, $\mathit{applies}(s,r)$, and $c \in \mathit{proofs\_of}(\mathit{subgoals}(s,r))$, then $\mathit{goals}(c) = \mathit{subgoals}(s,r)$. Whence, $\langle s,r,c\rangle \in \mathit{proof\_of}(s)$. □

This fact can be stated as a derived rule of inference.

Proposition 4.2 The following is a derived rule of inference.

$$\begin{array}{l} H \gg \mathit{proof\_of}(s) \text{ by } \mathit{intro\ using}\ r \\ \quad \gg s \in \mathit{sequent} \\ \quad \gg r \in \mathit{rule} \\ \quad \gg \mathit{applies}(s,r) \\ \quad \gg \mathit{proofs\_of}(\mathit{subgoals}(s,r)) \end{array}$$

□

The next rule records that an object in $\mathit{proofs\_of}(g)$ can be constructed from the proofs of the elements of the list g.

Lemma 4.3 The following is a derived rule of inference.

$$\begin{array}{l} H \gg \mathit{proofs\_of}(g) \text{ by } \mathit{intro\ using}\ s_1, \ldots, s_k \\ \quad \gg g = [s_1; \ldots; s_k] \in \mathit{sequent\ list} \\ \quad \gg \mathit{proof\_of}(s_1) \\ \quad\ \ \vdots \\ \quad \gg \mathit{proof\_of}(s_k) \end{array}$$

□

If we know what $\mathit{subgoals}(s,r)$ computes to, we can further tailor the derived rule given in proposition 4.2.

Proposition 4.4 Let s and r be given, and assume they satisfy

$$\mathit{subgoals}(s,r) \mapsto [s_1; \cdots; s_k].$$

The following is a derived rule of inference.

$$\begin{array}{l} H \gg \mathit{proof\_of}(s) \text{ by } \mathit{refine\ with}\ r \\ \quad \gg s \in \mathit{sequent} \\ \quad \gg r \in \mathit{rule} \\ \quad \gg \mathit{applies}(s,r) \\ \quad \gg \mathit{proof\_of}(s_1) \\ \quad\ \ \vdots \\ \quad \gg \mathit{proof\_of}(s_k) \end{array}$$

Proof

$H \gg proof\_of(s)$ [*rule 4.2: intro using r*]
| $\gg s \in sequent$ [*assumption*]
| $\gg r \in rule$ [*assumption*]
| $\gg applies(s, r)$ [*assumption*]
| $\gg proof\_of(subgoals(s, r))$ [*rule 4.3: intro using* $s_1, \ldots, s_k$]
| | $\gg subgoals(s, r) = [s_1; \ldots; s_k] \in sequent\ list$ *by computation*
| | $\gg proof\_of(s_1)$ [*assumptiion*]
| | $\vdots$
| | $\gg proof\_of(s_k)$ [*assumption*]

□

Rule 4.4 states, in essence, that goals can be symbolically refined at the meta-level. Combined with direct computation to reduce $s_1, \ldots, s_k$, this derived rule becomes a powerful tool for constructing objects in $proof\_of(s)$, including complete tactics. With it in hand, we proceed with the construction of the prod-formation tactic.

The next three lemmas state some simple properties of the *is_int_or_prod* predicate that will be used in constructing the tactic. These properties result from unrolling the definition of the predicate.

Lemma 4.5

$$\vdash \forall t{:}term.is\_int\_or\_prod(t) \rightarrow (is\_int(t) \vee is\_product(t))$$

□

Lemma 4.6

$$\vdash \forall b{:}ident \times term \times term.is\_int\_or\_prod(product\_term(b)) \rightarrow is\_product(fst(snd(b)))$$

□

Lemma 4.7

$$\vdash \forall b{:}ident \times term \times term.is\_int\_or\_prod(product\_term(b)) \rightarrow is\_product(snd(snd(b)))$$

□

The next three lemmas list some generally useful facts that are used in the proof of the tactic. The first states that substitution for a variable that does not occur free in a term results in the same term.

Lemma 4.8

$$\vdash \forall t, s{:}term.\forall x{:}ident.\neg free(x,t) \rightarrow (t[s/x] = t \in term)$$

□

The next lemma states that assuming two distinct term kinds for a term is a contradiction.

Lemma 4.9 Let α and β be distinct names of term kinds.

$$\vdash \forall t{:}term.\neg((term\_kind(t) = \text{“}\alpha\text{”})\ \&\ (term\_kind(t) = \text{“}\beta\text{”}))$$

□

The next lemma provides a derived rule for case analysis.

Lemma 4.10 The following is a derived rule of inference.

$$\begin{array}{l} H \gg G\ by\ cases\ A\ B \\ \quad \gg A \vee B \\ \ A \gg G \\ \ B \gg G \end{array}$$

□

The following lemma provides the main computation part of the tactic. As discussed above, it strengthens the statement of the tactic to apply to arbitrary declaration lists. In addition, the structure of the conclusion is made explicit. One of the antecedents has been squashed to emphasize that it is not used computationally in the proof. However, it is decidable, and so from the standpoint of proving the assertion it does not matter that the antecedent is squashed. It does, however, affect the efficiency of the program extracted from the proof.

Lemma 4.11

$$\begin{array}{l} \vdash \forall p{:}term.\forall u{:}term.is\_universe(u) \rightarrow \| is\_int\_or\_prod(p) \| \rightarrow \\ \quad \forall d{:}decl\ list.proof\_of(d, \ulcorner \wedge p \in \wedge u \urcorner) \end{array}$$

Proof

In the following proof outline, we have omitted most trivial subgoals so as not to obscure the form of the proof. The proof begins with

$\gg \forall p{:}term.\forall u{:}term.is\_universe(u) \to \| is\_int\_or\_prod(p) \| \to$
$\forall d{:}decl\ list.proof\_of(d, \ulcorner \wedge p \in \wedge u \urcorner)$ [*three intros*]
| $p{:}term, u{:}term, is\_universe(u)$
| $\gg\ \| is\_int\_or\_prod(p) \| \to \forall d{:}decl\ list.proof\_of(d, \ulcorner \wedge p \in \wedge u \urcorner)$
| *by elim p new q, h, b*
| | $q{:}term \to U_1,$
| | $h{:}p{:}term' \to \| is\_int\_or\_prod(p) \| \to \forall d{:}decl\ list.$
| | $proof\_of(d, \ulcorner \wedge p \in \wedge u \urcorner),$
| | $b{:}term' \times term'$
| | $\gg\ \| is\_int\_or\_prod(addition\_term(b)) \| \to$
| | $\forall d{:}decl\ list.proof\_of(d, \ulcorner \wedge addition\_term(b) \in \wedge u \urcorner)$ *by* ...
| | $q{:}term \to U_1,$
| | $h{:}p{:}term' \to \| is\_int\_or\_prod(p) \| \to \forall d{:}decl\ list.$
| | $proof\_of(d, \ulcorner \wedge p \in \wedge u \urcorner),$
| | $b{:}term'$
| | $\gg\ \| is\_int\_or\_prod(any\_term(b)) \| \to$
| | $\forall d{:}decl\ list.proof\_of(d, \ulcorner \wedge any\_term(b) \in \wedge u \urcorner)$ *by* ...
| | $\vdots$

where $term' := \{t{:}term \mid q(t)\}$

The elimination of term p results in a lot of subgoals, but only two enter into the computation of the tactic: the subgoals for the int type and for the product type. The remainder of the subgoals lead to contradictions based upon the assumptions of the term kind of p. We proceed by examining the subproof for the int type which forms the basis of the induction, then the subproof for the product type which forms the recurse part of the tactic program, and finally examine a representative of the remaining cases, and show how a contradiction is derived in these cases. The base case is proved as follows.

$p{:}term, u{:}term, is\_universe(u)$
$\gg\ \| is\_int\_or\_prod(\ulcorner int \urcorner) \| \to$
$\forall d{:}decl\ list.proof\_of(d, \ulcorner int \in \wedge u \urcorner))$ [*intro* (*twice*)]
| $\| is\_int\_or\_prod(\ulcorner int \urcorner) \|, d{:}decl\ list$
| $\gg proof\_of(d, \ulcorner int \in \wedge u \urcorner)$ [*rule 4.4:refine with int_formation*]
| | $\gg applies(\langle d, \ulcorner int \in \wedge u \urcorner \rangle, int\_formation)$ [*trivial*]

The subproof for the product type serves as the induction step of the proof. Let

$$v := new\_id(d, \ulcorner \wedge product\_term(b) \in \wedge u \urcorner)$$

and

$$d' := \textit{extend}(d, v, \textit{left\_of\_product}(\textit{product\_term}(b)))$$

in the following refinement.

$p{:}\textit{term},\ u{:}\textit{term},\ \textit{is\_universe}(u),\ q{:}\textit{term} \to U_1,$
$h{:}p{:}\textit{term}' \to \| \textit{is\_int\_or\_prod}(p) \| \to \forall d{:}\textit{decl list}.\textit{proof\_of}(d, \ulcorner \wedge p \in \wedge u \urcorner),$
$b{:}\textit{ident} \times \textit{term}' \times \textit{term}'$
$\gg \| \textit{is\_int\_or\_prod}(\textit{product\_term}(b)) \| \to$
$\forall d{:}\textit{decl list}.\textit{proof\_of}(d, \ulcorner \wedge \textit{product\_term}(b) \in \wedge u \urcorner))\ [\textit{intro (twice)}]$
$\mid\ \| \textit{is\_int\_or\_prod}(\textit{product\_term}(b)) \|,$
$\mid\ d{:}\textit{decl list}$
$\mid\ \gg \textit{proof\_of}(d, \ulcorner \wedge \textit{product\_term}(b) \in \wedge u \urcorner)$
$\mid\ [\textit{rule 4.4: refine with product\_formation}(v)]$
$\mid\ \mid\ \gg \textit{applies}(\langle d, \ulcorner \wedge \textit{product\_term}(b) \in \wedge u \urcorner \rangle, \textit{product\_formation}(v))$
$\mid\ \mid\ [\textit{trivial}]$
$\mid\ \mid\ \gg \textit{proof\_of}(d, \textit{member\_term}(\textit{left\_of\_product}(\textit{fst\_of\_equality}$
$\mid\ \mid\ \quad (\ulcorner \wedge \textit{product\_term}(b) \in \wedge u \urcorner)),$
$\mid\ \mid\ \qquad \textit{type\_of\_equality}(\ulcorner \wedge \textit{product\_term}(b) \in \wedge u \urcorner))))\ [\textit{computation}]$
$\mid\ \mid\ \mid\ \gg \textit{proof\_of}(d, \ulcorner \wedge \textit{fst}(\textit{snd}(b)) \in \wedge u \urcorner)\ \ [\textit{elim } h]$
$\mid\ \mid\ \mid\ \mid\ \gg \| \textit{is\_int\_or\_prod}(\textit{fst}(\textit{snd}(b))) \|\ [\textit{lemma 4.6}]$
$\mid\ \mid\ \gg \textit{proof\_of}(\textit{extend}(d, v, \textit{left\_of\_equality}(\textit{fst\_of\_equality}$
$\mid\ \mid\ \quad (\ulcorner \wedge \textit{product\_term}(b) \in \wedge u \urcorner))),$
$\mid\ \mid\ \qquad \textit{member\_term}(\textit{right\_of\_product}(\textit{fst\_of\_equality}$
$\mid\ \mid\ \qquad\quad (\ulcorner \wedge \textit{product\_term}(b) \in \wedge u \urcorner)),$
$\mid\ \mid\ \qquad\qquad \textit{type\_of\_equality}(\ulcorner \wedge \textit{product\_term}(b) \in \wedge u \urcorner)))$
$\mid\ \mid\ \qquad\qquad\quad [\textit{id\_of\_product\_term}(\textit{product\_term}(b))/v]\ [\textit{computation}]$
$\mid\ \mid\ \mid\ \gg \textit{proof\_of}(d', \ulcorner \wedge \textit{snd}(\textit{snd}(b)) \in \wedge u \urcorner)$
$\mid\ \mid\ \mid\ \quad [\textit{id\_of\_product\_term}(\textit{product\_term}(b))/v]\ [\textit{rule 4.8: } v \textit{ is not free}]$
$\mid\ \mid\ \mid\ \mid\ \gg \textit{proof\_of}(d', \ulcorner \wedge \textit{snd}(\textit{snd}(b)) \in \wedge u \urcorner)[\textit{elim } h]$
$\mid\ \mid\ \mid\ \mid\ \mid\ \gg \| \textit{is\_int\_of\_prod}(\textit{snd}(\textit{snd}(b))) \|\ [\textit{lemma 4.7}]$

Note that normally, the applications of rule 4.4 would be combined with the succeeding computation steps. They are separated here for clarity.

Finally, we examine one of the vacuous cases; all are proved similarly.

$p{:}term, u{:}term, is\_universe(u), q{:}term \to U_1,$
$h{:}p{:}term' \to \| is\_int\_or\_prod(p) \| \to \forall d{:}decl\ list.proof\_of(d, \ulcorner \wedge p \in \wedge u \urcorner),$
$b{:}term' \times term'$
$\gg \| is\_int\_or\_prod(addition\_term(b)) \| \to$
$\forall d{:}decl\ list.proof\_of(d, \ulcorner \wedge addition\_term(b) \in \wedge u \urcorner))$ *by intro*
| $\gg \forall d{:}decl\ list.proof\_of(d, \ulcorner \wedge addition\_term(b) \in \wedge u \urcorner)$
| *by seq false*
| | $\gg false$
| | [*rule 4.10: cases* $is\_int(addition\_term(b))\ is\_product(addition\_term(b))$]
| | | $\gg is\_int(addition\_term(b)) \vee is\_product(addition\_term(b))$
| | | [*lemma 4.5*]
| | | $is\_int(addition\_term(b)) \gg false$
| | | [*lemma 4.9: contradictory term kinds*]
| | | $is\_product(addition\_term(b)) \gg false$
| | | [*lemma 4.9: contradictory term kinds*]

□

An immediate corollary of this lemma is the definition of the prod-formation tactic.

Theorem 4.12

$\vdash s{:}\{s{:}sequent \mid is\_prod\_formation(s)\} \to proof\_of(s)$

□

This tactic works for the rather simple and contrived example of prod-formation goals. It should be apparent that using the same sorts of definitions and techniques the class of goals to which it applies could be generalized to a larger class of well-formedness goals.

The construction of the prod-formation tactic has been presented in great detail in order to illustrate the exact steps required to define and justify a tactic. Further examples will be worked at a more abstract level.

4.3 Partial Tactics

In the previous section we saw how to write tactics that formalize derived axioms, tactics that provide complete proofs for the sequents to which they apply. In this section, we develop a class of tactics which result in *incomplete* or *partial* proofs of the sequents to which they apply. These tactics, called *partial tactics*, are a generalization of derived rules of inference, in an analogous way that complete tactics generalize derived axioms.

Consider the following derived rule of inference.

$$\begin{array}{l} H \gg G \ \mathit{by\ cases}\ A\ B \\ \quad \gg A \vee B \\ \quad A \gg G \\ \quad B \gg G \end{array}$$

Writing a tactic which formalizes this rule requires that the proof fragment that connects the goal and subgoals of this derived rule be represented in Metaprl. The obvious approach for representing proof fragments, what we call *partial proofs*, is to define a type for proof trees with *open* or *unproved* nodes (the leaves of the proof fragment). This approach might be used in this context, but would result in very inefficient tactics.

A more efficient partial-tactic structure results from representing incomplete proofs in such a way that the unproved nodes are explicit, i.e., the unproved nodes can be located in an incomplete proof without search. A partial proof of the sequent s is represented by the product[2]

$$g{:}\mathit{sequent\ list} \times \mathit{validation}(g, s),$$

where

$$\mathit{validation}(g, s) = (\mathit{proofs\_of}(g) \rightarrow \mathit{proof\_of}(s)).$$

Notation 4.2

$\mathit{prf\_of}(s) := g{:}\mathit{sequent\ list} \times \mathit{validation}(g, s)$
$\quad \mathit{prf\_of} \in \mathit{sequent} \rightarrow U_1$
$\mathit{validation}(g, s) := \mathit{proofs\_of}(g) \rightarrow \mathit{proof\_of}(s)$
$\quad \mathit{validation} \in \mathit{sequent\ list} \rightarrow \mathit{sequent} \rightarrow U_1$

Fix $s \in \mathit{sequent}$ and $\langle g, v\rangle \in \mathit{sequent\ list} \times \mathit{validation}(g, s)$ for the moment. The list g corresponds to the open nodes in an incomplete proof tree. The function v is total and represents a procedure for constructing the proof that connects the goal sequent s and the subgoals g. In a later section, we will see that it is not always necessary to execute the validation, v, just as it is not always necessary to execute complete tactics.

[2] This definition is for representing partial proofs of fixed sequents s. We could define general partial proofs as

$$g{:}\mathit{sequent\ list} \times (\mathit{proofs\_of}(g) \rightarrow \mathit{proof}),$$

in analogy to the type of complete proofs, *proof*, but this would be of limited use.

The next two propositions provide sample elements of $prf\_of(s)$. The first states that any sequent may be considered a partial proof that consists only of the sequent. The second states that each refinement rule induces a simple partial proof of any sequent to which it applies.

Proposition 4.13

$$\vdash \forall s{:}sequent.\langle [s], \lambda c.hd(c)\rangle \in prf\_of(s)$$

□

Proposition 4.14

$$\begin{array}{l}\vdash \forall r{:}rule.\forall s{:}\{s{:}sequent \mid applies(s,r)\}. \\ \quad \langle subgoals(s,r), \lambda c.\langle s,r,c\rangle\rangle \in prf\_of(s)\end{array}$$

□

Recall that a complete tactic has the type

$$s{:}\{s{:}sequent \mid P(s)\} \rightarrow proof\_of(s)$$

where P is some applicability predicate. A partial tactic has the analogous type with partial proof replacing proof.

Notation 4.3

$p\_tactic(P) := s{:}\{s{:}sequent \mid P(s)\} \rightarrow prf\_of(s)$
$\quad p\_tactic \in (sequent \rightarrow U_1) \rightarrow U_1$

Propositions 4.13 and 4.14 can be used to construct partial tactics. Proposition 4.13 results in a degenerate partial tactic that applies to any sequent and computes the degenerate partial proof of that sequent.

Proposition 4.15

$$\vdash p\_tactic(true)\ [ext\ \lambda s.\langle [s], \lambda c.hd(c)\rangle]$$

□

Proposition 4.14 can be used to construct a family of partial tactics parameterized by a rule. The results of these tactics are partial proofs that are the results of refining the sequent by the rule.

Proposition 4.16

$$\vdash \forall r{:}rule.p\_tactic(\lambda s.applies(s,r))$$
$$[ext\ \lambda r.\lambda s.\langle subgoals(s,r), \lambda c.\langle s,r,c\rangle\rangle]$$

□

These sample partial tactics are quite small; both construct partial proofs with depth less than two. How do we construct complicated partial proofs, and thus complicated partial tactics? The most useful technique is to "graft" partial proofs together to form new partial proofs. If p is a partial proof with frontier labeled by the sequents $s_1, \ldots, s_k$ and if $p_1, \ldots, p_k$ are partial proofs of $s_1, \ldots, s_k$ respectively, then a new partial proof of s can be formed by grafting p_i onto p at the leaf labeled s_i, where $1 \leq i \leq k$. In order to make this grafting process precise, it is necessary to develop a little theory about combining validations. Towards this theory, the next several lemmas show how elements of *proof_of*(g) for $g \in$ *sequent list* can be synthesized and analyzed.

Lemma 4.17

$$\vdash \forall a{:}proof.\forall b{:}proof\ list.goals(a.b) = (goal(a).goals(b)) \in sequent\ list$$

□

Lemma 4.18

$$\vdash \forall c{:}proof\ list.\forall s{:}sequent.\forall sl{:}sequent\ list.$$
$$(goals(c) = s.sl \in sequent\ list) \rightarrow (c \neq nil \in proof\ list)$$

Proof

Let $c \in$ *proof list*, $s \in$ *sequent*, and $sl \in$ *sequent list* be given such that $goals(c) = s.sl \in$ *sequent list*. Suppose $c = nil \in$ *proof list*. Then,

$$\begin{aligned} 0 &= list\_ind(nil; 0; f,r,h.1) \\ &= list\_ind(goals(nil); 0; f,r,h.1) \\ &= list\_ind(goals(c); 0; f,r,h.1) \\ &= list\_ind(s.sl; 0; f,r,h.1) \\ &= 1 \end{aligned}$$

Therefore, $c \neq nil \in$ *proof list*. □

Lemma 4.19

$$\vdash \forall s{:}sequent.\forall sl{:}sequent\ list.$$
$$proof\_of(s) \times proofs\_of(sl) \rightarrow proofs\_of(s.sl)$$

□

Lemma 4.20

$$\vdash \forall s{:}sequent.\forall sl{:}sequent\ list.$$
$$proofs\_of(s.sl) \rightarrow proof\_of(s)$$

and

$$\vdash \forall s{:}sequent.\forall sl{:}sequent\ list.$$
$$proofs\_of(s.sl) \rightarrow proofs\_of(sl)$$

□

Lemma 4.21

$$\vdash \forall a{:}sequent\ list.\forall b{:}sequent\ list.$$
$$proofs\_of(a@b) \rightarrow proofs\_of(a)$$

and

$$\vdash \forall a{:}sequent\ list.\forall b{:}sequent\ list.$$
$$proofs\_of(a@b) \rightarrow proofs\_of(b)$$

□

Next, we need a term to represent the type of partial proofs for a given list of sequents, g.

$$prfs\_of(g) = \begin{cases} true & \text{if } null\ g \\ prf\_of(hd\ g) \times prfs\_of(tl\ g) & \text{otherwise} \end{cases}$$

The type *true* serves only as a place-holder when the list g is null. This type could also be represented as a subset of a list type, as is $proofs\_of(g)$. However, this definition would be complicated by the fact that there is no uniform way to calculate the goal of a partial proof.

Notation 4.4

$prfs\_of(g) := list\_ind(g; true; f, r, h.prf\_of(f) \times h)$
  $prfs\_of \in sequent\ list \rightarrow U1$

For a given $g \in$ *sequent list*, $x \in prfs\_of(g)$, the next function computes the aggregate frontier of x, i.e., the list formed by appending together the frontiers of each of the partial proofs that make up x. Abstractly, the mappings is:

$$frontiers(g, x) = \begin{cases} nil & \text{if } null\ g \\ fst(fst(x))@frontiers(tl\ g,\ snd\ x) & \text{otherwise} \end{cases}$$

The Metaprl term for this mapping differs from the abstract version by first constructing a function for frontiers of the appropriate length (the length of g), and then applying it to x.

Notation 4.5

$frontiers(g, x) := list\_ind(g; \lambda y.nil; f, r, h.\lambda y.fst(fst(y))@h(snd(y)))(x)$
$frontiers \in g{:}sequent\ list \rightarrow prfs\_of(g) \rightarrow (sequent\ list)$

The next lemma verifies that *frontiers* represents the desired computation.

Lemma 4.22

$$\begin{array}{l} \vdash \forall s{:}sequent.\forall sl{:}sequent\ list.\forall x{:}prfs\_of(s.sl). \\ \quad frontiers(s.sl, x) = fst(fst(x))@frontiers(sl, snd(x)) \\ \qquad \in sequent\ list \end{array}$$

□

The following facts are used in composing validation functions.

Lemma 4.23

$$\begin{array}{l} \vdash \forall s{:}sequent.\forall sl{:}sequent\ list.\forall x{:}prfs\_of(s.sl). \\ \quad proofs\_of(frontiers(s.sl, x)) \rightarrow \\ \qquad proofs\_of(fst(fst(x))) \end{array}$$

□

Proposition 4.24

$$\begin{array}{l} \vdash g{:}sequent\ list \rightarrow x{:}prfs\_of(g) \rightarrow \\ \quad proofs\_of(frontiers(g, x)) \rightarrow \\ \qquad proofs\_of(g) \end{array}$$

□

We are now in a position to prove that partial proofs can be grafted together. The next proposition encodes a method for combining matching partial proofs. The set type on the consequent of the statement guarantees that the resulting partial proof incorporates all of the argument proofs, and is not degenerate.

Proposition 4.25

$$\begin{aligned}&\vdash \forall s{:}sequent.\forall g{:}sequent\ list.\\&\quad validation(g,s)\ \&\ x{:}prfs\_of(g) \rightarrow\\&\qquad \{y{:}prf\_of(s) \mid fst(y) = frontiers(g,x) \in sequent\ list\}\end{aligned}$$

Proof

Let $s \in sequent$, $g \in sequent\ list$, $v \in validation(g,s)$, and $x \in prfs\_of(g)$ be given. Let *flatten* be the extraction from proposition 4.24. Then

$$\begin{aligned}&\langle frontiers(g,x), v\circ(flatten(g)(x))\rangle \in\\&\quad \forall s{:}sequent.\forall g{:}sequent\ list.\\&\qquad validation(g,s)\ \&\ x{:}prfs\_of(g) \rightarrow\\&\qquad\quad \{y{:}prf\_of(s) \mid fst(y) = frontiers(g,x) \in sequent\ list\}.\end{aligned}$$

□

A particularly useful application of this proposition is a derived rule that allows partial proofs to be constructed by symbolic refinement.

Lemma 4.26 Let $k \in \mathbf{N}$ be given. The following is a derived rule of inference.

$$\begin{aligned}&\gg \forall s_1,\ldots,s_k \in sequent.\\&\quad (prf\_of(s_1) \times \cdots \times prf\_of(s_k)) \rightarrow prfs\_of([s_1;\cdots;s_k])\end{aligned}$$

□

Proposition 4.27 Let $k \in \mathbf{N}$ be given. Let $s, s_1, \ldots, s_k, r$ be terms satisfying

$$\vdash subgoals(s,r) = [s_1;\cdots;s_k] \in sequent\ list.$$

The following is a derived rule of inference.

$$\begin{aligned}&\gg prf\_of(s)\\&\quad \gg s \in sequent\\&\quad \gg r \in rule\\&\quad \gg applies(s,r)\\&\quad \gg prf\_of(s_1)\\&\qquad \vdots\\&\quad \gg prf\_of(s_k)\end{aligned}$$

□

$H \gg G$ *by seq* $A \vee B$ *new* x
| $\gg A \vee B$
| $x{:}A \vee B \gg G$ *by elim* x *new* y, y
| | $x{:}A \vee B, y{:}A \gg G$ [*thinning* x]
| | | $H, y{:}A \gg G$ [*assumption*]
| | $x{:}A \vee B, y{:}B \gg G$ [*thinning* x]
| | | $H, y{:}B \gg G$ [*assumption*]

Figure 4.2: Proof that the Cases Rule is Derived

Note that in many cases, the terms $s_1, \ldots, s_k$ may be calculated by simply computing on the term *subgoals*(s, r) if enough of the structure of s and r is known.

4.4 Example Partial Tactics

Propositions 4.15 and 4.16 provide the first example partial tactics: the identity partial tactic which results in a degenerate partial proof whenever it is applied, and a partial tactic parameterized by a rule that produces a proof fragment that corresponds to one refinement step. Proposition 4.27 provides a simple mechanism whereby partial proofs can be combined. This yields a way to construct partial tactics in a top down fashion by symbolic refinement.

Consider once more the cases rule.

$$\begin{array}{l} H \gg G \textit{ by cases } A\ B \\ \quad \gg A \vee B \\ \quad A \gg G \\ \quad B \gg G \end{array}$$

We can implement a partial tactic for this rule by emulating the proof that the rule is derived. The proof is given in figure 4.2. Figure 4.3 contains the main step from a proof that justifies a partial tactic for the cases rule.

It is not always desirable to construct partial tactics by using proposition 4.27 and simulating the proof that the rule to be implemented is derived. For example, consider the following derived rule.

$$\begin{array}{l} H, n{:}\mathbf{N}, H' \gg G \textit{ by induction new } k, h, m \\ \quad k{:}\mathbf{N}, h{:}\forall m{:}\{m{:}\mathbf{N} \mid m < k\}.G[m/n] \gg G[k/m] \end{array}$$

$h{:}decl\ list, c{:}term, x{:}ident, a{:}term, b{:}term$
$\gg prf\_of(h', c)$ [*Rule 4.27*: $(union\text{-}elim\ x, y, y)$]
$\mid applies(\langle h', c\rangle, (union\text{-}elim\ x, y, y))$
$\mid prf(h' + y{:}a, c)$
$\mid prf(h' + y{:}b, c)$
$where\ \ y := new\_id(h, c)$
$and\ h' := extend(h, x, (union\_term\ a\ b))$

Figure 4.3: One Step in Constructing a Partial Tactic for Cases

If a partial tactic were constructed for this rule by emulating the proof that this is a derived rule, then it would be inefficient to calculate the one subgoal of the rule. In fact, as we will discuss in the next chapter, often only the subgoals of partial tactic need to be computed, not the validation. In this instance, it is preferable to construct the tactic in such a way that it directly computes the subgoal (without reference to proposition 4.27) and then uses proposition 4.27 to construct the validation that proves that the subgoal was correctly computed.

4.5 Search Tactics

The two classes of tactics considered above, *complete* and *partial* tactics, require that an explicit applicability predicate be provided. This is not difficult for tactics that encode a small or easily described computations. For example, it is usually easy to define the domain of applicability of a tactic representing a derived rule of inference. It can be more more difficult to define a domain of applicability for a tactic whose actions include complicated heuristics or search. In some cases, the applicability predicate would have to simulate the intended action of the tactic resulting in a near duplication of the tactic computation.

For those situations where the definition of the applicability predicate would be difficult or result in an inefficient test, we define a third class of tactics, *search tactics*, which do not require applicability predicates.

Notation 4.6

$s\_tactic := s{:}sequent \rightarrow prf\_of(s)?$
$s\_tactic \in U_1$

Search tactics use failure types, $prf\_of(s)?$, to indicate if they apply to a sequent s. The first sample search tactic implements the following uninteresting derived rule of inference.

$$\frac{H \gg G}{\quad H \gg G}$$

That is, the tactic implements the identity rule of inference.

Notation 4.7

$Idtac := \lambda s.inl([s], \lambda c.hd(c))$
$\quad Idtac \in s\_tactic$

Note that *Idtac* never fails as is indicated by the *inl* in the definition. The next tactic, *Refine*, is an encoding of the basic inference rules of the logic, indexed by the the rule. Expressed in pseudo-code, *Refine* is the following function.

```
Refine(r, s) :=
  if applies(s, r) then
    ⟨subgoals(s, r), λc.⟨s, r, c⟩⟩
  else
    failwith "rule does not apply"
```

Expressed formally, we have the following definition for *refine*.

Notation 4.8

$Refine(r) := \lambda s.decide(decide\_applies(s, r);$
$\quad inl(\langle subgoals(s, r), \lambda c.\langle s, r, c\rangle\rangle);$
$\quad failwith$ *"rule does not apply"*$)$
$Refine \in rule \rightarrow s\_tactic$

4.5.1 Combining Search Tactics Using Tacticals

In a previous section we saw that partial tactics could be combined (composed) using derived rules of inference. In this section we examine how search tactics can be combined using explicit higher-order functions called *tacticals*. It is then possible to derive the implicit combining forms in analogy to partial tactics. Tacticals provide basic alternation, sequencing and iteration of tactics, along with some more specialized actions.

The tactical *Orelse* provides a form of alternation of tactics. The search tactic represented by the expressions (t_1 *Orelse* t_2) when applied to a sequent first applies t_1; if this fails then it applies t_2 to the sequent and otherwise returns the result of t_1 applied to the sequent.

Notation 4.9

t_1 *Orelse* $t_2 := \lambda s.((t_1\ s)\ ?\ (t_2\ s))$
Orelse $\in$ *s_tactic* $\rightarrow$ *s_tactic* $\rightarrow$ *s_tactic*

The tactical *Then* provides a form of sequencing for search tactics. The expression (t_1 *Then* t_2) represents a search tactic which when applied to a sequent first applies t_1 and then extends this result by applying t_2 to each of the unproved subgoals.

Let *flatten* be the extract of proposition 4.24. Recall,

$$flatten \in \forall s{:}sequent.\forall g{:}sequent\ list.validation(g,s) \rightarrow x{:}prfs\_of(g) \rightarrow \{y{:}prf\_of(s) \mid fst\ y = frontiers(g,x) \in sequent\ list\}.$$

The expression *anyfailure* scans a list of union elements (for example elements of $prf\_of(s)?$) and returns the boolean false if any are right elements of the union, and returns true otherwise.

Notation 4.10

$anyfailure(l) := list\_ind(l; tt; e, r, h.decide(e; h; ff))$

The *Then* tactical is a formalization of the following pseudo-code.

```
λt1.λt2.λs.
  if isr(t1(s)) then fail
  else
    let g, v = outl(t1(s)) in
      if anyfailure(map t2 g) then fail
      else
        let x = map outl∘t2 g in
          (flatten s g v x)
```

The tactical *ThenL* is a variant of *Then*. The arguments of *ThenL* are a search tactic and a list of search tactics. The resulting tactic applies the argument tactic and then to each sequent in the subgoal list applies the corresponding tactic from the search tactic list; the tactic fails if the lengths of the lists are different or any of the argument tactics fail.

$$ThenL \in s\_tactic \rightarrow s\_tactic\ list \rightarrow s\_tactic$$

The *Try* tactical converts a tactic to one that does not fail. It catches any failures of the argument tactic and returns the degenerate partial proof instead.

Notation 4.11

$Try\ t_1 := \lambda s.(t_1(s)\ ?\ Idtac(s))$
  $Try \in s\_tactic \rightarrow s\_tactic$

The $Repeat_n$ tactical provides bounded iteration of search tactics. The behavior of $Repeat_n\ t$ is to apply t and then iteratively try applying t to each remaining unproved subgoals until the tactic t fails or the bound on iterations, n, is reached. The tactical should satisfy the following equations.

$$Repeat_n\ t = t\ Then\ (Rep_n\ t)$$

where

$$Rep_n\ t = \begin{cases} Idtac & \text{if } n = 0 \\ Try\ (t\ Then\ (Rep_{n-1}\ t)) & \text{if } n > 0 \end{cases}$$

Notation 4.12

$Repeat_n\ t := t\ Then\ ind_{\mathbb{N}}(n; Idtac; x, h.Try\ (t\ Then\ (h\ t)))$
  $Repeat_n \in s\_tactic \rightarrow s\_tactic$

Using the rules for partial functions that have been proposed for Nuprl (see Constable and Smith [39] and Constable and Mendler [37]), it is possible to write an unbound repeat tactical. In practice, for n sufficiently large, the bound on $Repeat_n$ would be unimportant. We will use the notation $Repeat_\infty$ for $Repeat_k$ where k is a large natural number.

The next two tacticals, *Progress* and *Complete*, are for controlling what is an acceptable result of a tactic. The *Progress* tactical applies the argument tactic and fails if the result is a degenerate partial proof, i.e., a partial proof with the only subgoal being the original goal.

```
Progress t :=
  λs.let ⟨g, v⟩ = t(s) in
    if length(g) = 1 cand α-equal(hd(g), s) then
      failwith "Progress"
    else
      ⟨g, v⟩
```

The tactical *Complete* fails unless the argument tactic constructs a complete proof, i.e., a partial proof with no unproved subgoals.

```
Complete t :=
  λs.let g, v = t(s) in
    if g ≠ nil then
      failwith "Complete"
    else
      ⟨g, v⟩
```

$$
\begin{array}{l}
\textit{Hypothesis} := \\
\quad \lambda s.\textit{let } \langle h, c\rangle = s \textit{ in} \\
\quad\quad \textit{list\_ind}(h; \\
\quad\quad\quad \textit{failwith } \text{“}\textit{Hypothesis}\text{”}; \\
\quad\quad\quad a, b, k.\textit{let } \langle x, d\rangle = a \textit{ in } (*\ a \textit{ is a declaration } *) \\
\quad\quad\quad\quad \textit{if } \alpha\textit{-equal}(c, d) \textit{ then} \\
\quad\quad\quad\quad\quad \textit{Refine}(\textit{hyp } x) \\
\quad\quad\quad\quad \textit{else} \\
\quad\quad\quad\quad\quad k)
\end{array}
$$

Figure 4.4: The Code for the Hypothesis Tactic

The explicit tacticals can be used to justify derived rules where the extraction of the rule would be the computation of the tactical. For example, we can derive the following rule.

$$
\begin{array}{l}
H \gg s\_tactic \textit{ by Then } [ext\ t_1 \textit{ Then } t_2] \\
\quad \gg s\_tactic\ [ext\ t_1] \\
\quad \gg s\_tactic\ [ext\ t_2]
\end{array}
$$

The tacticals defined above represent the major tactic combining forms currently used in Nuprl. There are several variations of tacticals, such as *ThenM* and *ThenW* which are easily defined from the forms given above.

4.6 Example Search Tactics

The first search tactic that we present is a simple tactic that scans the hypothesis list of the argument sequent for a term that is *α-equal* to the conclusion of the sequent. If one is located, then the hypothesis rule is applied, thus proving the sequent. Otherwise, the tactic fails. Figure 4.4 contains the code for the *Hypothesis* tactic. Since,

$$\forall c, d \in term.(\alpha\textit{-equal}(c, d) \textit{ is decidable}),$$

it is easy to verify that *Hypothesis* $\in$ *s_tactic*.

The next example of a search tactic we present is a back-chain tactic. This tactic either proves the sequent using the *Hypothesis* tactic or scans the hypothesis list of the argument sequent for an implication, $x{:}a \rightarrow b$, where b is α-equal to the conclusion of the sequent, c. The implication is eliminated, and then the back-chain tactic is invoked in an attempt to prove the antecedent of the implication, a.

$H, f{:}a \rightarrow c, H' \gg C$ *by seq a new x*
| $\gg a$ [*open subgoal*]
| $x{:}a \gg c$ *by elim f on x new y, z*
| | $\gg x \in a$ *by intro*
| | | $y{:}c, z{:}(y = f(x) \in c) \gg c$ *by hyp y*

Figure 4.5: The Schematic Proof Fragment Constructed by *Back*

The main component of the *Backchain* tactic is a tactic, *Back*, which tries to construct one back-chaining step. If successful, *Back* implements the rule

$$\begin{array}{l} H, f{:}a \rightarrow c, H' \gg c \\ \quad \gg a. \end{array}$$

Figure 4.5 contains a schematic representation of the proof fragment constructed by *Back* that justifies this rule.

Figure 4.6 contains the code for the search tactic *Back*. The tactic *Back* uses a sub-function *match_consequent*. The function *match_consequent* takes a term and a declaration list as arguments. It scans the declaration list for an implication (function type) where the consequent matches the argument term. If one is found, then the identifier labeling the declaration and the antecedent of the implication are returned as a left-injected pair (left-injection is, by convention, the flag indicating success). If no matching consequent is found, then the function returns a failure flag.

The tactic *Backchain* is a simple combination of *Back* and *Hypothesis*.

$$Backchain := Repeat_{\infty}(Hypothesis\ Orelse\ Back)$$

There are a number of simple improvements that would result in a dramatically better tactic. The routine that matches the consequent could be generalized so that the conclusion need not be an immediate consequent, but a suffix of the implication (e.g., $x{:}a \rightarrow y{:}b \rightarrow c$ would match c). The matching function could also be improved by having it infer instantiations of the binding variables of the function type (e.g., $x{:}int \rightarrow (f(x) = x \in int)$ would match the conclusion $f(0) = 0 \in int$). The tactic could also be improved by generalizing *Back* to treat conjunctions and disjunctions in the conclusion and the implication non-atomically.

```
Back s :=
  let ⟨h, c⟩ = s and
    r = match_consequent c h in
      if isr(r) then
        failwith "Back: no match"
      else
        let ⟨f, a⟩ = outl(r) and (* identifier and antecedent *)
        ⟨x, y, z⟩ = (new_ids 3 s) in
          Refine (seq a x) ThenL
            [Idtac;
              Refine (function_elim f x y z) ThenL
                [Refine (equality_intro); Refine (hyp y)]]
match_consequent c h :=
  list_ind(h; fail;
    a, b, k.
      let ⟨f, i⟩ = a in (* identifier and term *)
        if is_function(i) cand α-equal(consequent_of_function(i), c)
          then inl(⟨f, antecedent_of_function(i)⟩)
        else
          k)
```

Figure 4.6: The Code for *Back*

Complete Tactics $\Rightarrow$ Partial Tactics

$$\forall P{:}sequent \rightarrow U_1 . c\_tactic(P) \rightarrow p\_tactic(P)$$

Partial Tactics $\Rightarrow$ Search Tactics

$$\forall P{:}\{P{:}sequent \rightarrow U_1 \mid decidable(P)\}.p\_tactic(P) \rightarrow s\_tactic$$

Partial Tactics $\Rightarrow$ Complete Tactics

$$\begin{array}{l} \forall P{:}sequent \rightarrow U_1. \\ \quad t{:}p\_tactic(P) \rightarrow (\forall s{:}\{s{:}sequent \mid P(s)\}.null(fst(t(s)))) \\ \qquad \rightarrow c\_tactic(P) \end{array}$$

Search Tactics $\Rightarrow$ Partial Tactics

$$t{:}s\_tactic \rightarrow p\_tactic(P_t)$$

where

$$P_t = \lambda s.decide(t(s); true; false)$$

Figure 4.7: Relationships Between Tactic Kinds

4.7 Conclusion

In this chapter we described how the logic Metaprl can be used to express proof procedures. We defined three class of tactics, complete, partial, and search, and gave sample tactics of each sort.

Through a series of derived rules, we showed that complete tactic and partial tactics may be constructed in a natural, top-down fashion that corresponds to symbolic refinement of goals. We also showed how search tactics can be combined using explicit tacticals

There are strong relationships between the various kinds of tactics. For any complete tactic there is a corresponding partial tactic that computes partial proofs with no subgoals (finished partial proofs) for the goals to which it applies. For any partial tactic that has a decidable applicability predicate, there is a corresponding search tactic. For any partial tactic that always computes finished partial proofs, there is a corresponding complete tactic. Finally, for any search tactic, there is a corresponding partial tactic where the applicability predicate of the partial tactic simulates the execution of the search tactic. These relationships are summarized in figure 4.7.

It might appear that the only class of tactics necessary are search tactics since they subsume the tactics that can be defined as complete or partial tactics and because they can be combined using explicit tacticals. However, the the important differences in the tactic kinds become apparent not in defining tactics, but in using them. This is the subject of the next chapter.

Chapter 5

Using Tactics to Construct Proofs

5.1 Introduction

The previous two chapters defined a formal metalanguage, Metaprl, for the Nuprl logic and explained how proof tactics are expressed in Metaprl. This chapter describes the mechanism by which tactics are used to construct Nuprl proofs. We define the connection between Nuprl proofs and the representation in Metaprl of the proofs, and describe how the implementation of the proof development system would make using tactics feasible.

Recall from section 3.4 that there is a bijective correspondence between a proof of a sequent s in Nuprl and an irreducible element in the type $proof\_of(s)$ in Metaprl. Thus, a Nuprl proof of s can be considered an element in Metaprl of

$$proof\_of(s).$$

Conversely, for any element of $proof\_of(s)$, p, there is an irreducible element $\bar{p}$ such that

$$p = \bar{p} \in proof\_of(s).$$

The irreducible element $\bar{p}$ can be obtained from p by computation. Thus any element of $proof\_of(s)$ may be viewed as a proof of s in Nuprl via computation and the bijective correspondence between irreducible elements of *proof* and Nuprl proofs.

In this way the type *proof* in Metaprl is a generalization of Nuprl proofs since it contains both irreducible elements corresponding to Nuprl proofs and elements that compute to irreducible elements corresponding to meta-level terms for Nuprl proofs.

The key idea in using Metaprl to construct Nuprl proofs is to accept elements of *proof* (under no hypotheses) as Nuprl proofs. To do this directly would result in a cumbersome notation for Nuprl proofs. However, under the correspondence described above, Nuprl proofs as conventionally written are just an alternative syntax, a different notation for constructions in the type *proof* in Metaprl.

It is at this point that the implementation of the proof development system becomes crucial. It is the system that implements the correspondence between Nuprl proofs and *proof* in Metaprl and in so doing allows the user to construct proofs in the Nuprl logic using the conventional syntax and proof editor of Nuprl. This is how Nuprl proofs in the base logic (proofs that do not involve tactics) are constructed. The next three sections describe how proofs that do employ tactics are constructed.

5.2 Using Complete Tactics

This section describes how complete tactics are used to prove Nuprl theorems. Let $s \in \mathit{sequent}$ and $\tau \in \mathit{c\_tactic}(P)$ be fixed. Under the proof correspondence, it suffices to prove

$$\gg^1 \mathit{proof\_of}(s)$$

in order to prove s in Nuprl. By definition,

$$\tau \in s{:}\{s{:}\mathit{sequent} \mid P(s)\} \rightarrow \mathit{proof\_of}(s).$$

Whence, if $s \in \{s{:}\mathit{sequent} \mid P(s)\}$, then

$$\tau(s) \in \mathit{proof\_of}(s).$$

This reasoning is encapsulated in the following derived rule of inference.

Proposition 5.1 Let $s \in \mathit{sequent}$ and $\tau \in \mathit{c\_tactic}(P)$ for $P \in \mathit{sequent} \rightarrow U_1$ be given. Then the following is a derived rule of inference.

$$\begin{array}{c}\gg^1 \mathit{proof\_of}(s)\\ \gg^1 P(s)\end{array}$$

Proof

$\gg$ *proof_of*(s) *by explicit intro* $\tau(s)$
| $\gg$ $\tau(s) \in$ *proof_of*(s) *by intro using c_tactic*(P)
| | $\gg$ $\tau \in$ *c_tactic*(P) [*definition*]
| | $\gg$ $s \in \{s{:}sequent \mid P(s)\}$ *by intro at* U_1
| | | $\gg$ $s \in$ *sequent* [*assumption*]
| | | $\gg$ $P(\ulcorner\sigma\urcorner)$ [*assumption*]
| | | $s{:}sequent \gg P(s) \in U_1$ [*definition*]

□

The implementation of the proof development system is employed to make this a reasonable mechanism to use. First, the system implements the correspondence between Nuprl proofs and the type *proof* in Metaprl allowing the user to work with a familiar proof syntax. Second, the system implements the previous derived rule of inference for Metaprl.[1] The result is a "rule" that appears to the user as follows:

$$\begin{array}{l} \gg^0 \sigma \textit{ by complete tactic } \tau \\ \quad \gg^1 P(\ulcorner\sigma\urcorner) \end{array}$$

Consider using this rule to apply the *prod_formation* tactic from section 4.2. As it stands, this is not a useful mechanism for employing complete tactics in this instance. The tactic proves a trivial subgoal

$$\gg^0 (int \times int) \times int \in U_1$$

by reducing it to the equally trivial subgoal

$$\gg^1 is\_prod\_formation(\ulcorner(int \times int) \times int \in U_1\urcorner).$$

Since an applicability predicate is an arbitrary mapping from *sequent* to U_1, it is not possible, in general, to eliminate the proof obligation of verifying that the predicate is true on the given sequent. However, a large class of applicability predicates can be identified on which a mechanical proof of applicability can be provided, thereby shifting the requirement from the user to the implementation. The class of predicates for which this is possible is the decidable mappings in *sequent* $\rightarrow U_1$.

[1]Although this derived rule could be stated as a rule of inference for the logic of Metaprl, and this would simplify what is required of the implementation, it would needlessly complicate the formalization of Metaprl proofs which will be discussed in the next chapter.

A *decidable* applicability predicate is a mapping, $p \in sequent \rightarrow U_1$, satisfying

$$\forall s{:}sequent.(P(s) \vee \neg P(s)).$$

The applicability predicate for the *prod_formation* tactic, *is_prod_formation*, is an example of a decidable applicability predicate.

Let P be a decidable applicability predicate and $d \in \forall s{:}sequent.(P(s) \vee \neg P(s))$. Let $\bar{s}$ be a Nuprl sequent. The expression $d(\bar{s})$ computes either to a term of the form $inl(a)$ or $inr(a)$ in $P(\bar{s}) \vee \neg P(\bar{s})$. If $d(\bar{s}) \mapsto_h inl(a)$, then $\vdash P(\bar{s})$. The next three lemmas state this reasoning formally and justify a system implemented mechanical proof of a decidable applicability predicate.

Lemma 5.2 The following is a derived rule of inference.

$$H, inl(a) = inr(b) \in A \mid B, H' \gg G \textit{ by union contradiction}$$

Proof

$$0 = decide(inl(a); x.0; x.1) = decide(inr(b); x.0; x.1) = 1$$

□

Lemma 5.3 The following is a derived rule of inference.

$$\begin{array}{l} H \gg^1 A \textit{ by disjunction } a \\ \quad \gg inl(a) \in A \vee \neg A \end{array}$$

Proof

$H \gg A$ *by seq* $inl(a) \in A \vee \neg A$
| $\gg inl(a) \in A \vee \neg A$ [*assumption*]
| $inl(a) \in A \vee \neg A \gg A$ *by explicit intro* $decide(inl(a); x.x; x.0)$
| | $\gg decide(inl(a); x.x; x.0) \in A$ *by intro using* $A \vee \neg A$ *new* a', b
| | | $a'{:}A, inl(a) = inl(a') \in A \vee \neg A \gg a' \in A$ *by equality*
| | | $b{:}\neg A, inl(a) = inr(b) \in A \vee \neg A \gg 0 \in A$
| | | [*Rule 5.2:union contradiction*]

□

Theorem 5.4 Let P be a decidable applicability predicate such that

$$\vdash P \in sequent \rightarrow U_1$$

and

$$\vdash \forall s{:}sequent.(P(s) \vee \neg P(s)) \textit{ ext } d$$

Edit Theorem

$\bar{H} \gg (int \times int) \times int$ *by prod_formation*

Figure 5.1: Example Use of a Complete Tactic

Let $\bar{s}$ be an irreducible sequent. Then if

$$d(\bar{s}) \mapsto inl(a),$$

for some term a, then

$$\vdash P(\bar{s})$$

Proof

$\gg P(\bar{s})$ *[Rule 5.3:disjunction on a]*
$\mid \gg inl(a) \in P(\bar{s}) \vee \neg P(\bar{s})$ *by seq* $d(\bar{s}) \in P(\bar{s}) \vee \neg P(\bar{s})$
$\mid \mid \gg d(\bar{s}) \in P(\bar{s}) \vee \neg P(\bar{s})$ *[assumption]*
$\mid \mid d(\bar{s}) \in P(\bar{s}) \vee \neg P(\bar{s}) \gg inl(a) \in P(\bar{s}) \vee \neg P(\bar{s})$ *by computation*
$\mid \mid \mid inl(a) \in P(\bar{s}) \vee \neg P(\bar{s}) \gg inl(a) \in P(\bar{s}) \vee \neg P(\bar{s})$ *by hypothesis*

□

This derived rule provides a mechanism whereby the implementation of the proof development system can verify if a decidable applicability predicate is true of a given sequent by computing; if it is true (computes to an *inl* term) then the derived rule can be used to prove it is applicable. This results in a rule for applying complete tactics that appears to the user as in the example in figure 5.1.

In summary, when a complete tactic $\tau \in c\_tactic(P)$ is applied to the Nuprl sequent σ, the implementation performs the following.

1. The defining theorem for τ is consulted in the theorem library, and the applicability predicate, P, is obtained. If τ is not defined then an error is reported to the user.

2. If P has not been proved decidable, the user is presented with the proof obligation

$$\gg^{1} P(\ulcorner\sigma\urcorner).$$

   This justifies $\gg^{0} \sigma$ via the proof correspondence and rule 5.1.

3. If P has been proved decidable, then the extract, d, from this proof is obtained.

4. The expression $d(\ulcorner\sigma\urcorner)$ is computed until canonical. This will either be of the form $inl(a)$ or $inr(a)$ for some term a. If it is of the form $inr(a)$, then it is reported to the user that the tactic τ is not applicable to the sequent σ. If $d(\ulcorner\sigma\urcorner)$ computes to $inl(a)$, then σ is accepted as proved, as justified by rules 5.1 and 5.4.

5.3 Using Partial Tactics

The proof development system provides a mechanism for using partial tactics that is similar to that for complete tactics. Let σ be a Nuprl sequent and fix $\gamma \in p\_tactic(P)$ for $P \in sequent \rightarrow U_1$. Under the proof correspondence, in order to prove

$$\gg {}^0\sigma$$

it suffices to prove

$$\gg {}^1 proof\_of(\ulcorner\sigma\urcorner).$$

The partial tactic γ is in $p\_tactic(P)$, hence by definition,

$$\gamma \in s{:}\{s{:}sequent \mid P(s)\} \rightarrow prf(s),$$

i.e.,

$$\gamma \in s{:}\{s{:}sequent \mid P(s)\} \rightarrow g{:}sequent\ list \times validation(g, s),$$

where

$$validation(g, s) = proofs\_of(g) \rightarrow proof\_of(s).$$

The following derived rule justifies the reduction of a goal to the list of subgoals that is the antecedent of a validation function.

Proposition 5.5 The following is a derived rule of inference.

$$\begin{array}{l}\gg {}^1 proof\_of(s)\ by\ validation\ with\ g\\ \quad \gg validation(g, s)\\ \quad \gg proofs\_of(g)\end{array}$$

□

An immediate corollary is a derived rule that justifies the use of partial tactics.

Proposition 5.6 Let $s \in \mathit{sequent}$ and $\gamma \in \mathit{p\_tactic}(P)$ for $P \in \mathit{sequent} \rightarrow U_1$ be given. The following is a derived rule of inference.

$$\begin{array}{l} \gg^1 \mathit{proof\_of}(s) \text{ by partial tactic } \gamma \\ \quad \gg P(s) \\ \quad \gg \mathit{proofs\_of}(\mathit{fst}(\gamma(s))) \end{array}$$

□

If the applicability predicate is decidable, then the system can implement a "rule" that appears as follows:

$$\begin{array}{l} \gg^0 \sigma \\ \quad \gg^0 \theta^{-1}(\bar{s_1}) \\ \quad\quad \vdots \\ \quad \gg^0 \theta^{-1}(\bar{s_k}) \end{array}$$

where $P(\ulcorner\sigma\urcorner)$ is true (as checked by computation) and $\gamma(s) \mapsto \langle[\bar{s_1},\ldots,\bar{s_k}],v\rangle$, $\bar{s_1},\ldots,\bar{s_k}$ irreducible, and θ is the correspondence between Nuprl sequents and irreducible elements of *sequent* defined in section 3.4. It is desirable to have a rule of the form

$$\begin{array}{l} \gg^0 \sigma \\ \quad \gg^1 P(\ulcorner\sigma\urcorner) \\ \quad \gg^0 \theta^{-1}(\bar{s_1}) \\ \quad\quad \vdots \\ \quad \gg^0 \theta^{-1}(\bar{s_k}) \end{array}$$

when P is not known to be decidable. Note that, were this a rule of logic, it would fail to be a refinement rule since $\gamma(\ulcorner\sigma\urcorner)$ cannot be computed in order to generate the subgoals until $P(\ulcorner\sigma\urcorner)$ has been verified, proving that $\ulcorner\sigma\urcorner$ is in the domain of γ.

In summary, when a partial tactic $\gamma \in \mathit{p\_tactic}(P)$ is applied to a Nuprl sequent σ, the implementation performs the following steps.

1. The defining theorem for γ is consulted in the theorem library, and the applicability predicate, P, is obtained. If γ is not defined then an error is reported to the user.

2. If P has not been proved decidable, the user is presented with the proof obligation
$$\gg^1 P(\ulcorner\sigma\urcorner).$$
Once this is supplied, the system continues with step 5. Note that the subgoals cannot be computed until the predicate has been verified.

3. If P has been proved decidable, then the extract, d, from this proof is obtained.

4. The expression $d(\ulcorner\sigma\urcorner)$ is computed until canonical. This will either be of the form $inl(a)$ or $inr(a)$ for some term a. If it is of the form $inr(a)$, then it is reported to the user that the tactic τ is not applicable to the sequent σ. Otherwise $d(\ulcorner\sigma\urcorner)$ computes to $inl(a)$.

5. The term $\gamma(\ulcorner\sigma\urcorner)$ is computed until it is of the form $\langle[\bar{s}_1, \ldots, \bar{s}_k], v\rangle$, for $\bar{s}_1, \ldots, \bar{s}_k$ irreducible. The user is presented with the proof obligations $\gg^0 \theta^{-1}(\bar{s}_1); \cdots; \gg^0 \theta^{-1}(\bar{s}_k)$.

5.4 Using Search Tactics

Search tactics differ from partial tactics only in the way in which applicability is determined. Whereas partial tactics have explicit domain predicates, search tactics indicate applicability via the form of their results. Let $\xi \in s\_tactic$ be fixed. Then, by definition

$$\xi \in s{:}sequent \rightarrow prf\_of(s)?$$

If a term $inl(a) \in prf(s)?$ for some term a, then, $a \in prf(s)$. Whence the result of a search tactic, if successful, can be treated in much the same manner as the result of a partial tactic.

When the search tactic ξ is applied to a Nuprl sequent σ, the following steps are performed by the implementation.

1. The defining theorem for ξ is consulted in the theorem library. If ξ is not defined then an error is reported to the user.

2. The term $\xi(\ulcorner\sigma\urcorner)$ is computed until canonical. If $\xi(\ulcorner\sigma\urcorner) \mapsto_h inr(b)$, then the user is informed that the search tactic has failed. Otherwise, $\xi(\ulcorner\sigma\urcorner) \mapsto_h inl(a)$ for some a.

3. Since $inl(a) \in prf\_of(\ulcorner\sigma\urcorner)?$, it follows $a \in prf\_of(\ulcorner\sigma\urcorner)$. The term a is computed until it is of the form $\langle[\bar{s}_1, \ldots, \bar{s}_k], v\rangle$, for $\bar{s}_1, \ldots, \bar{s}_k$ irreducible. The user is presented with the proof obligations $\gg^0 \theta^{-1}(\bar{s}_1); \cdots; \gg^0 \theta^{-1}(\bar{s}_k)$.

5.5 Implications of Non-canonical Proofs

Implicit in the acceptance of $p \in proof\_of(\ulcorner\sigma\urcorner)$ as a Nuprl proof of sequent σ is a generalization of the usual concept of proof. The justification that allows elements of *proof* in Metaprl to be associated with Nuprl proofs has two parts.

1. There is a bijective correspondence between irreducible elements of *proof* and Nuprl proofs.

2. For every element $p \in proof$ there is an irreducible term $\bar{p}$ (obtained by computation from p) such that $p = \bar{p} \in proof$.

Thus while $p \in proof$ may not correspond directly to a Nuprl proof, it computes to an element $\bar{p} \in proof$ that does. That is, $p \in proof$ may be a *reducible* proof.

Although a reducible proof, p may always be computed upon to obtain an irreducible proof, for efficiency reasons, it is desirable to avoid doing so whenever possible. There are two reasons that an irreducible proof would need to be computed: the user would like to inspect the proof or extraction is to take place from the proof.

These considerations lead to a representation of proofs as (potentially) reducible objects with computation taking place on the objects only when an irreducible proof is required. Of course, a user should be allowed to inspect a proof whenever he desires, and there is nothing formally we can say about his desires. As for computing proofs so that extraction can be performed, more can be said.

First, not all proofs have terms extracted. Some proofs have trivial or uninteresting extracts, or are used only as lemmas in non-computational parts of other proofs (see below). Second, not every branch of a proof enters into the calculation of the extract.

The subgoals of a refinement rule may be partioned in *computational* and *non-computational* subgoals based upon the extraction clause for the rule. A subgoal is said to be *computational* if the extract from the subgoal is a component in the extract of the goal of the refinement rule. Otherwise, the subgoal is said to be *non-computational*. For example, in the following rule only the second subgoal is computational; the extract from the first and third subgoal does not affect the extract of the goal sequent.

$$
\begin{array}{l}
H \gg x{:}A \times B \textit{ by intro } a \textit{ at } U_i \textit{ new } y \ [\textit{ext } \langle a, b\rangle] \\
\quad \gg a \in A \\
\quad \gg B[a/x] \ [\textit{ext } b] \\
\quad y{:}A \gg B[y/x] \in U_i
\end{array}
$$

In the following rule, all the subgoals are non-computational.

$$
\begin{aligned}
&H \gg \langle a, b\rangle \in x{:}A \times B \text{ by intro at } U_i \text{ new } y \text{ [ext axiom]} \\
&\quad \gg a \in A \\
&\quad \gg b \in B \\
&\quad y{:}A \gg B[y/x] \in U_i
\end{aligned}
$$

In general, all the subgoals of intro rules for conclusions that are equalities (i.e., of the form $t \in T$ or $t = t' \in T$) are non-computational. In addition, subgoals with conclusions that are always of the form of equalities have trivial extracts and thus do not affect the extract of the goal. In the presentation of the Nuprl refinement rules in appendix A, only the computational subgoals have "ext" clauses listed.

We can generalize the computational versus non-computational distinction from the subgoals of rules to the branches of proofs as follows. A branch of a proof is said to be *computational* if the path from the root of the proof tree to the branch follows only computational subgoals. Otherwise, the branch said to be is *non-computational*. When extracting a witness term from a complete proof, only the computational branches of the proof need to be examined. The non-computational branches, by definition, do not affect the term extracted from the proof. Thus even for extraction, it is not necessary to have an irreducible proof. The proof need only be computed far enough that all the computational branches are explicit. This means that representing proofs as reducible objects and reducing parts of the proof upon demand may result in a substantial computational savings. In particular, instances of tactic use that lie on non-computational branches of a proof do not need to be computed.

Despite this, using a tactic on a non-computational branch is not free. It is still necessary, either through proof or computation, to verify that the applicability predicate of the tactic is true. Furthermore, the subgoals of partial tactics and search tactics need to be explicitly computed in order to present the remaining proof obligations to the user.

In summary, reduction of tactic instances in a proof should be lazy since a witness may never be extracted from the proof and only computational branches of proofs are required for extraction. The amount of computation that is necessary immediately when a tactic is invoked depends upon the kind of tactic.

Complete tactics Only the applicability predicate must be verified. This will involve computation if the predicate is known to be decidable.

Partial tactics The applicability predicate must be verified and the subgoals (left) part of the resultant pair must be computed. The validation does not need to be computed.

Search tactics The instance must be computed until it is of the form $inl(a)$. The inl indicates that the tactic has succeeded. Then the subterm a of $inl(a)$ is treated analogously to the result of a partial tactic.

Further computation on a tactic instance takes place only upon demand: when the user wishes to examine the proof or extraction is to take place and the instance is in a computational branch of the proof.

Chapter 6

A Hierarchy of Formal Metalanguages

6.1 Introduction

Constructing a tactic in Metaprl involves the same activities as any proof in Nuprl: defining notations, proving theorems from which functions are extracted, and proving properties of those functions. In the example tactics presented in chapter 4, we employed derived rules of inference in order to make the exposition tractable. We have argued that metamathematical extensibility is required for Nuprl. If we are to implement such derived rules for use in writing tactics in Metaprl, then metamathematical extensibility is also required for Metaprl.

Providing a language for metamathematical extensions of Metaprl is facilitated by the work already done on Metaprl itself and the similarity of the logics of Metaprl and Nuprl. The identifiers, terms, rules and proofs of Metaprl are similar to the identifiers, terms, rules and proofs of Nuprl. Thus, a language almost the same as Metaprl will serve as the metamathematical extension language of Metaprl. However, for this language, too, we will require a language for metamathematical extensions.

In this chapter, a hierarchy of formal theories is defined such that each is the formalization of the proof theory of the previous language and can be used to express metamathematical extensions for it. At the base of the hierarchy is the Nuprl type theory. At any stage in the hierarchy, the next theory is formed by adjoining to the present theory concepts axiomatizing the proof theory of that theory. The construction of the next theory is uniform, requiring the addition of only a finite number of types, terms, and rule schemata, thus implementing the hierarchy is a simple extension to the implementation of Metaprl.

The chapter is organized as follows. First, the hierarchy of metalanguages is defined as extensions to the Metaprl logic. Next, we extend the work of chapter 5 on using tactics to encompass the hierarchy. Next, we discuss how tactics defined at one level may be elevated to a logic higher in the hierarchy using *reflection principles*.

6.2 Definition of the Hierarchy

In this section, we define a hierarchy of formal logics, PRL^k, $k \in \mathbf{N}$. For each $k \in \mathbf{N}$, PRL^{k+1} is a metalanguage for metamathematical extensibility of PRL^k just as Metaprl is such a metalanguage for Nuprl. The logic PRL^0 is defined to be Nuprl. The logic PRL^1 is defined to be Metaprl. The remaining logics are defined inductively from this basis. The construction of PRL^{k+1} from PRL^k closely follows the construction of Metaprl from Nuprl. We present the construction only schematically since it follows this familiar pattern.

Let $k \in \mathbf{N}^+$ be given and assume that PRL^k has been defined. By definition, every term of PRL^k is a term of PRL^{k+1} and every rule of inference of PRL^k is a rule of inference of PRL^{k+1}. Additional rules are included in PRL^{k+1} that formalize the PRL^k proof theory. The definition of PRL^{k+1} proceeds as follows.

6.2.1 Formalizing Identifiers

The set of identifiers is the same for all of the logics in the hierarchy. Thus, the formalization of Nuprl identifiers encoded in the type *ident* of Metaprl will suffice to formalize the identifiers of PRL^k in PRL^{k+1} without change.

6.2.2 Formalizing Terms

The type term^k is introduced into PRL^{k+1} to represent the terms of PRL^k. For each $i \in \mathbf{N}$, the terms of PRL^{i+1} are all the terms of the preceding logic plus those representing the proof theory of PRL^i. This cumulativity of the terms is reflected in the following introduction rule for term^k in the logic PRL^{k+1}.

$$\begin{array}{l} H \gg t \in \mathit{term}^k \textit{ by cumulativity at } j \\ \quad \gg t \in \mathit{term}^j \end{array}$$

where $j < k$. Additional introduction rules are included in PRL^{k+1} for the *new* terms of PRL^k, i.e., those terms of PRL^k that were introduced to formalize the PRL^{k-1} proof theory. The following is an example of one of these introduction rules.

$$H \gg^{k+1} rule^{k-1} \in term^k$$

The noncanonical form for $term^k$ is $term\_ind^k$ with rules and reductions that are analogous to $term\_ind^0$ in Metaprl.

6.2.3 Formalizing Refinement

The type $rule^k$ in PRL^{k+1} represents the refinement rules of PRL^k. This type, like $term^k$, is cumulative:

$$\begin{array}{l} H \gg^{k+1} r \in rule^k \textit{ by cumulativity at } j \\ \quad \gg^{k+1} r \in rule^j \end{array}$$

where $j < k$. Additional introduction rules are included for the *new* rules of PRL^k. The computation clauses for the terms that encode the refinement function of PRL^k, $applies^k$ and $subgoals^k$, also reflect the cumulativity of the logics. If

$$applies^{k-1}(\hat{s}, \hat{r}) \mapsto_h c$$

is a redex of PRL^k, then

$$applies^k(\hat{s}, \hat{r}) \mapsto_h c$$

is a redex of PRL^{k+1}. Similarly for the redicies of $subgoals^k$. The new refinement rules of PRL^k are translated into additional redicies for $applies^k$ and $subgoals^k$ in PRL^{k+1} following the pattern of the formalization of rules in Metaprl.

6.2.4 Formalizing Proofs

The type $proof^k$ in PRL^{k+1} represents PRL^k proofs. The noncanonical term for $proof^k$ is $proof\_ind^k$. The rules of inference for $proof^k$ are those for $proof^0$ in Metaprl with the *0* superscripts replaced by k superscripts. This concludes the sketch of the construction of the hierarchy of languages. The next sections address the questions of how the implementation of the hierarchy would be employed in proof development and how tactics at one level may be elevated to higher levels.

6.3 Tactics in the Hierarchy

The tactic structures developed for Metaprl in chapters 4 and 5 have direct generalizations for the hierarchy of metalanguages. For each language, the types for complete, partial and search tactics are defined as follows.

Notation 6.1

$$\begin{array}{l}
c_tactic^k(P) := s{:}\{s{:}sequent^k \mid P(s)\} \to proof_of^k(s) \\
\quad c_tactic^k \in (sequent^k \to U_1) \to U_1 \\
prf_of^k(s) := g{:}sequent^k\ list \times validation^k(g, s) \\
\quad prf_of^k(s) \in sequent^k \to U_1 \\
validation^k(g, s) := proofs_of^k(g) \to proof_of^k(s) \\
\quad validation^k \in sequent^k\ list \to sequent^k \to U_1 \\
p_tactic^k(P) := s{:}\{s{:}sequent^k \mid P(s)\} \to prf_of^k(s) \\
\quad p_tactic^k \in (sequent^k \to U_1) \to U_1 \\
s_tactic^k := s{:}sequent^k \to prf_of^k(s)? \\
\quad s_tactic^k \in U_1
\end{array}$$

Tactics are constructed in the hierarchy using the same techniques as developed for Metaprl in chapter 4. Validations can be combined in the same ways and the derived rules for combining partial tactics in Metaprl have analogous forms for each PRL^k, $k \in N^+$. Each of the tacticals for the search tactics has a corresponding form for each PRL^k, $k \in N^+$.

The implementation of the proposed system is once again employed to make using tactics feasible. Fix $k \in N$ and let σ be a PRL^k sequent. A bijective correspondence is established between proofs of σ and irreducible elements of the type $proof\_of^k(\ulcorner\sigma\urcorner)$ in PRL^{k+1}. Then, since every element

$$p \in proof\_of^k(\ulcorner\sigma\urcorner)$$

computes to an irreducible element, $\bar{p}$, satisfying

$$\bar{p} = p \in proof\_of^k(\ulcorner\sigma\urcorner),$$

every element of $proof\_of^k(\ulcorner\sigma\urcorner)$ corresponds to a proof of σ in PRL^k. As with Metaprl, the system implementation provides reflection principles based upon this correspondence to accept elements of $proof\_of^k(\ulcorner\sigma\urcorner)$ in PRL^{k+1} as proofs of σ in PRL^k. The procedures for applying tactics are the same as those presented in chapter 5 except that they are relativized to a particular level of the hierarchy. For example, a complete tactic $\tau \in c\_tactic^k(P)$ is applied to a PRL^k sequent σ in the following way.

1. The defining theorem for τ is consulted in the theorem library, and the applicability predicate, P, is obtained. If τ is not defined then an error is reported to the user.

2. If P has not been proved decidable, the user is presented with the proof obligation
$$\gg^{k+1} P(\ulcorner\sigma\urcorner).$$
This justifies $\gg^{k}\sigma$ via the proof correspondence.

3. If P has been proved decidable, then the extract, d, from this proof is obtained.

4. The expression $d(\ulcorner\sigma\urcorner)$ is computed until canonical. This will either be of the form $inl(a)$ or $inr(a)$ for some term a. If it is of the form $inr(a)$, then it is reported to the user that the tactic τ is not applicable to the sequent σ. If $d(\ulcorner\sigma\urcorner)$ computes to $inl(a)$, then σ is accepted as proved, as justified by the relativized analogs of rules 5.1 and 5.4.

When applying a complete tactic in Metaprl that does not have an applicability predicate that has been proved decidable, the user is obliged to prove $\gg^{1} P(\ulcorner\sigma\urcorner)$ without the assistance of tactics since tactics work only for Nuprl, and not Metaprl. With the hierarchy, the user can employ tactics for PRL^{k+1} in establishing a goal $\gg^{k} P(\ulcorner\sigma\urcorner)$.

6.4 Reflecting Tactics

Each language in the hierarchy requires its own set of tactics. The similarity between the languages does allow tactics to be *lifted* or *reflected* to higher languages in some cases. The proof systems are cumulative; a sequent at PRL^{j} is a sequent at PRL^{k} and a proof in PRL^{j} is a proof in PRL^{k} for all $j \leq k$.

Proposition 6.1 Let $j, k, l \in \mathrm{N}^{+}$ be given, $j \leq k < l$. Then,

1. $\vdash^{l} \lambda x.x \in (sequent^{j} \rightarrow sequent^{k})$
2. $\vdash^{l} \lambda x.x \in (proof^{j} \rightarrow proof^{k})$
3. $\vdash^{l} \forall s \in sequent^{j}.(\lambda x.x \in (proof\_of^{j}(s) \rightarrow proof\_of^{k}(s)))$

□

For every $k \in \mathrm{N}^+$, let

$$level^k{:}\mathrm{N} \times sequent^{k-1} \rightarrow U_1,$$

be a mapping satisfying

$$level^k(j,s) = \begin{cases} true & \text{if } s \text{ is a level } j \text{ sequent} \\ false & \text{otherwise,} \end{cases}$$

for every $j \in \mathrm{N}$ and $s \in sequent^{k-1}$. Such a predicate is easily defined. Every sequent $s \in sequent^{k-1}$ satisfying $level^k(j,s)$ is an element of $sequent^j$.

Proposition 6.2 Let $k \in \mathrm{N}^+$ and $j \in \mathrm{N}$ be given. Then,

$$\vdash^{k+1} \lambda x.x \in (\{s{:}sequent^k \mid level^k(j,s)\} \rightarrow sequent^j).$$

□

The following proposition states that complete and partial tactics can be lifted by strengthening the applicability predicates.

Proposition 6.3 Let $j,k \in \mathrm{N}^+, j < k$ and $P \in sequent^k \rightarrow U_1$ be given. If

$$\tau \in c\_tactic^j(P),$$

then

$$\tau \in c\_tactic^k(P'),$$

and if

$$\gamma \in p\_tactic^j(P),$$

then

$$\gamma \in p\_tactic^k(P'),$$

where $P' := \lambda s.(level^k(j,s)\ cand\ P(s))$. □

Although this proposition justifies a lifted form of every complete and partial tactic, the applicability predicate on the lifted tactic is too restrictive in many instances. For example, the *cases* tactic defined in chapter 4 would work without modification on any PRL^1 sequent in PRL^2. But, the version produced by lifting using the proposition does not apply to the sequent $t{:}term^1 \gg int$ in PRL^1 for example.

Not all tactics can be lifted while maintaining the original applicability predicate. The proof constructed by the tactic may depend upon the level at which the tactic is defined; for example, an analysis of an element of $term^j$ does not encompass all the cases for an element of $term^k$ where $k > j$.

Tactics like *cases* that are not dependent upon the level at which they are defined may be lifted in many instances by simulating the proof of the tactic at the higher level. For each $k \in \mathrm{N}^+$, a mapping

$$lift^k{:}proof^{k-1} \rightarrow s\_tactic^k,$$

can be defined so that for every $p \in proof^{k-1}$, $s \in sequent^k$, $lift^k(p)(s)$ simulates the proof p step-by-step in attempting to prove the sequent s, failing if a rule from p fails to apply in the new context. The function $lift^k$ allows tactics like *cases* to be mechanically lifted.

The distinct sets of tactics required for each language in the hierarchy might be a severe practical impediment to using the hierarchy for metamathematical extensibility except that, I conjecture, only the first few languages will be important just as only the first few universes in the cumulative hierarchy of type universes of Nuprl are important.

There is an alternative definition of the hierarchy that makes lifting tactics trivial, but at the cost of complicating the construction of tactics at level 1 and higher. The theoretical limitations presented in the first chapter apply to self-reflection principles, and not to representing the proof theory in the same logic. In fact, in chapter 3 we saw that the Metaprl logic could be encoded in Nuprl. By generalizing that encoding we could define a hierarchy of languages which are essentially the same language, except that there are *syntactic abbreviations* that are the encoding of the proof theory of the previous language. With this scheme, there would be just one type of sequent, and lifting is trivial: any tactic can be lifted without change. The complication is that there is not a unique recognizable term for every term in the languages above level 0; for example, $term^1$ would be an abbreviation for a union term. There would also be additional expense in evaluating the coded terms. I believe that it is better to keep the term structures of the languages distinct, and complicate the rare lifting of generic tactics.

Chapter 7

Conclusion

This thesis has studied using a formal type-theoretic metalanguage as the basis for metamathematical extensibility for Nuprl. We began by showing that theoretical considerations preclude direct attempts to use Nuprl as its own metalanguage for extensibility. We then defined the Metaprl logic; Metaprl is an extension of the Nuprl type theory that is tailored to representing the Nuprl proof theory. As a higher-order type-theoretic programming logic, Metaprl is well suited to specifying and expressing the programs underlying proof tactics.

We identified three classes of tactics: complete tactics, partial tactics, and search tactics. Complete tactics are a generalization of derived axioms. They require explicit statements of the domain, but do not need to be executed when used in non-computational branches of a proof. Partial tactics are a generalization of derived rules of inference. They also require explicit statements of the domain, and need to be executed only far enough to calculate the subgoals when applied in non-computational branches. We chose the representation of partial proofs to allow as much computation as possible to be avoided in these cases and to allow efficient *grafting* to form complex partial tactics from smaller ones. Search tactics are also a generalization of derived rules of inference. Search tactics do not require explicit domain predicates, but rather rely upon computation of the tactic itself to determine applicability. Even though computation is required to determine applicability and to calculate the subgoals of an application of a search tactic, the validation does not have to be executed to complete the proof if the tactic was used on a non-computational branch.

The implementation of the proof development system was used to provide the necessary reflection principles to connect object-level and meta-level proofs, and thereby avoided the theoretical limitations on self-reflection. The implementation was also required to represent proofs so as to allow the lazy evaluation of proof tactics. Finally, we generalized the construction of Metaprl to a hierarchy of metalanguages, each providing for metamathematical extensibility for its predecessor.

It is illuminating to compare the types of tactics in the procedural metalanguage, ML, to the type of search tactics in the formal metalanguage. Stated in the syntax of Nuprl, tactics in ML have the type[1]

$$sequent \rightsquigarrow (sequent\ list \times (prf\ list \rightsquigarrow prf?))?$$

where "$\rightsquigarrow$" is the partial function space constructor. Compare this to the type of search tactics, which is equal to

$$s{:}sequent \rightarrow (g{:}sequent\ list \times (prfs\_of(g) \rightarrow prf\_of(s)))?$$

A ML program t could have the type of a tactic, and fail to be correct in several ways. Let s be a sequent. The tactic application, t(s) could diverge. Suppose that $t(s)$ does converge to $\langle g, v\rangle$. Let c be a list of partial proofs. Then, $v(c)$ could diverge. Also, the type of v does not require c to correspond to the subgoals g, i.e., c is not required to be an achievement of g. The validation could converge, but to a partial proof which does not correspond to s. Finally, the validation can fail even if applied to correct achievements.

All of these problems except for the divergence of the tactic itself are ruled out if the tactic is known to be *valid*. Gordon, Milner, and Wadsworth [58] said:

> Validity is clearly a necessary condition for a tactic to be useful; indeed we may deny that invalid tactics are tactics at all. But it is hard to see how to design a programming language so that all definable objects of type tactic are valid, or how to gain this effect by a type discipline. At most we can adopt a style which encourages the programming of valid tactics...

By using a *programming logic* rather than a programming language, we have been able to give a type that corresponds to only the valid tactics.

In the course of this research we produced an implementation of a fragment of Metaprl that allowed us to experiment with tactic and tactical definition. This implementation demonstrated that tactics could feasibly be written using a formal metalanguage; ultimately, however, the utility of the proposed method for metamathematical extensibility can only be determined by a complete implementation and extensive use.

But such an implementation effort should be undertaken only as part of the next generation of Nuprl. In many respects, the current Nuprl logic provides

[1] Refinement tactics in Nuprl actually have a slightly more complicated type based upon (degenerate) partial proofs rather than sequents.

an excellent framework for metamathematical extensibility. Chief amongst its advantages is the unified treatment of logic and computations that facilitates the development of verified tactics. Another advantage is the expressive, higher-order, constructive logic that is the Nuprl type theory. This language simplifies reasoning about and combining tactics, and allows abstract tactics to be naturally expressed. The formulation of the Nuprl formal system as a refinement logic makes it possible to guarantee tactic validity. The fact that it has constructible subgoals makes it possible to construct tactics using symbolic refinement. But, there are also disadvantages.

Many of the disadvantages stem from the particular formulation of the Nuprl refinement rules and not from anything intrinsic in type theory. These rules were intended to be used in constructing base-logic level proofs, without any tactics. Built into the rules are a number of features for user convenience, features like default arguments in rules and optional new identifiers. In a system with a metalanguage, even the current procedural metalanguage, base-logic level proof steps are almost never written; refinement is performed using the metalanguage and tactics. In the presence of a metalanguage, such "convenience features" are an unnecessary complication since the formalization of proofs must anticipate defaults, and handle additional cases based upon, for example, the absence of optional rule arguments. In the formulation of the Nuprl logic presented in appendix A, we have removed some of the most egregious, least useful, and most easily excised of these features from the logic of study. However, because we intended this work to be assayed in the context of a real proof development system, we have continued to use a logic that is very close to the implemented version of Nuprl.

With the next generation of Nuprl, which is now under preliminary design, there is the opportunity to design a type theory that from the start is intended to support metamathematical extensibility. From the perspective of a metalanguage, the syntax of terms and proofs of the the object language should be as simple as possible. Much of the complexity of Metaprl derives from the complexity of the syntax of Nuprl terms and proofs. Rather than incorporating user conveniences into the base logic, one would use the metalanguage to encode proof abstractions. In addition to the simplifications of the logic listed in appendix A, consideration should be given to coalescing the explicit and implicit forms of rules and using the metalanguage to simulate the implicit forms. An alternate treatment of the decision procedures would also simplify the formalization of Nuprl proofs. The decision procedures are currently presented as rules of inference in the object language, Nuprl. They have the status, and should be implemented as, primitive tactics that will, upon demand, produce a primitive justification for goals that they accept. In most respects, type theory is an excellent language for expressing tactics, but the particular formulation of type theory currently used in Nuprl and the Nuprl system needs to be reconsidered with metamathematical extensibility in mind.

The next generation of Nuprl will also include a new proof development system. From the standpoint of metamathematical extensibility, the most important change to the system will be a new syntactic abbreviation mechanism, a replacement for the current *defs*. Syntactic abbreviation mechanisms for proof development systems are currently being studied by Griffin [60] and by Allen [3].

Appendix A

Summary Of Nuprl Rules

This appendix contains the version of the Nuprl rules of inference that are used as the basis of study in this thesis. The rules are essentially those of the implemented version of the Nuprl system as documented in Constable et al. [41]. However, some changes and simplifications have been made. The highlights of these changes are:

1. Independent versions of the types have been omitted.
2. All equalities are binary.
3. All "optional" arguments in rules (*new*, *over*, etc.) are assumed to be present.
4. All hypotheses are labeled by an identifier.
5. A simple version of recursive types is included.
6. The quotient type and associated rules are omitted.
7. Computation rules that are made redundant by the inclusion of the direct computation rules are omitted.

The details of the individual changes to the rules are listed in section A.14. The numbering of rules corresponds to that used in Constable et al. [41].

A.1 Atom

Formation

1. $H \gg U_i$ *by intro atom* [*ext atom*]
2. $H \gg atom \in U_i$ *by intro*

Canonical

3. $H \gg atom$ *by intro* "β" [*ext* "β"]
4. $H \gg$ "β" $\in atom$ *by intro*

   where 'β' is any sequence of characters.

5. $H \gg atom\_eq(a; b; t; t') \in T$ *by intro new* x
   $\gg a \in atom$
   $\gg b \in atom$
   $x{:}(a = b \in atom) \gg t \in T$
   $x{:}((a = b \in atom) \rightarrow void) \gg t' \in T$

A.2 Equality

Formation

1. $H \gg U_i$ *by intro equality* A $[ext\ a = b \in A]$
   $\gg A \in U_i$
   $\gg A$ $[ext\ a]$
   $\gg A$ $[ext\ b]$
2. $H \gg (a = b \in A) \in U_i$ *by intro*
   $\gg A \in U_i$
   $\gg a \in A$
   $\gg b \in A$

Canonical

3. $H \gg axiom \in (a = b \in A)$ *by intro*
   $\gg a = b \in A$
4. $H, x{:}T, H' \gg x \in T$ *by intro*

A.3 Function

Formation

1. $H \gg U_i$ *by intro function A new x* $[ext\ x{:}A \rightarrow B]$
   $\gg A \in U_i$
   $x{:}A \gg U_i\ [ext\ B]$
2. $H \gg x{:}A \rightarrow B \in U_i$ *by intro new y*
   $\gg A \in U_i$
   $y{:}A \gg B[y/x] \in U_i$

Canonical

5. $H \gg x{:}A \rightarrow B$ *by intro at* U_i *new y* $[ext\ \lambda y.b]$
   $y{:}A \gg B[y/x]\ [ext\ b]$
   $\gg A \in U_i$
6. $H \gg \lambda x.b \in y{:}A \rightarrow B$ *by intro at* U_i *new z*
   $z{:}A \gg b[z/x] \in B[z/y]$
   $\gg A \in U_i$

Noncanonical

7. $H, f{:}(x{:}A \rightarrow B), H' \gg T$ *by elim f on a new y, z* $[ext\ t[f(a)/y]]$
   $\gg a \in A$
   $y{:}B[a/x], z{:}(y = f(a) \in B[a/x]) \gg T\ [ext\ t]$
9. $H \gg f(a) \in B[a/x]$ *by intro using* $x{:}A \rightarrow B$
   $\gg f \in x{:}A \rightarrow B$
   $\gg a \in A$

Equality

10. $H \gg f = g \in x{:}A \rightarrow B$ *by extensionality at* U_i
using $x'{:}A' \rightarrow B',\ x''{:}A'' \rightarrow B''$ *new* z
$z{:}A \gg f(z) = g(z) \in B[z/x]$
$\gg A \in U_i$
$\gg f \in x'{:}A' \rightarrow B'$
$\gg g \in x''{:}A'' \rightarrow B''$

A.4 Int

Formation

1. $H \gg U_i$ *by intro int* [*ext int*]
2. $H \gg int \in U_i$ *by intro*

Canonical

3. $H \gg int$ *by intro c* [*ext c*]
4. $H \gg c \in int$ *by intro*

   where c is an integer constant.

Noncanonical

5. $H \gg \; -t \in int$ *by intro*
   $\gg t \in int$
6. $H \gg int$ *by intro* $\oplus$ [*ext* $m \oplus n$]
   $\gg int$ [*ext m*]
   $\gg int$ [*ext n*]
7. $H \gg m \oplus n \in int$ *by intro*
   $\gg m \in int$
   $\gg n \in int$

   where $\oplus$ is one of $+, -, *, /$, or *mod*.

8. $H, x{:}int, H' \gg T$ *by elim x new* y, w, z [*ext* $ind(x; y, z.t_d; t_b; y, z.t_u)$]
   $y{:}int, w{:}y < 0, z{:}T[y + 1/x] \gg T[y/x]$ [*ext* t_d]
   $\gg T[0/x]$ [*ext* t_b]
   $y{:}int, w{:}0 < y, z{:}T[y - 1/x] \gg T[y/x]$ [*ext* t_u]

9. $H \gg ind(e;x,y.t_d;t_b;x,y.t_u) \in T[e/z]$ *by intro over z.T new u,v,w*
   $\gg e \in int$
   $u{:}int, w{:}u < 0, v{:}T[u+1/z] \gg t_d[u,v/x,y] \in T[u/z]$
   $\gg t_b \in T[0/z]$
   $u{:}int, w{:}0 < u, v{:}T[u-1/z] \gg t_u[u,v/x,y] \in T[u/z]$

10. $H \gg int\_eq(a;b;t;t') \in T$ *by intro new x*
   $\gg a \in int$
   $\gg b \in int$
   $x{:}(a = b \in int) \gg t \in T$
   $x{:}((a = b \in int) \rightarrow void) \gg t' \in T$

11. $H \gg less(a;b;t;t') \in T$ *by intro new x*
   $\gg a \in int$
   $\gg b \in int$
   $x{:}(a < b) \gg t \in T$
   $x{:}((a < b) \rightarrow void) \gg t' \in T$

Computation

12a. $H \gg ind(nt;x,y.t_d;t_b;x,y.t_u) = t \in T$ *by reduce 1 down*
   $\gg t_d[nt,(ind(nt+1;x,y.t_d;t_b;x,y.t_u))/x,y] = t \in T$
   $\gg nt < 0$

12b. $H \gg ind(zt;x,y.t_d;t_b;x,y.t_u) = t \in T$ *by reduce 1 base*
   $\gg t_b = t \in T$
   $\gg zt = 0 \in int$

12c. $H \gg ind(nt;x,y.t_d;t_b;x,y.t_u) = t \in T$ *by reduce 1 up*
   $\gg t_u[nt,(ind(nt-1;x,y.t_d;t_b;x,y.t_u))/x,y] = t \in T$
   $\gg 0 < nt$

A.5 Less

Formation

1. $H \gg U_i$ *by intro less* $[ext\ a < b]$
   $\gg int\ [ext\ a]$
   $\gg int\ [ext\ b]$
2. $H \gg a < b \in U_i$ *by intro*
   $\gg a \in int$
   $\gg b \in int$

Equality

3. $H \gg axiom \in a < b$ *by intro*
   $\gg a < b$

A.6 List

Formation

1. $H \gg U_i$ *by intro list* [*ext A list*]
   $\gg U_i$ [*ext A*]
2. $H \gg A\ list \in U_i$ *by intro*
   $\gg A \in U_i$

Canonical

3. $H \gg A\ list$ *by intro nil at* U_i [*ext nil*]
   $\gg A \in U_i$
4. $H \gg nil \in A\ list$ *by intro at* U_i
   $\gg A \in U_i$
5. $H \gg A\ list$ *by intro* . [*ext* $a.b$]
   $\gg A$ [*ext* a]
   $\gg A\ list$ [*ext* b]
6. $H \gg a.b \in A\ list$ *by intro*
   $\gg a \in A$
   $\gg b \in A\ list$

Noncanonical

7. $H, x{:}A\ list, H' \gg T$ *by elim* x *new* u, v, w [*ext* $list\_ind(x; t_b; u, v, w.t_u)$]
   $\gg T[nil/x]$ [*ext* t_b]
   $u{:}A, v{:}A\ list, w{:}T[v/x] \gg T[u.v/x]$ [*ext* t_u]
8. $H \gg list\_ind(e; t_b; x, y, z.t_u) \in T[e/z]$
   by intro over $z.T$ *using* $A\ list$ *new* u, v, w
   $\gg e \in A\ list$
   $\gg t_b \in T[nil/z]$
   $u{:}A, v{:}A\ list, w{:}T[v/z] \gg t_u[u, v, w/x, y, z] \in T[u.v/z]$

A.7 Miscellaneous

Hypothesis

1. $H, x{:}A, H' \gg A'$ *by hyp x* $[ext\ x]$

   where A' is α-convertible to A.

Sequence

2. $H \gg T$ *by seq* T' *new* x $[ext\ (\lambda x.t)(t')]$
   $\gg T'\ [ext\ t']$
   $x{:}T' \gg T\ [ext\ t]$

Explicit Intro

5. $H \gg T$ *by explicit intro* t $[ext\ t]$
   $\gg t \in T$

Cumulativity

6. $H \gg T \in U_i$ *by cumulativity at* U_j
   $\gg T \in U_j$

   where $j < i$.

Substitution

7. $H \gg T[t/x]$ *by subst* $t = t' \in T'$ *over* $x.T$ *at* U_i $[ext\ s]$
   $\gg t = t' \in T'$
   $\gg T[t'/x]\ [ext\ s]$
   $x{:}T' \gg T \in U_i$

Direct computation

8. $\cdot H \gg\ T$ *by compute using* S [*ext* t]
   $\gg T'$ [*ext* t]

9. $H, x{:}T, H' \gg T''$ *by compute hyp* i *using* S [*ext* t]
   $H, x{:}T', H' >> T''$ [*ext* t]

   where $x{:}T$ is the i^{th} hypothesis, S is obtained from T by "tagging" some of its subterms, and T' is generated by the system by performing some computation steps on subterms of T, as indicated by the tags. A subterm t is tagged by replacing it by $[[n; t]]$, where n is a positive integer constant. The tag causes t to be computed for n steps. There are somewhat complicated restrictions on what subterms of A may be tagged, but most users will likely find it sufficient to know that any subterm of T may be tagged that does not occur within the scope of a binding identifier occurrence in T. An application of one of these rules will fail if the tagging is illegal, or if removing the tags from S does not yield a term equal to T. For a more complete description of this rule, see appendix C of Constable et al. [41].

Equality

10. $H \gg t = t' \in T$ *by equality*

    where the equality of t and t' can be deduced from assumptions that are equalities over T (or equalities over T' where $T = T'$ is deducible using only reflexivity, commutativity and transitivity) using only reflexivity, commutativity and transitivity.

Arith

11. $H \gg C$ *by arith at* U_i

    The *arith* rule is used to justify conclusions which follow from hypotheses by a restricted form of arithmetic reasoning. For a detailed account of the *arith* decision procedure and a proof of its correctness, see the article by Tat-hung Chan in Constable, Johnson, and Eichenlaub [29].

A.8 Product

Formation

1. $H \gg U_i$ *by intro product A new x [ext* $x{:}A \times B$*]*
   $\gg A \in U_i$
   $x{:}A \gg U_i$ *[ext B]*
2. $H \gg x{:}A \times B \in U_i$ *by intro new y*
   $\gg A \in U_i$
   $y{:}A \gg B[y/x] \in U_i$

Canonical

5. $H \gg x{:}A \times B$ *by intro a at* U_i *new y [ext* $\langle a,b\rangle$*]*
   $\gg a \in A$
   $\gg B[a/x]$ *[ext b]*
   $y{:}A \gg B[y/x] \in U_i$
6. $H \gg \langle a,b\rangle \in x{:}A \times B$ *by intro at* U_i *new y*
   $\gg a \in A$
   $\gg b \in B[a/x]$
   $y{:}A \gg B[y/x] \in U_i$

Noncanonical

9. $H, z{:}(x{:}A \times B), H' \gg T$ *by elim z new u, v, w [ext spread(z; u, v.t)]*
   $u{:}A, v{:}B[u/x], w{:}(z = \langle u,v\rangle \in x{:}A \times B) \gg T[\langle u,v\rangle/z]$ *[ext t]*
10. $H \gg spread(e; x, y.t) \in T[e/z]$
    by intro over z.T using $w{:}A \times B$ *new u, v, a*
    $\gg e \in w{:}A \times B$
    $u{:}A, v{:}B[u/w], a{:}(e = \langle u,v\rangle \in w{:}A \times B) \gg t[u,v/x,y] \in T[\langle u,v\rangle/z]$

A.9 Rec

Formation

1. $H \gg rec(z.T) \in U_i$ *by intro new* y
   $y{:}U_i \gg T[y/z] \in U_i$

   where no instance of z bound in $rec(z.T)$ may occur in the domain type of a function space, in the argument of a function application or in a principle argument of an elimination form.

Canonical

2. $H \gg rec(z.T)$ *by intro at* U_i $[ext\ t]$
   $\gg T[rec(z.T)/z]$ $[ext\ t]$
   $\gg rec(z.T) \in U_i$
3. $H \gg t \in rec(z.T)$ *by intro at* U_i
   $\gg t \in T[rec(z.T)/z]$
   $\gg rec(z.T) \in U_i$

Noncanonical

4. $H, x{:}rec(z.T), H' \gg G$ *by unroll* x *new* y, w $[ext\ g[x/y]]$
   $y{:}T[rec(z.T)/z], w{:}(y = x \in T[rec(z.T)/z]) \gg G$ $[ext\ g]$
5. $H, r{:}rec(z.T), H' \gg G$
   by elim r *at* U_i *new* q, h, x $[ext\ rec\_ind(r; h, x.g[\lambda r.true/q])]$
   $q{:}rec(z.T) \to U_i, h{:}r{:}\{r{:}rec(z.t) \mid q(r)\} \to G, x{:}T[\{r{:}rec(z.t) \mid q(r)\}/z]$
   $\gg G[x/r]$ $ext\ g$
   $\gg rec(z.T) \in U_i$
6. $H \gg rec\_ind(r; h, x.t) \in G[r/v]$
   by intro over $v.G$ *using* $rec(z.T)$ *at* U_i *new* q, k, y
   $\gg r \in rec(z.T)$
   $q{:}rec(z.T) \to U_i, k{:}v{:}\{v{:}rec(z.T) \mid q(v)\} \to G,$
   $y{:}T[\{v{:}rec(z.T) \mid q(v)\}/z] \gg t[k, y/h, x] \in G[y/v]$
   $\gg rec(z.T) \in U_i$

A.10 Set

Formation

1. $H \gg U_i$ *by intro set A new x* $[ext\ \{x{:}A \mid B\}]$
   $\gg A \in U_i$
   $x{:}A \gg U_i\ [ext\ B]$

2. $H \gg \{x{:}A \mid B\} \in U_i$ *by intro new y*
   $\gg A \in U_i$
   $y{:}A \gg B[y/x] \in U_i$

Canonical

5. $H \gg \{x{:}A \mid B\}$ *by intro a at* U_i *new y* $[ext\ a]$
   $\gg a \in A$
   $\gg B[a/x]$
   $y{:}A \gg B[y/x] \in U_i$

   All hidden hypothesis in H are visible in the second subgoal.

6. $H \gg a \in \{x{:}A \mid B\}$ *by intro at* U_i *new y*
   $\gg a \in A$
   $\gg B[a/x]$
   $y{:}A \gg B[y/x] \in U_i$

Noncanonical

9. $H, u{:}\{x{:}A \mid B\}, H' \gg T$ *by elim u at* U_i *new y, z* $[ext\ (\lambda y.t)(u)]$
   $y{:}A \gg B[y/x] \in U_i$
   $y{:}A, [B[y/x]], z{:}(u = y \in A) \gg T[y/u]\ [ext\ t]$

   Note that the second new hypotheses of the second subgoal is hidden.

Equality

10. $H \gg \{x{:}A \mid B\} = \{y{:}A' \mid B'\} \in U_i$ *by intro new z*
    $\gg A = A' \in U_i$
    $z{:}A \gg B[z/x] \rightarrow B'[z/y]$
    $z{:}A \gg B'[z/y] \rightarrow B[z/x]$

A.11 Union

Formation

1. $H \gg U_i$ *by intro union* $[ext\ A \mid B]$
   $\gg U_i\ [ext\ A]$
   $\gg U_i\ [ext\ B]$
2. $H \gg A \mid B \in U_i$ *by intro*
   $\gg A \in U_i$
   $\gg B \in U_i$

Canonical

3. $H \gg A \mid B$ *by intro left at* $U_i\ [ext\ inl(a)]$
   $\gg A\ [ext\ a]$
   $\gg B \in U_i$
4. $H \gg inl(a) \in A \mid B$ *by intro at* U_i
   $\gg a \in A$
   $\gg B \in U_i$
5. $H \gg A \mid B$ *by intro right at* $U_i\ [ext\ inr(b)]$
   $\gg B\ [ext\ b]$
   $\gg A \in U_i$
6. $H \gg inr(b) \in A \mid B$ *by intro at* U_i
   $\gg b \in B$
   $\gg A \in U_i$

Noncanonical

7. $H, z{:}A \mid B, H' \gg T$ *by elim* z *new* $x, y, w\ [ext\ decide(z; x.t_l; y.t_r)]$
   $x{:}A, w{:}(z = inl(x) \in A \mid B) \gg T[inl(x)/z]\ [ext\ t_l]$
   $y{:}B, w{:}(z = inr(y) \in A \mid B) \gg T[inr(y)/z]\ [ext\ t_r]$

8. $H \gg decide(e; x.t_l; y.t_r) \in T[e/z]$
$\quad$ *by intro over* $z.T$ *using* $A \mid B$ *new* u, v, w
$\quad \gg e \in A \mid B$
$\quad u{:}A, w{:}(e = inl(u) \in A \mid B) \gg t_l[u/x] \in T[inl(u)/z]$
$\quad v{:}B, w{:}(e = inr(v) \in A \mid B) \gg t_r[v/y] \in T[inr(v)/z]$

A.12 Universe

Canonical

1. $H \gg U_i$ *by intro universe* U_j [*ext* U_j]
2. $H \gg U_j \in U_i$ *by intro*

   where $j < i$. Note that all the formation rules are intro rules for a universe type.

A.13 Void

Formation

1. $H \gg U_i$ *by intro void* [*ext void*]
2. $H \gg void \in U_i$ *by intro*

Noncanonical

3. $H, z{:}void, H' \gg T$ *by elim z* [*ext any(z)*]
4. $H \gg any(e) \in T$ *by intro*
   $\gg e \in void$

A.14 Differences

This section summarizes the differences between the version of the Nuprl logic rules given above, and those documented in Constable et al. [41]. The numbers for the rules refer to the numbering used in Constable et al.

A.14.1 Atom

Rules 6a and 6b are subsumed by the direct computation rules.

A.14.2 Equality

Rules 1 and 2 are changed to assume binary equalities.

A.14.3 Function

The rules for independent functions, rules 3, 4, and 8 are omitted. Rule 10 has been strengthened to correct deficiencies in the original rule. Rule 11 is subsumed by the direct computation rules.

A.14.4 Int

Rules 13a, 13b, 14a and 14b are subsumed by the direct computation rules.

A.14.5 Less

There are no changes in these rules.

A.14.6 List

Rules 9a and 9b are subsumed by the direct computation rules.

A.14.7 Miscellaneous

Rule 2, seq, has been simplified to cut in only 1 formula. Rules 3 and 4, lemma and def, are omitted. See the discussion in chapter 3.

A.14.8 Product

The rules for independent product, rules 3, 4 and 8 are omitted. Rule 11 is subsumed by the direct computation rules.

A.14.9 Rec

These rules are a simple form of the rec-type rules currently implemented in Nuprl.

A.14.10 Set

The rules for independent sets, rules 3, 4, 7, and 8 are omitted.

A.14.11 Union

Rules 9a and 9b are subsumed by the direct computation rules.

A.14.12 Universe

There are no changes in these rules.

A.14.13 Void

There are no changes in these rules.

Appendix B

Summary Of Metaprl Rules

This appendix contains a summary of the inference rules for the Metaprl logic. By definition, every rule of Nuprl is a rule of Metaprl. In addition, the rules listed in this appendix are also rules of Metaprl. In the presentation of these rules we have used the syntactic abbreviations and notation of antiquotation defined for Metaprl in chapter 3.

The rules given in this appendix are intended to encode the Nuprl proof theory including the rules of inference listed in appendix A. Because of the number of rules of inference in Nuprl, only a representative sample of the rules of inference of Metaprl that encode these rules of inference are given here.

Although, we have tried to follow the categories of rules used in the presentation of the Nuprl logic, the naming convention is strained by some of the rules. For example, the "formation" rule for the $subgoals^0$ type is applicable to a conclusion of the form

$$subgoals^0(s,r) \in sequent^0 \; list$$

rather than

$$subgoals^0(s,r) \in U_i.$$

Although this rule is officially a noncanonical rule of $sequent^0$ *list*, it serves the purpose of a formation rule. Unlike the presentation of the Nuprl logic the reduction rules for noncanonical terms are grouped under the appropriate types.

B.1 Char_string

Formation

1. $H \gg U_i$ *by intro char_string* [*ext char_string*]

Canonical

2. $H \gg$ *char_string by intro* 'β' [*ext* 'β']
3. $H \gg$ 'β' $\in$ *char_string by intro*

   where β is any non-null string in $\{\_, a, b, c, \ldots, A, B, C, \ldots\}^*$.

4. $H \gg \mathit{char\_string\_eq}(a; b; t; t') \in T$ *by intro new* x
   $\gg a \in \mathit{char\_string}$
   $\gg b \in \mathit{char\_string}$
   $x{:}(a = b \in \mathit{char\_string}) \gg t \in T$
   $x{:}((a = b \in \mathit{char\_string}) \rightarrow \mathit{void}) \gg t' \in T$

B.2 Ident

Formation

1. $H \gg U_i$ *by intro ident [ext ident]*

Canonical

2. $H \gg$ *ident by intro [ext a@b]*
   $\gg$ *char_string [ext a]*
   $\gg \{x{:}int \mid 0 \le x\}$ *[ext b]*
3. $H \gg a@b \in$ *ident by intro*
   $\gg a \in$ *char_string*
   $\gg b \in \{x{:}int \mid 0 \le x\}$

Noncanonical

4. $H \gg id\_parts(x) \in char\_string \times \{x{:}int \mid 0 \le x\}$ *by intro*
   $\gg x \in ident$
5. $H, v{:}ident, H' \gg G$ *by elim v new x, y, z*
   $x{:}char\_string, y{:}\{x{:}int \mid 0 \le x\}, z{:}(id\_parts(v) = \langle x, y\rangle \in char\_string)$
   $\gg G$

Redicies

6. $id\_parts(v@c) \mapsto_h \langle v, c\rangle$

B.3 Term

Formation

1. $H \gg term^0 \in U_i$ *by intro*

Canonical

2. $H \gg addition\_term^0(b) \in term^0$ *by intro*
   $\gg b \in term^0 \times term^0$
3. $H \gg any\_term^0(b) \in term^0$ *by intro*
   $\gg b \in term^0$
4. $H \gg application\_term^0(b) \in term^0$ *by intro*
   $\gg b \in term^0 \times term^0$
5. $H \gg atom\_term^0(b) \in term^0$ *by intro*
   $\gg b \in term^0$
6. $H \gg atom\_type\_term^0 \in term^0$ *by intro*
7. $H \gg atom\_eq\_term^0(b) \in term^0$ *by intro*
   $\gg b \in term^0 \times term^0 \times term^0 \times term^0$
8. $H \gg axiom\_term^0 \in term^0$ *by intro*
9. $H \gg cons\_term^0(b) \in term^0$ *by intro*
   $\gg b \in term^0 \times term^0$
10. $H \gg decide\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times ident \times term^0 \times ident \times term^0$
11. $H \gg division\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times term^0$
12. $H \gg equality\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times term^0 \times term^0$
13. $H \gg function\_type\_term^0(b) \in term^0$ *by intro*
    $\gg b \in ident \times term^0 \times term^0$
14. $H \gg in\_left\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0$

15. $H \gg in\_right\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0$
16. $H \gg induction\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times ident \times ident \times term^0 \times term^0 \times ident \times ident \times term^0$
17. $H \gg int\_eq\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times term^0 \times term^0 \times term^0$
18. $H \gg integer\_term^0(b) \in term^0$ *by intro*
    $\gg b \in int$
19. $H \gg integer\_type\_term \in term^0$ *by intro*
20. $H \gg lambda\_term^0(b) \in term^0$ *by intro*
    $\gg b \in ident \times term^0$
21. $H \gg less\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times term^0 \times term^0 \times term^0$
22. $H \gg less\_than\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times term^0$
23. $H \gg list\_induction\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times term^0 \times ident \times ident \times ident \times term^0$
24. $H \gg list\_type\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0$
25. $H \gg modulo\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times term^0$
26. $H \gg multiplication\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term \times term^0$
27. $H \gg negation\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0$
28. $H \gg nil\_term^0 \in term^0$ *by intro*
29. $H \gg pair\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times term^0$
30. $H \gg product\_type\_term^0(b) \in term^0$ *by intro*
    $\gg b \in ident \times term^0 \times term^0$
31. $H \gg rec\_term^0(b) \in term^0$ *by intro*
    $\gg b \in ident \times term^0$
32. $H \gg rec\_ind\_term^0(b) \in term^0$ *by intro*
    $\gg b \in term^0 \times ident \times ident \times term^0$
33. $H \gg set\_type\_term^0(b) \in term^0$ *by intro*
    $\gg b \in ident \times term^0 \times term^0$

34. $H \gg spread\_term^0(b) \in term^0$ *by intro*
$\gg b \in term^0 \times ident \times ident \times term^0$

35. $H \gg subtraction\_term^0(b) \in term^0$ *by intro*
$\gg b \in term^0 \times term^0$

36. $H \gg union\_type\_term^0(b) \in term^0$ *by intro*
$\gg b \in term^0 \times term^0$

37. $H \gg universe\_term^0(b) \in term^0$ *by intro*
$\gg b \in int$

38. $H \gg variable\_term^0(b) \in term^0$ *by intro*
$\gg b \in ident$

39. $H \gg void\_term^0 \in term^0$ *by intro*

Noncanonical

40. $H \gg term\_ind^0(t; h, b.d_1; \cdots; d_{38}) \in G[t/v]$
by intro over $v.G$ *new* q, h', b'
$\gg t \in term^0$
$q{:}term^0 \to U_1, h'{:}v{:}term' \to G, b'{:}term' \times term'$
$\gg d_1[h', b'/h, b] \in G[addition\_term^0(b')/v]$
$q{:}term^0 \to U_1, h'{:}v{:}term' \to G, b'{:}term'$
$\gg d_2[h', b'/h, b] \in G[any\_term^0(b')/v]$
$q{:}term^0 \to U_1, h'{:}v{:}term' \to G, b'{:}term' \times term'$
$\gg d_3[h', b'/h, b] \in G[application\_term^0(b')/v]$
$q{:}term^0 \to U_1, h'{:}v{:}term' \to G, b'{:}term',$
$\gg d_4[h', b'/h, b] \in G[atom\_term^0(b')/v]$
$\gg d_5 \in G[atom\_type\_term^0/v]$
$\vdots$

where $term' := \{t{:}term^0 \mid q(t)\}$. The $d_k s$ are restricted so that h and b are not free in those $d_k s$ that do not have argument terms.

41. $H \gg G$ *by elim t new* q, h, b
$[ext\ term\_ind^0(t; h, b.d_1; \cdots; d_{38})]$
$q{:}term^0 \rightarrow U_1, h{:}t{:}term' \rightarrow G, b{:}term' \times term'$
$\gg G[addition\_term^0(b')/t]\ [ext\ d_1]$
$q{:}term^0 \rightarrow U_1, h{:}t{:}term' \rightarrow G, b{:}term'$
$\gg G[any\_term^0(b')/t]\ [ext\ d_2]$
$q{:}term^0 \rightarrow U_1, h{:}t{:}term' \rightarrow G, b{:}term' \times term'$
$\gg G[application\_term^0(b')/t]\ [ext\ d_3]$
$q{:}term^0 \rightarrow U_1, h{:}t{:}term' \rightarrow G, b{:}term',$
$\gg G[atom\_term^0(b')/t]\ [ext\ d_4]$
$\gg G[atom\_type\_term^0/t]\ [ext\ d_5]$
$\vdots$

where $term' := \{t{:}term^0 \mid q(t)\}$.

Redicies

42. $term\_ind^0(addition\_term^0(a); h, b.d_1; \cdots; d_{38}) \mapsto_h$
$d_1[\lambda z.term\_ind^0(z; h, b.d_1; \cdots; d_{38}), a/h, b]$

43. $term\_ind^0(any\_term^0(a); h, b.d_1; \cdots; d_{38}) \mapsto_h$
$d_2[\lambda z.term\_ind^0(z; h, b.d_1; \cdots; d_{38}), a/h, b]$

44. $term\_ind^0(application\_term^0(a); h, b.d_1; \cdots; d_{38}) \mapsto_h$
$d_3[\lambda z.term\_ind^0(z; h, b.d_1; \cdots; d_{38}), a/h, b]$

etc.

B.4 Rule

The Nuprl logic contains a large number of rules of inference. To present the formalization of every Nuprl rule, would take many times the length of appendix A. Thus, we present here only those rules for the Nuprl product type. The pattern of the other rules should be apparent from these. Along with the clauses formalizing each rule, we have listed the reference type and number of the rule as given in appendix A.[1]

Formation

1. $H \gg rule^0 \in U_i$

Canonical

2. (*product 1*)
   $H \gg universe\_intro\_product^0(a) \in rule^0$ *by intro*
   $\gg a \in term^0 \times ident$
3. (*product 2*)
   $H \gg product\_equality^0(a) \in rule^0$ *by intro*
   $\gg a \in ident$
4. (*product 5*)
   $H \gg product\_intro^0(a) \in rule^0$ *by intro*
   $\gg a \in term^0 \times \mathbf{N}^+ \times ident$
5. (*product 6*)
   $H \gg product\_equality\_pair^0(a) \in rule^0$ *by intro*
   $\gg a \in \mathbf{N}^+ \times ident$
6. (*product 9*)
   $H \gg product\_elim^0(a) \in rule^0$ *by intro*
   $\gg a \in ident \times ident \times ident \times ident$
7. (*product 10*)
   $H \gg product\_equality\_spread^0(a) \in rule^0$ *by intro*
   $\gg a \in (ident \times term^0) \times term^0 \times ident \times ident \times ident$

[1]The naming convention used here for the names of canonical terms of type $rule^0$ is the same as that used in the procedural metalanguage except that independent versus dependent suffixes have been omitted (see chapter 8 of Constable et al. [41]).

Noncanonical

8. $H \gg rule\_case^{0}(r; a.d_1; d_2; d_3; \cdots) \in G$ *by intro new* x
   $\gg r \in rule^{0}$
   $a{:}term^{0} \times ident, x{:}(r = universe\_intro\_product^{0}(a) \in rule^{0}) \gg d_1 \in G$
   $a{:}ident, x{:}(r = product\_equality^{0}(a) \in rule^{0}) \gg d_2 \in G$
   $a{:}term^{0} \times \mathrm{N}^{+} \times ident, x{:}(r = product\_intro^{0}(a) \in rule^{0}) \gg d_3 \in G$
   $\vdots$
9. $H, r{:}rule, H' \gg G$ *by elim* r *new* a, x
   $a{:}term^{0} \times ident, x{:}(r = universe\_intro\_product^{0}(a) \in rule^{0}) \gg G$
   $a{:}ident, x{:}(r = product\_equality^{0}(a) \in rule^{0}) \gg G$
   $a{:}term^{0} \times \mathrm{N}^{+} \times ident, x{:}(r = product\_intro^{0}(a) \in rule^{0}) \gg G$
   $\vdots$

Redicies

10. $rule\_case^{0}(universe\_intro\_product^{0}(a); b.d_1; d_2; d_3; \cdots) \mapsto_h d_1[a/b]$
11. $rule\_case^{0}(product\_equality^{0}(a); b.d_1; d_2; d_3; \cdots) \mapsto_h d_2[a/b]$
12. $rule\_case^{0}(product\_intro^{0}(a); b.d_1; d_2; d_3; \cdots) \mapsto_h d_3[a/b]$

etc.

B.5 Applies and Subgoals

Formation

1. $H \gg applies^{o}(s,r) \in U_i$ *by intro*
   $\gg s \in sequent^{o}$
   $\gg r \in rule^{o}$
2. $H \gg subgoals^{o}(s,r) \in sequent^{o}$ *list by intro*
   $\gg s \in sequent^{o}$
   $\gg r \in rule^{o}$
   $\gg applies^{o}(s,r)$
3. $H \gg applies^{o}(s,r) \vee \neg applies^{o}(s,r)$
   by decidable [*ext* $decide\_applies^{o}(s,r)$]
   $\gg s \in sequent^{o}$
   $\gg r \in rule^{o}$

Redicies

4. $applies^{o}(s, universe\_intro\_product^{o}(d)) \mapsto_h$
   let $c,h = s$ *and* $A,x = d$ *in*
   $is\_universe^{o}(c)$ &
   $fresh^{o}(x,h)$
5. $subgoals^{o}(s, universe\_intro\_product^{o}(d)) \mapsto_h$
   let $c,h = s$ *and* $A,x = d$ *in*
   $[\langle h, \ulcorner \wedge A \in \wedge c \urcorner\rangle; \langle h + x{:}A, c\rangle]$
6. $applies^{o}(s, product\_equality^{o}(d)) \mapsto_h$
   let $c,h = s$ *in*
   $is\_membership^{o}(c)$ *cand*
   $is\_product^{o}(element\_of\_member^{o}(c))$ &
   $is\_universe^{o}(type\_of\_member^{o}(c))$ &
   $fresh^{o}(d,h)$
7. $subgoals^{o}(s, product\_equality^{o}(d)) \mapsto_h$
   let $c,h = s$ *and* $\ulcorner \wedge x{:}\wedge A \times \wedge B \in \wedge Ui \urcorner = c$ *in*
   $[\langle h, \ulcorner \wedge A \in \wedge Ui \urcorner\rangle; \langle h + x{:}A, \ulcorner \wedge B[\wedge d/\wedge x] \in \wedge Ui \urcorner\rangle]$

8. $applies^0(s, product\_intro^0(d)) \mapsto_h$
$\quad let\ c, h = s\ in$
$\quad\quad is\_product^0(c)\ \&$
$\quad\quad fresh^0(snd(snd(d)), h)$

9. $subgoals^0(s, product\_intro^0(d)) \mapsto_h$
$\quad let\ c, h = s\ and\ a, i, y = d\ and$
$\quad\quad \ulcorner \wedge x{:}\wedge A \times \wedge B \urcorner = c\ in$
$\quad\quad\quad [\langle h, \ulcorner \wedge a \in \wedge A \urcorner \rangle;$
$\quad\quad\quad \langle h, \ulcorner \wedge B[\wedge a / \wedge x] \rangle;$
$\quad\quad\quad \langle h + x{:}A, \ulcorner \wedge B[\wedge y / \wedge x] \in \wedge universe\_term^0(i) \urcorner \rangle]$

10. $applies^0(s, product\_equality\_pair^0(d)) \mapsto_h$
$\quad let\ c, h = s\ in$
$\quad\quad is\_membership^0(c)\ cand$
$\quad\quad\quad is\_pair^0(element\_of\_membership^0(c)\ \&$
$\quad\quad\quad is\_product^0(type\_of\_membership^0(c)\ \&$
$\quad\quad\quad fresh^0(snd(d), h)$

11. $subgoals^0(s, product\_equality\_pair^0(d)) \mapsto_h$
$\quad let\ c, h = s\ and\ i, y = d\ and$
$\quad\quad \ulcorner \langle \wedge a, \wedge b \rangle \in \wedge x{:}\wedge A \times \wedge B \urcorner = c\ in$
$\quad\quad\quad [\langle h, \ulcorner \wedge a \in \wedge A \urcorner \rangle;$
$\quad\quad\quad \langle h, \ulcorner \wedge b \in \wedge B[\wedge a / \wedge x] \urcorner \rangle;$
$\quad\quad\quad \langle h + y{:}A, \ulcorner \wedge B[\wedge d / \wedge x] \in \wedge universe\_term^0(i) \urcorner \rangle]$

12. $applies^0(s, product\_elim^0(d)) \mapsto_h$
$\quad let\ c, h = s\ and\ z, u, v, w = d\ in$
$\quad\quad declared^0(z, h)\ cand$
$\quad\quad\quad is\_product^0(declaration^0(z, h))\ \&$
$\quad\quad\quad distinct^0([u; v; w])\ \&$
$\quad\quad\quad fresh^0(u, h)\ \&\ fresh^0(v, h)\ \&\ fresh^0(w, h)$

13. $subgoals^0(s, product\_elim^0(d)) \mapsto_h$
$\quad let\ c, h = s\ and\ z, u, v, w = d\ and$
$\quad\quad t = declaration^0(z, h)\ and$
$\quad\quad \ulcorner \wedge x{:}\wedge A \times \wedge B \urcorner = t\ in$
$\quad\quad\quad [\langle h + u{:}A + v{:}B[u/x] + w{:}\ulcorner \wedge z = \langle \wedge u, \wedge v \rangle \in \wedge t \urcorner,$
$\quad\quad\quad\quad c[pair\_term^0(u, v)/z] \rangle]$

14. $applies^{O}(s, product\_equality\_spread^{O}(d) \mapsto_h$
$let\ c, h = s\ and\ \langle z, T\rangle, p, u, v, a = d\ in$
$is\_membership^{O}(c)\ cand$
$let\ \ulcorner spread(\wedge e; \wedge x; \wedge y.\wedge t)\urcorner = element\_of\_member^{O}(c)\ in$
$instance^{O}(type\_of\_member^{O}(c), \langle z, T\rangle)\ \&$
$is\_product^{O}(p)\ \&$
$distinct^{O}([u; v; a])\ \&$
$fresh^{O}(u, h)\ \&\ fresh^{O}(v, h)\ \&\ fresh^{O}(a, h)$

15. $subgoals^{O}(s, product\_equality\_spread^{O}(d)) \mapsto_h$
$let\ c, h = s\ and\ \langle z, T\rangle, p, u, v, a = d\ and$
$\ulcorner \wedge w{:}\wedge A \times \wedge B\urcorner = p\ and$
$\ulcorner spread(\wedge e; \wedge x; \wedge y.\wedge t)\urcorner = element\_of\_member^{O}(c)\ in$
$[\langle h, \ulcorner \wedge e \in \wedge p\urcorner\rangle;$
$\langle h + u{:}A + v{:}B[u/w] + a{:}(\ulcorner \wedge e = \langle \wedge u, \wedge v\rangle \in \wedge p\urcorner),$
$\ulcorner \wedge t[\wedge u, \wedge v/\wedge x, \wedge y] \in \wedge T[\langle \wedge u, \wedge v\rangle/\wedge z]\urcorner\rangle]$

B.6 Proof

Formation

1. $H \gg proof^0 \in U_i$ *by intro*

Canonical

2. $H \gg \langle s, r, c \rangle \in proof^0$ *by intro*
$\quad \gg s \in sequent^0$
$\quad \gg r \in rule^0$
$\quad \gg c \in proof^0\ list$
$\quad \gg applies^0(s, r)$
$\quad \gg subgoals^0(s, r) = goals(c) \in sequent^0\ list$

Noncanonical

3. $H \gg proof\_ind^0(p; h, z.d) \in G[p/v]$ *by intro using* $v.G$ *new* q, h', z'
$\quad \gg p \in proof^0$
$\quad q{:}proof^0 \rightarrow U_1, h'{:}v{:}proof' \rightarrow G,$
$\quad\quad z'{:}\{s{:}sequent^0 \times r{:}rule^0 \times c{:}proof'\ list \mid applies^0(s, r)\ cand$
$\quad\quad\quad subgoals^0(s, r) = goals(c) \in sequent^0\ list\}$
$\quad\quad\quad\quad \gg d[h', z'/h, z] \in G[z'/v]$

where $proof' = \{p{:}proof^0 \mid q(p)\}$

4. $H, p{:}proof^0, H' \gg G$ *by elim* p *new* q, h, z
$\quad q{:}proof^0 \rightarrow U_1, h{:}p{:}proof' \rightarrow G,$
$\quad\quad z{:}\{s{:}sequent^0 \times r{:}rule^0 \times c{:}proof'\ list \mid applies^0(s, r)\ cand$
$\quad\quad\quad subgoals^0(s, r) = goals(c) \in sequent^0\ list\} \gg G[z/p]$

where $proof' = \{p{:}proof^0 \mid q(p)\}$.

Redicies

5. $\textit{proof\_ind}^{o}(\langle s, r, c\rangle; h, z.d) \mapsto_h d[\lambda p.\textit{proof\_ind}^{o}(p; h, z.d), \langle s, r, c\rangle / h, z]$

Bibliography

[1] Luigia Aiello and Richard W. Weyhrauch. Using meta-theoretic reasoning to do algebra. In W. Bibel and R. Kowalski, editors, *5th Conference on Automated Deduction*, pages 1–13, Springer-Verlag, Berlin, 1980.

[2] Mario Aiello and Richard W. Weyhrauch. Checking proofs in the metamathematics of first order logic. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, pages 1–8, 1975.

[3] Stuart Allen. November 1987. Private communication.

[4] Stuart Allen. A non-type-theoretic definition of Martin-Löf's types. In *Symposium on Logic in Computer Science*, pages 215–221, 1987.

[5] Stuart Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. Ph.D. thesis, Cornell University, 1987.

[6] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, Orlando, Florida, 1986.

[7] Peter B. Andrews. Theorem proving via general matings. *Journal of the Association for Computing Machinery*, 28(2):193–214, 1981.

[8] Peter B. Andrews. Transforming matings into natural deduction proofs. In W. Bibel and R. Kowalski, editors, *5th Conference on Automated Deduction*, pages 281–292, Springer-Verlag, Berlin, 1980.

[9] Arnon Avron, Furio A. Honsell, and Ian A. Mason. *Using Typed Lambda Calculus to Implement Formal Systems on a Machine*. LFCS Report Series ECS–LFCS–87–31, Department of Computer Science, University of Edinburgh, 1987.

[10] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–624, 1978.

[11] Joseph L. Bates. *A Logic For Correct Program Development.* Ph.D. thesis, Cornell University, 1979.

[12] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.

[13] John L. Bell and Moshé Machover. *A Course in Mathematical Logic.* North-Holland, Amsterdam, 1977.

[14] W. Bibel and R. Kowalski, editors. *5th Conference on Automated Deduction*, Springer-Verlag, Berlin, 1980.

[15] W. W. Bledsoe and Peter Bruell. A man-machine theorem-proving system. *Artificial Intelligence*, 5:51–72, 1974.

[16] George S. Boolos and Richard C. Jeffrey. *Computability and Logic.* Cambridge University Press, Cambridge, second edition, 1980.

[17] Kenneth A. Bowen and Robert A. Kowalski. *Amalgamating Language and Metalanguage in Logic Programming*, pages 153–172. *A.P.I.C. Studies in Data Processing* volume 16, Academic Press, New York, 1982.

[18] R. S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, chapter 3, Academic Press, Orlando, Florida, 1981.

[19] Robert S. Boyer and J Strother Moore. *A Computational Logic. ACM Monograph Series*, Academic Press, New York, 1979.

[20] Frank Malloy Brown. An investigation into the goals of research in automatic theorem proving as related to mathematical reasoning. *Artificial Intelligence*, 14:221–242, 1980.

[21] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, London, 1983.

[22] Alan Bundy and Bob Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. In W. Bibel and R. Kowalski, editors, *5th Conference on Automated Deduction*, pages 24–38, Springer-Verlag, Berlin, 1980.

[23] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[24] Luca Cardelli. *A Polymorphic λ-calculus with Type:Type.* Technical Report, Digital Equipment Corporation, Systems Research Center, 1986. In preparation.

[25] Luca Cardelli. Typechecking dependent types and subtypes (draft). January 1987. Unpublished manuscript.

[26] K. L. Clark and S. -A. Tärnlund, editors. *Logic Programming. A.P.I.C. Studies in Data Processing* volume 16, Academic Press, Orlando, Florida, 1982.

[27] Rance Cleaveland and Prakash Panangaden. *Type Theory and Concurrency.* Technical Report 85–714, Department of Computer Science, Cornell University, 1985.

[28] Walter Rance Cleaveland II. *Type-Theoretic Models of Concurrency.* Ph.D. thesis, Cornell University, 1987.

[29] R. L. Constable, S. D. Johnson, and C. D. Eichenlaub. *An Introduction to the PL/CV2 Programming Logic. Lecture Notes in Computer Science* volume 135, Springer-Verlag, Berlin, 1982.

[30] Robert L. Constable. *Constructive Mathematics as a Programming Logic I: Some Principles of Theory.* Technical Report 83–554, Computer Science Department, Cornell University, 1983.

[31] Robert L. Constable. *Intensional Analysis of Functions and Types.* Internal Report CSR-118-82, University of Edinburgh, Department of Computer Science, 1982.

[32] Robert L. Constable. *Mathematics as Programming.* Technical Report 83–565, Computer Science Department, Cornell University, 1983.

[33] Robert L. Constable. *Programs as Proofs.* Technical Report 82-532, Computer Science Department, Cornell University, 1982.

[34] Robert L. Constable. *Universally Closed Classes of Total Computable Functions.* Technical Report TR 84–640, Cornell University, Department of Computer Science, 1984.

[35] Robert L. Constable and Joseph L. Bates. *The Nearly Ultimate PEARL.* Technical Report 83–551, Computer Science Department, Cornell University, 1983.

[36] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1:285–326, 1985.

[37] Robert L. Constable and Nax P. Mendler. Recursive definitions in type theory. In Rohit Parikh, editor, *Logics of Programs*, pages 61–78, Springer-Verlag, Berlin, 1985.

[38] Robert L. Constable and Michael J. O'Donnell. *A Programming Logic*. Winthrop, Cambridge, Massachusetts, 1978.

[39] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *Symposium on Logic in Computer Science*, pages 183–193, 1987.

[40] Robert L. Constable and Daniel R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, 1984.

[41] Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[42] Thierry Coquand and Gérard Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics*. Technical Report 401, INRIA, 1985.

[43] Thierry Coquand and Gérard Huet. A theory of constructions. 1984. Unpublished manuscript.

[44] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*. Volume 1, North-Holland, Amsterdam, 1958.

[45] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*. Volume 2, North-Holland, Amsterdam, 1972.

[46] M. Davis. The prehistory and early history of automated deduction. In Jörg Siekmann and Graham Wrightson, editors, *Automation of Reasoning, Classical Papers on Computational Logic 1957–1966*, pages 1–28, Springer-Verlag, Berlin, 1983.

[47] Martin Davis and Jacob T. Schwartz. Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, 5:217–230, 1979.

[48] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, Academic Press, Orlando, Florida, 1980.

[49] Nachum Dershowitz. Program abstraction and instantiation. *ACM Transactions on Programming Languages and Systems*, 7(3):446–477, 1985.

[50] Michael Dummett. *Elements of Intuitionism.* Clarendon Press, Oxford, 1977.

[51] Peter Dybjer, Bengt Nordström, Kent Peterson, and Jan Smith, editors. *Proceedings of the Workshop on Specification and Derivation of Programs*, Programming Methodology Group, Department of Computer Science, Chalmers University of Technology and University of Göteborg, 1985. Report 18.

[52] Jon Fairbairn. A new type-checker for a functional language. *Science of Computer Programming*, 6:273–290, 1986.

[53] Solomon Feferman. Inductively presented systems and the formalization of meta-mathematics. In D. van Dalen, D. Lascar, and J. Smiley, editors, *Logic Colloquium '80*, pages 95–128, North-Holland, Amsterdam, 1982.

[54] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, 1969.

[55] C. A. Goad. Proofs as descriptions of computation. In W. Bibel and R. Kowalski, editors, *5th Conference on Automated Deduction*, pages 39–52, Springer-Verlag, Berlin, 1980.

[56] Christopher Alan Goad. *Computational Uses of the Manipulation of Formal Proofs.* Ph.D. thesis, Stanford University, 1980.

[57] Kurt Gödel. On formally undecidable propositions of Principia Mathematica and related systems I. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 596–616, Harvard University Press, Cambridge, Massachusetts, 1967.

[58] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation. Lecture Notes in Computer Science* volume 78, Springer-Verlag, Berlin, 1979.

[59] Timothy G. Griffin. *An Environment for Formal Systems.* Technical Report 87–846, Computer Science Department, Cornell University, 1987.

[60] Timothy G. Griffin. *A Formal Account of Notational Definition.* Technical Report, Cornell University, 1987.

[61] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, 1987.

[62] Robert Harper, David MacQueen, and Robin Milner. *Standard ML.* LFCS Report Series ECS–LFCS–86–2, Department of Computer Science, University of Edinburgh, March 1986.

[63] Robert Willam Harper, Jr. *Aspects of the Implementation of Type Theory.* Ph.D. thesis, Cornell University, 1985.

[64] J. Hartmanis. *On Goedel Speed-Up and Succinctness of Language Representation.* Technical Report TR 82-485, Computer Science Department, Cornell University, 1982.

[65] D. Hilbert and W. Ackermann. *Principles of Mathematical Logic.* Chelsea, New York, second edition, 1950. English translation.

[66] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490, Academic Press, Orlando, Florida, 1980.

[67] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* Ph.D. thesis, Cornell University, 1987.

[68] Douglas J. Howe. The computational behaviour of Girard's paradox. In *Symposium on Logic in Computer Science*, pages 205–214, 1987.

[69] Douglas J. Howe. *Implementing Number Theory: An Experiment with Nuprl.* Technical Report TR 86–752, Department of Computer Science, Cornell University, 1986.

[70] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[71] Jussi Ketonen and Joseph S. Weening. *EKL—An Interactive Proof Checker User's Reference Manual.* Technical Report STAN-CS-84-1006, Department of Computer Science, Stanford University, 1984.

[72] Jussi Ketonen and Joseph S. Weening. *The Language of an Interactive Proof Checker*. Technical Report STAN-CS-83-992, Department of Computer Science, Stanford University, 1983.

[73] Stephen C. Kleene. *Introduction to Metamathematics*. van Nostrand, Princeton, 1952.

[74] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Symposium on Logic in Computer Science*, pages 237–248, 1986.

[75] Christoph Kreitz. *Implementing Automata Theory*. Technical Report, Department of Computer Science, Cornell University, 1986.

[76] Daniel Leivant. Syntactic translations and provably recursive functions. *The Journal of Symbolic Logic*, 50(3):682–688, 1985.

[77] Douglas B. Lenat. Eurisko: a program which learns new heuristics and domain concepts. *Artificial Intelligence*, 21(1):161–198, 1983.

[78] Douglas B. Lenat and Jon Seely Brown. Why AM and Eurisko appear to work. *Artificial Intelligence*, 23(3):269–294, 1984.

[79] M. Löb. Solution of a problem of Leon Henkin. *The Journal of Symbolic Logic*, 20(2):115–118, 1955.

[80] David Lawrence Luner. *A Concurrent Procedure for Unification*. Master's thesis, Syracuse University, 1984.

[81] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, North-Holland, Amsterdam, 1982.

[82] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. F. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, North-Holland, Amsterdam, 1973.

[83] Per Martin-Löf. Intuitionistic type theory. Notes by Giovanni Sambin of a series of lectures given in Padova, June 1980. Unpublished manuscript.

[84] Per Martin-Löf. *Notes on Constructive Mathematics*. Almquist and Wiksell, Stockholm, 1970.

[85] Per Martin-Löf. *A Theory of Types*. Report 71-3, Department of Mathematics, University of Stockholm, February 1971. Revised October 1971.

[86] John McCarthy. Computer programs for checking mathematical proofs. In *Proceedings of the Symposia in Pure Mathematics, Volume V, Recursive Function Theory*, pages 219–228, American Mathematics Society, Providence, Rhode Island, 1962.

[87] Charles McCarty. Intuitionism: an introduction to a seminar. *Journal of Philosophical Logic*, 12(2):105–149, 1983.

[88] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Symposium on Logic in Computer Science*, pages 30–36, 1987.

[89] Paul Francis Mendler. *Inductive Definitions in Type Theory*. Ph.D. thesis, Cornell University, 1987.

[90] Albert R. Meyer, John C. Mitchell, Eugenio Moggi, and Richard Statman. Empty types in polymorphic lambda calculus (preliminary report). In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 253–262, 1987.

[91] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Symposium on Logic in Computer Science*, pages 98–105, 1987.

[92] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. In *Proceedings of the 12th ACM Symposium on Principles of Programing Languages*, 1985.

[93] Ketan Mulmuley. *Full Abstraction and Semantic Equivalence*. Ph.D. thesis, Carnegie-Mellon University, 1985.

[94] Ketan Mulmuley. *A Mechanizable Theory for Existence Proofs of Inclusive Predicates*. Technical Report CMU-CS-85-149, Carnegie-Mellon University, Department of Computer Science, 1985.

[95] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, 1980.

[96] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[97] Bengt Nordström. *Martin-Löf's Type Theory as a Programming Logic.* Technical Report 27, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, 1986.

[98] Bengt Nordstrom. Programming in constructive set theory: some examples. In *Proceedings 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 290–341, Portsmouth, England, 1981.

[99] Prakash Panangaden and Michael I. Schwartzbach. *Categorical Type Theory.* Technical Report 85–716, Computer Science Department, Cornell University, 1985.

[100] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[101] Lawrence Paulson. *Verifying the Unification Algorithm in LCF.* Technical Report 50, Computer Laboratory, University of Cambridge, 1984.

[102] Dag Prawitz. Ideas and results in proof theory. In *Studies in Logic and the Foundations of Mathematics*, pages 235–307, North-Holland, Amsterdam, 1971.

[103] Tomas Reps and Bowen Alpern. Interactive proof checking. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1984.

[104] James Toshio Sasaki. *Extracting Efficient Programs from Constructive Proofs.* Ph.D. thesis, Cornell University, 1986.

[105] William L. Scherlis and Dana S. Scott. First steps towards inferential programming. In R.E.A. Mason, editor, *Information Processing 83*, pages 199–211, Elsevier Science Publishers, 1983.

[106] David A. Schmidt and Jacek Leszczylowski. *On Developing A Logic For Program Derivation and Verification.* Technical Report 86–16, Computer Science Department, Iowa State University, 1986.

[107] Michael I. Schwartzbach. *A Category Theoretic Analysis of Predicative Type Theory.* Ph.D. thesis, Cornell University, 1986.

[108] Dana S. Scott. Constructive validity. In *Symposium on Automatic Demonstration*, pages 237–275, Spinger-Verlag, Berlin, 1970.

[109] N. Shankar. *A Mechanical Proof of the Church-Rosser Theorem.* Technical Report 45, Institute for Computing Science. The University of Texas at Austin, 1984.

[110] N. Shankar. *Towards Mechanical Metamathematics.* Technical Report 43, University of Texas at Austin, Institute for Computing Science, 1984.

[111] Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Massachusetts, 1967.

[112] Jörg Siekmann and Graham Wrightson, editors. *Automation of Reasoning, Classical Papers on Computational Logic 1957–1966.* Volume 1, Springer-Verlag, Berlin, 1983.

[113] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language.* Ph.D. thesis, Massachusetts Institute of Technology, 1982.

[114] C. Smoryński. *Self-Reference and Modal Logic.* Springer-Verlag, Berlin, 1985.

[115] Stefan Sokolowski. *A Note on Tactics in LCF.* Internal Report CSR–140–83, Department of Computer Science, University of Edinburgh, 1983.

[116] Ryan Stansifer. *Representing Constructive Theories in High-Level Programming Languages.* Ph.D. thesis, Cornell University, 1985.

[117] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

[118] Carolyn Talcott. *The Essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation.* Ph.D. thesis, Stanford University, 1985.

[119] A. Tarski. *Logic, Semantics, Metamathematics, Papers from 1923 to 1938.* Clarendon Press, Oxford, 1956.

[120] A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics* volume 344, Springer-Verlag, Berlin, 1973.

[121] Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931.* Harvard University Press, Cambridge, Massachusetts, 1967.

[122] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *ACM Symposium on LISP and Functional Programming*, 1986.

[123] Richard W. Weyhrauch. *Prolegomena to a Theory of Formal Reasoning.* Technical Report STAN-CS-78-687, Stanford University, Computer Science Department, 1978.

[124] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[125] Wu Wen-Tsun. Basic principles of mechanical theorem proving in elementary geometries. *Journal of Automated Reasoning*, 2:221–252, 1986.