Implementing Mathematics with The Nuprl Proof Development System

Article · January 1996

Source: CiteSeer

CITATIONS
883

READS
1,152

12 authors, including:

Robert Constable
Cornell University
184 PUBLICATIONS **3,423** CITATIONS
SEE PROFILE

Robert Harper
Carnegie Mellon University
91 PUBLICATIONS **6,185** CITATIONS
SEE PROFILE

Prakash Panangaden
McGill University
239 PUBLICATIONS **7,182** CITATIONS
SEE PROFILE

Implementing Mathematics

with

The Nuprl Proof Development System

Draft of 18 October 1995

By the PRL Group:

R. L. Constable
S. F. Allen
H. M. Bromley
W. R. Cleaveland
J. F. Cremer
R. W. Harper
D. J. Howe
T. B. Knoblock
N. P. Mendler
P. Panangaden
J. T. Sasaki
S. F. Smith

Computer Science Department
Cornell University
Ithaca, NY 14853

This research supported in part by the National Science Foundation under grant DCR83-03327.

Contents

Part I: Tutorial

Part III: Advanced

Preface

We hope to accomplish four things by writing this book. Our first goal is to offer a tutorial on the new mathematical ideas which underlie our research. In doing so we have tried to provide several entry points into the material, even at the cost of considerable redundancy. We hope that many of the ideas will be accessible to a well–trained undergraduate with a good background in mathematics and computer science. Second, parts of this book should serve as a manual for users of the Nuprl system (pronounced "new pearl"). As the system has grown so has the demand, both here at Cornell and at other institutions, for better documentation. We have tried to collect here all material relevant to the operation of the system. Third, we give an overview of our project for those interested in applications of the results and for those inclined to basic research in the area. Finally, we present new research which has arisen as we have worked on the Nuprl system. This system embodies contributions to the foundations of computer science and semantics, to automated reasoning and to system design, and it has shown promise as an intelligent system.

Authoring this book was a collective task; many individual efforts found their ways into these pages. Most chapters have several authors. That we were able to proceed in this fashion owes to the fact that we have assembled at Cornell a unique group of computer scientists who had worked for more than two years on the project.

We would like to acknowledge warmly the special contributions of Joseph L. Bates to this enterprise. Joe has been a chief architect of this system and was a major force in creating the PRL project, from which the system emerged. This manuscript is replete with Joe's ideas. We also want to thank Evan Steeg, who joined the project as an undergraduate and has since contributed significantly to our efforts, especially in the area of writing tactics. Tim Griffin's implementation of the rules for recursive types and partial functions is greatly appreciated as is his contribution of a new interface to the evaluator. All of our efforts are built on the contributions of Fran Corrado, our only full–time programmer. We also thank Christoph Kreitz, who has used the system extensively and provided us with insightful reports on its strengths and weaknesses. We also appreciate the constructive criticism of James Hook, who spent many hours reading and discussing early drafts. We thank Ryan Stansifer, Hal Perkins, and Aleta Ricciardi for proof–reading. Finally, we thank Donette Isenbarger, Michele Fish and Dawn Hall for their conscientious assistance in preparing the manuscript.

We appreciate the help we have received from the National Science Foundation through a series of grants supporting the Nuprl project, the Cor-

nell Computer Science Department computing facility, and various pieces of equipment needed for this research (MCS80-03349, MCS81-04018, DCR80-03327, DCR81-05763, and DRC84-06052). The NSF has also supported Mr. Cleaveland and Mr. Smith with fellowships. We also appreciate the generous support of IBM, whose fellowships have supported Mr. Mendler. Finally, we acknowledge the support of Cornell University and especially our department for providing fellowships, matching funds, and the environment to make this work possible.

This is the first version of the manual. We consider it to be a preliminary effort and welcome timely comments that will help us improve the presentation. The manual describes version 1.0 of Nuprl.

Robert L. Constable
for the PRL Group
Ithaca, NY 1985

Chapter 1

Overview

1.1 Brief Description of Nuprl

Problem solving is a significant part of science and mathematics and is the most intellectually significant part of programming. Solving a problem involves understanding the problem, analyzing it, exploring possible solutions, writing notes about intermediate results, reading about relevant methods, checking results, and eventually assembling a solution. Nuprl is a computer system which provides assistance with this activity. It supports the interactive creation of proofs, formulas, and terms in a formal theory of mathematics; with it one can express concepts associated with definitions, theorems, theories, books and libraries. Moreover, the theory is sensitive to the computational meaning of terms, assertions and proofs, and the system can carry out the actions used to define that computational meaning. Thus Nuprl includes a programming languages, but in a broader sense it is a system for implementing mathematics.

1.2 Highlights of Nuprl

One of the salient features of Nuprl is that the logic and the system take account of the computational meaning of assertions and proofs. For instance, given a constructive existence proof the system can use the computational information in the proof to build a representation of the object which demonstrates the truth of the assertion. Such proofs can thus be used to provide data for further computation or display. Moreover, a proof that for any object x of type A we can build an object y of type B satisfying relation $R(x, y)$ implicitly defines a computable function f from A to B. The system can build f from the proof, and it can evaluate f on inputs of type A. For example, given any mapping f of a nonempty set A onto a finite set B of

smaller but nonzero cardinality, one can say that there will be two points of A mapped to the same point of B. From a proof of this statement the system can extract a function which given specific A, B and f produces two points of A mapped to the same point of B. This function expresses the computational content of the theorem and can be evaluated.

As a computer system Nuprl supports an interactive environment for text editing, proof generation and function evaluation. The interaction is oriented around a computer terminal with a screen and a mouse, for our intention is to provide a medium for doing mathematics different from that provided by paper and blackboard. Eventually such a medium may support a variety of input devices and may provide communication with other users and systems; the essential point, however, is that this new medium is active, whereas paper, for example, is not. This enables the interactive style of proof checking that characterizes Nuprl; in this system it is impossible to develop an incorrect proof.

Nuprl also possesses some of the characteristics of an intelligent computer system in that it provides its users with a facility for writing proof–generating programs in a metalanguage, ML. The implementation of the logic codes into Nuprl certain primitive mathematical knowledge in the form of rules for generating proofs and in the form of certain defined types in ML. As people use Nuprl they create libraries of mathematical facts, definitions and theorems; they can also create libraries of ML programs which use these results and other ML programs to generate proofs of theorems. In a very real sense, as Nuprl is used its capacity for providing help in proving theorems increases. By virtue of this property, Nuprl possesses aspects of an intelligent system.

The system design exhibits several key characteristics. The style of the logic is based on the stepwise refinement paradigm for problem solving in that the system encourages the user to work backward from goals to subgoals until one reaches what is known. Indeed, the system derives its name, Proof Refinement Logic,[1] from this method of presentation. The logic has a constructive semantics in that the meaning of propositions is given by rules of use and in terms of computation. We discuss these features in more detail later.

In a larger sense the Nuprl system serves as a tool for experimenting with ways of applying computer power to solving problems and to generating exact explanations of solutions, especially in the realm of computational mathematics. Because the difficult part of computer programming is precisely in problem solving and in explaining algorithmic solutions, we think that a system of this kind will eventually have a lasting impact on our ability to produce reliable and understandable programs.

[1] Nuprl is version "nu" of our Proof Refinement Logics. Earlier versions were "micro" and "lambda" [Nuprl Staff 83].

1.3 Motivations

In high school algebra solutions to most problems can be checked by simple computation; for instance, one can easily verify by substitution and reduction that 17 is a root of $x^3 - 16x^2 - 19x + 34$. Those who are pleased with the certainty of such solutions may also hope to find ways of checking solutions to more abstract problems (such as showing that $\sqrt{2}^{\sqrt{2}}$ is irrational) computationally. The idea that computers can check proofs is a step toward achieving this ambition [McCarthy 62, deBruijn 80], and there are several interesting implementations of this idea [Suppes 81, Weyhrauch 80, Constable, Johnson, & Eichenlaub 82, Gordon, Milner, & Wadsworth 79]. The computer system described here represents another approach to this problem.

Someone who has struggled for hours or perhaps days to untangle the details of a complex mathematical argument will understand the dream of building a computer system which helps *fill in the details* and keeps track of the proof obligations that must be met at the critical points of an argument. There are computer systems which do this, and it is possible in principle to solve problems in a system of this kind which are beyond the patience and capability of an unaided human being, as K. Mulmuley has already demonstrated in LCF [Mulmuley 84]. Nuprl uses some of the mechanisms of LCF which make this possible, namely the metalanguage ML in which the user writes programs to provide help with the details.

People who have spent hours or days getting a mathematical construction such as Gauss' construction of a regular 17–gon exactly right will know the desire to watch this mathematical construction "come to life." Nuprl was built to meet such aspirations. Every construction described in Nuprl's mathematics can, in principle, be executed mechanically. In particular, Nuprl can execute functions and thus serve as a programming language in the usual sense.

Anyone who has tried to write a mathematical paper or system consisting of several interacting theorems and constructions will appreciate having a uniform notation for discourse and a facility for creating an encyclopedia of results in this notation. The Bourbaki effort [Bourbaki 68] manifests such aspirations on a grand scale, and Nuprl is a step toward realizing this goal. It supports a particular mathematical theory, *constructive type theory*, whose primitive concepts can serve as building blocks for nearly any mathematical concept. Thus Nuprl provides a uniform language for expressing mathematics. This is a characteristic that distinguishes Nuprl from most other proof–checking or theorem–generating systems.

Anyone who has written a heuristic procedure such as one for playing games or finding proofs and who has seen it do more than was expected will understand the dream of using a computer system which can provide

unexpected help in proving theorems. It is easy to extrapolate to dreams of machines offering critical help in solving real problems. Substantial effort has been put into "theorem–proving" programs which attempt to provide just such help, usually in a specific domain of mathematics. While Nuprl is not a theorem–proving program in the usual sense, it is a tool for exploring this kind of heuristic programming. Users may express a variety of procedures which search for proofs or attempt to fill in details of a proof. The system has already provided interesting experiences of unexpected behavior by finishing proofs "on its own."

People who have experienced the new electronic medium created by computers may imagine how mathematics will be conducted in it. We see electronic mail and electronic text editing. We see systems like the Cornell Program Synthesizer [Teitelbaum & Reps 81] that help users specify objects such as programs directly in terms of their structure, leaving the system to generate the textual description. We can imagine these ideas being applied to the creation of mathematical objects such as functions and proofs and their textual descriptions. In this case, because mathematical structure can be extremely complex, one can imagine the computer providing considerable assistance in producing objects as well as help in displaying the text and giving the reader access to the underlying structure in various ways. Nuprl offers such capabilities, the most striking among which are the help the system provides in writing formal text (see chapter 3) and in reading and generating highly structured objects such as proofs and function terms. As with all aspects of Nuprl there is much to be done to achieve the goals that motivated the design, but in every case the system as it stands makes a contribution. We hope it will show the way to building better systems of this kind.

1.4 The Nuprl Logical Language

An algorithm to add two integers can be expressed in Nuprl as $\backslash x.\backslash y.x + y$. In general, if $b(x)$ is an expression such as $x * 2$ or $\backslash y.x + y$ in the variable x, then $\backslash x.b(x)$ is the function of one argument which on input a computes the value $b(a)$, where $b(a)$ stands for the expression b with each occurrence of x replaced by a. Application of the function is expressed by $(\backslash x.b(x))(a) = b(a)$; for example, $(\backslash x.x * 2)(2) = 2 * 2 = 4$. We speak of $\backslash x.b(x)$ as the *abstraction* of $b(x)$ with respect to x. This notation is essentially the same as Church's lambda notation, which is systematized in the *lambda calculus* [Church 51]. The form $\backslash x.b(x)$ is a visual approximation to $\lambda x.b(x)$ of the lambda calculus. Our notation is also very close to that used in ML [Gordon, Milner, & Wadsworth 79] except that type information about the variable can be included in the ML form. This marks a significant departure from the Algol–like languages as well, for in these languages the

type information is given with the parameters of a function. For example, one would write **function**(x:**int**)**int** in the heading of an Algol function definition to show that the function maps integers to integers.

In Nuprl, as in the work of H.B. Curry [Curry, Feys, & Craig 58, Curry, Hindley, & Seldin 72], a type discipline is imposed on algorithms to describe their properties. Thus, when we say that $\backslash x.x + 1$ is a function from integers to integers, we are saying that when an integer n is supplied as an argument to this algorithm, then $n + 1$ will denote a specific integer value. In general, meaning is given to a function term $\backslash x.b(x)$ by saying that it has type $A \rightarrow B$ for some type A called the *domain* and some type B called the *range*. The type $A \rightarrow B$ denotes functions which on input a from A produce a value in B. The fact that the functions always return a value is sometimes emphasized by calling them *total functions*.

The type structure hierarchy of Nuprl resembles that of *Principia Mathematica*, the ancestor of all type theories. The hierarchy manifests itself in an unbounded cumulative hierarchy of universes, U_1, U_2, ..., where by cumulative hierarchy we mean that U_i is in U_{i+1} and that every element of U_i is also an element of U_{i+1}. Universes are themselves types,[2] and every type occurs in a universe. In fact, A is a type if and only if it belongs to a universe. Conversely all the elements of a universe are types.

It is pedagogically helpful to think of the other Nuprl types in five stages of decreasing familiarity. The most familiar types and type constructors are similar to those found in typed programming languages such as Algol 68, ML or Pascal. These include the atomic types `int`, `atom` and `void` along with the three basic constructors: cartesian product, disjoint union and function space.[3] If A and B are types, then so is their cartesian product, A`#`B, their disjoint union, A`|`B, and the functions from A to B, A`->`B.

The second stage of understanding includes the dependent function and dependent product constructors, which are written as x:A`->`B and x:A`#`B, respectively, in Nuprl. The dependent function space is used in AUTOMATH and the dependent product was suggested by logicians studying the correspondence between propositions and types; see Scott [Scott 70] and Howard [Howard 80]. These types will be explained in detail later, but the intuitive idea is simple. In a dependent function space represented by x:A`->`B, the range type, B, is indexed by the domain type, A. This is exactly like the indexed product of a family of sets in set theory, which is usually written as $(\Pi x \in A)B(x)$ (see [Bourbaki 68]). In the dependent product represented by x:A`#`B, the type of the second member of a pair can depend on the value

[2] The concept of a universe in this role, to organize the hierarchy of types, is suggested in [Artin, Grothendieck, & Verdier 72] and was used by Martin-Löf [Martin-Löf 73]. This is a means of achieving a *predicative* type structure as opposed to an *impredicative* one as in Girard [Girard 71] or Reynolds [Reynolds 83].

[3] In the discussion which follows we will use `typewriter` font to signify actual Nuprl text and *mathematical* font for metavariables.

of the first. This is exactly the indexed disjoint sum of set theory, which is usually written as $(\Sigma x \in A)B(x)$ (see [Bourbaki 68]).

The third stage of understanding includes the quotient and set types introduced in [Constable 85]. The set type is written $\{x{:}A \mid B\}$ and allows Nuprl to express the notions of constructive set and of partial function. The quotient type allows Nuprl to capture the idea of equivalence class used extensively in algebra to build new objects.

The fourth stage includes propositions considered as types. The atomic types of this form are written a `=` b `in` A and express the proposition that a is equal to b in type A. A special case of this is a `=` a `in` A, written also as a `in` A. These types embody the propositions–as–types principle discovered by H. B. Curry [Curry, Feys, & Craig 58], W. Howard [Howard 80], N. G. deBruijn [deBruijn 80] and H. Lauchli [Lauchli 70] and used in AUTOMATH [deBruijn 80, Jutting 79]. This principle is also central to P. Martin-Löf's Intuitionistic type theories [Martin-Löf 82, Martin-Löf 73].

The fifth stage includes the recursive types and the type of partial functions as presented by Constable and N. P. Mendler [Constable & Mendler 85]. These are discussed in chapter 12, but they have not been completely implemented so we do not use them in examples taken directly from the computer screen. These are the only concepts in this book not taken directly from the 1984 version of the system.

The logical language is interesting on its own. It is built around tested logical notions from H. Curry, A. Church, N. deBruijn, B. Russell, D. Scott, E. Bishop, P. Martin-Löf and ourselves among others and as such forms part of a well–known and thoroughly studied tradition in logic and philosophy. However, this tradition has not until now entered the realm of usable formal systems and automated reasoning in the same way as the first–order predicate calculus. Thus the design and construction of this logic is a major part of Nuprl, and becoming familiar with it will be a major step for our readers. We recommend in addition to this account the forthcoming book of B. Nordström, K. Petersson and J. Smith, the paper "Constructive Validity" [Scott 70], the paper "Constructive Mathematics and Computer Programming" [Martin-Löf 82] and "Constructive Mathematics as a Programming Logic" [Constable 85]. An extensive guide to the literature appears in the references. We hope that this account will be sufficient to allow readers to use the system, which in the end may be its own best tutor.

1.5 Components of the Nuprl System

General

The Nuprl system has six major components: a window manager, a text editor, a proof editor, a library module, a command module and a func-

tion evaluator. Interaction with Nuprl takes place in the context of three languages—the command language, which is extremely simple, the object language, which is the mathematical language of the system, and the meta-language, which is a programming language with data types referring to the object language. The command language is used to initiate editing, use the library and initiate executions in the object and metalanguages, to name a few of its functions. The object language is a constructive theory of types. The metalanguage is the programming language ML from the Edinburgh LCF system modified to suit Nuprl.

The window manager provides the interface for the interactive creation of certain kinds of linguistic objects called *definitions, theorems, proofs* and *libraries* on a terminal screen. Windows offer views of objects; using these views the user can navigate through the object, stopping to modify it, to copy parts of it to other windows or to insert parts into it from other windows, etc. Three special windows provide for editing theorems, viewing the library and entering commands.

Definitions

The definition facility allows one to develop new notations in the form of templates which can be invoked when entering text. This feature provides a flexible way to introduce new notation. For instance, we might have defined a function, `abs(x)`, which computes the absolute value of a number, but we might like to display it in the text as `|x|`. This can be accomplished by defining a template. The format is `|<x:number>| == abs(<x>)`, where the left–hand side of `==` is the template and the right–hand side is the value (the angle brackets are used to designate the parameter).

Proof Editing

The proof–editing facility supports top–down construction of proofs. Goals are written as *sequents*; they have the form $x_1 : H_1, \ldots, x_n : H_n$ >> G, where the H_i are the hypotheses and G is the conclusion. To prove a goal, the user selects a rule for making progress toward achieving this goal, and the system responds with a list of subgoals. For example, if the goal is prove $A\&B$ and the rule selected is "and introduction," the system generates the following subgoals.

1. prove A
2. prove B

Proofs can be thought of as finite trees, where each node corresponds to the application of a logical rule. A proof can be read to the desired level of detail by descending into subtrees whose structure is interesting and passing over others.

Evaluation

The evaluation mechanism allows us to regard Nuprl as a functional programming language. For example, the multiplication function in Nuprl notation is `\x.\y.(x*y)`, and one can evaluate `(\x.\y.(x*y))(2)(3)` to obtain the value `6`. Of course, one can define much more complex data such as infinite precision real numbers and functions on them such as multiplication (see section 11.6). In this case we might also evaluate the form `(\x.\y.mult(x,y))(e)(pi)(50)`, which might print the first 50 digits of the infinite precision multiplication of the transcendental numbers e and π.

The evaluation mechanism can also use a special form, `term_of(`t`)`, where t is a theorem. The `term_of` operation extracts the constructive meaning of a theorem. Thus if we have proved a theorem named t of the form *for all* x *of type* A *there is a* y *of type* B *such that* $R(x, y)$ then `term_of(`t`)` extracts a function mapping any a in A to a pair consisting of an element from B, say b, and a proof, say p, of $R(a, b)$. By selecting the first element of this pair we can build a function f from A to B such that $R(x, f(x))$ for all x in A. The system also provides a mechanism for naming and storing executable terms in the library.

1.6 Programming Modes

The `term_of` operation enables two new modes of programming in Nuprl in addition to the conventional mode of writing function terms. To program in the first new mode one writes *proof outlines*, p, which contain the computational information necessary for `term_of(`p`)` to be executable. The second mode is a refinement of the first in which complete proofs, cp, are supplied as arguments to `term_of`. In this case one knows that the program meets its specification, so this mode might be called "verified programming".

1.7 Physical Characteristics

The system described here is written in about 60,000 lines of Lisp. It runs in Zetalisp on Symbolics Lisp Machines and in Franz Lisp under Unix 4.2BSD. There are also several thousand lines of ML programs included in the basic system. We have been using it since June 1984 principally to test the ideas behind its design but also to begin accumulating a small library of formal algorithmic mathematics. It can be used as a programming environment, and to a limited extent it has seen such service. But unlike its simpler predecessor, Lambda–prl [Nuprl Staff 83], it has not been provided with a compiler nor an optimizer, so it can be very inefficient. We hope that in due course there will be facilities to make it run acceptably fast. The system is in fact growing, but the major thrust over the next year is to substantially

enhance its deductive power and its user–generated knowledge in the form of libraries of definitions, theorems and proof methods (see section 1.9 for some of our detailed plans).

1.8 The "Feel" of the System

Nuprl is an example of an entity which is more than the sum of its parts. It is more than a proof checker or proof–generating editor. It is more than an evaluator for a rich class of functional expressions and more than a system for writing heuristic procedures to find proofs in a specific foundational theory. The integration of these parts creates a new kind of system. One can sense new possibilities arising from this combination both in the system as it exists now and in its potential for growth.

We find that Nuprl as it runs today serves not only as a new tool for writing programs and program specifications but also as a tool for writing nearly any kind of mathematics. Working in the Nuprl environment results in a distinctive style of mathematics—a readable yet formal algorithmic style. The style in turn suggests new mathematical substance such as our treatment of recursive types and partial functions [Constable & Mendler 85]. Nuprl also offers a coherent way to organize and teach a collection of concepts that are important in computer science.

As we extrapolate the course of Nuprl's development we see the emergence of a new kind of "intelligent" system. The mechanisms for the accumulation of mathematical knowledge in the form of definitions, theorems and proof techniques are already in place, and as more of this knowledge is accumulated the system exhibits a more widely usable brand of formal mathematics. Furthermore, since the tactic mechanism gives the system access to the contents of its libraries, one can envision altering the system so that it generates and stores information about itself. In this sense Nuprl is an embryonic intelligent system.

1.9 This Document

Scope

More than just a user's manual for a specific system, this book serves as an introduction to a very expressive foundational theory for mathematics and computer science, a theory which brings together many diverse ideas in modern computing theory. The book is also an introduction to formal logic in general and to constructive logic in particular, and it introduces new methods of program development and verification.

This book also describes a computer system for doing mathematics. The system provides a medium distinct from the traditional paper and black-

board, an active medium that can detect errors, implement tedious steps of argument or computation and carry out suggestions for building proofs. In Nuprl a new level of precision is made useful.

This book also introduces an ongoing enterprise and a new area of research. It presents a first example of the kind of system that we believe will become significant in the next decade. There is clearly much to be done in this area, and this document will make readers aware of certain open problems and challenging tasks.

Organization

This book includes a tutorial, a reference manual and a summary of research conducted on, and in, the system. The tutorial spans chapters 2 to 6. Chapter 2 presents a systematic foundational explanation of the mathematical theory underlying the Nuprl system, and it attempts to make the basic notions clear and self–evident. Chapter 3, with its collection of examples of statements in Nuprl, serves as the most concrete introduction to the system, and indeed certain readers may wish to start there. Chapter 4 introduces the reader to the Nuprl proof theory, again with lots of examples, and chapter 5 illustrates the system's function in the context of programming. Chapter 6 presents the metalanguage and the concept of tactics.

The reference section of this book comprises chapters 7 through 9. Chapter 7 details the organization of the system and contains a summary of commands and techniques users will find handy. Chapter 8 contains a brief account of the theoretical underpinnings of the system and gives a summary of the rules and ML constructors and destructors associated with the logic. Chapter 9 provides a detailed explanation of tactics.

The remaining chapters in this book consist of information and research results which will be of interest to the advanced user. Chapter 10 contains several recommendations about theory development which should prove extremely helpful for heavy users of Nuprl. Chapter 11 lays out a library of mathematical results we have obtained using the system, while chapter 12 presents recent developments in the logic which should be implemented by September 1985.

The remainder of this chapter discusses the place of Nuprl in computer science research, specifically its relationship to logic, semantics, programming methodology, automated reasoning and intelligent systems. For those readers interested in research, this will provide both a motivation and a framework for further reading.

1.10 Research Topics

Programming Languages

As a programming language Nuprl is similar to functional programming languages with a rich type structure such as Edinburgh ML. However, the type structure of Nuprl is richer than that of any existing programming language in that it allows dependent functions and products as well as sets, quotients and universes. Moreover, our approach to type checking enables the user to design automatic type–checking algorithms which extend those of other type systems. There are interesting connections between concepts in Nuprl and various new ideas being suggested for programming languages; see for example Russell [Donahue & Demers 79], Pebble [Burstall & Lampson 84] and Standard ML [Milner 85].

Presently Nuprl is only interpreted, so it is much slower than languages with compilers. However, for a simpler subsystem, Lambda–prl, an optimizing compiler has been implemented (see [Sasaki 85]). There are many challenging issues here for people interested in building a new kind of optimizing compiler, as, for instance, the fact that programs usually occur in the context of their specifications means that there is more information available for good optimizations than is typically the case.

Nuprl is also a formal logical theory, so one would expect a connection with logic programming and with PROLOG [Clocksin & Mellish 81]. In fact it is possible to express some aspects of PROLOG in a subset of Nuprl and to use a linear resolution tactic to execute specifications in this subset. Nuprl does not optimize such a proof tactic, however, so the execution would be slower than in existing PROLOG evaluators. Moreover, the usual way to use theorem–proving power computationally in Nuprl involves proving a theorem of the form

$$\textit{for all } x \textit{ of type } A \textit{ there is a } y \textit{ of type } B \textit{ such that } R(x, y)$$

and then extracting a program from this proof. Thus in Nuprl a theorem prover is used more as a compiler than as an evaluator.

Semantics

From a semantic point of view the logic of Nuprl is like set theory in that it is a foundational theory of mathematics [Aczel 78, Barwise 75, Friedman 73, McCarty 84, Myhill 75]. We can try to explain the basic intuitions behind the theory to the point that the rules become self–evident; chapter 2 attempts this. We can also relate this theory to other foundational theories such as classical set theory. On the other hand it is just as meaningful and interesting to attempt to understand classical set theory in terms of type

theory and to relate this theory to other apparently simpler ones such as constructive number theory (see [Beeson 83, Beeson 85]).

For computer scientists it is especially interesting to relate this theory to the theory of domains [Scott 76, Stoy 77] because both are attempts to give a mathematically rigorous account of computation. Thus, while it is not necessary to provide a "denotational semantics" for Nuprl as the basis for reasoning about it, one might do so to compare approaches. It is also reasonable to use Nuprl to give a constructive denotational semantics for programming languages; this topic is discussed briefly in chapter 11.

Methodology and Verification

Nuprl embodies a particular programming methodology and carries it to the level of complete formalization and implementation. However, one can use the system and employ the methods at various levels of rigor. It is possible to program in Nuprl in a freewheeling manner with no attention to correctness; it is also possible to state only some properties that a program will satisfy and to present only the barest sketch of an argument that the program meets these specifications. On the other hand, in the same system one can also tighten control over the program step by step and level by level until there is a completely finished machine–checked proof that the program meets its specifications. In a sense the entire book illustrates this methodology, but in chapter 11 there are examples which relate our methods to those of the more popularly exposited school of E. W. Dijkstra, C. A. R. Hoare, E. C. R. Hehner, D. Gries and others [Dijkstra 76, Hoare 72, Gries 81, Hehner 79].

Environments

The editing facilities of this system were inspired in part by those of the Cornell Program Synthesizer [Teitelbaum & Reps 81] and by AVID [Krafft 81], the system which first applied these ideas in the setting of proof synthesis. We have been using these editors in the Lambda–prl system since 1983. We know how to use them effectively, and while we have not seen demonstrated a better way of generating proofs, based on our experience with these systems we now understand feasible ways to improve them significantly. Research on this topic is under way in our group.

Theorem Proving

Mathematical texts consist not only of definitions and theorems but also of proof techniques. To formalize mathematics one must formalize these methods as well. In Nuprl this is done by expressing proof methods as programs in the ML programming language. We think of ML as formalizing the computational part of the metalanguage. One of the research topics

of our group concerns the complete formalization of the metalanguage (see [Knoblock 86]).

This approach to formalization brings with it a method of mechanizing reasoning in Nuprl, for as users write more and more elaborate proof–building methods the system can do more of the work of finding proofs. This approach to "theorem proving" continues the tradition of our earlier work on PL/CV [Constable, Johnson, & Eichenlaub 82, Constable & O'Donnell 78] and follows in the spirit of the Edinburgh LCF project, which spawned ML. The style of formalization extends the AUTOMATH [deBruijn 80] approach to presenting proofs, an approach which relies mainly on a very high–level language for writing proofs. As more and more complex tactics are built this line of research comes closer to the mainstream work in *automated theorem proving* [Andrews 71, Bibel 80, Bledsoe 77, Boyer & Moore 79, Loveland 78, Weyhrauch 80, Wos et al. 84]. It is conceivable that one would write a tactic which is so ambitious that it would be classified as a "theorem prover".

Although we have written many useful tactics and although the system provides a nontrivial level of deductive support, none of our methods can presently be classified as theorem provers. We are, however, exploring some interesting new methods such as analogy tactics and "expert reasoners" which we think will be contributions to the subject of *automated reasoning*.

Program Synthesis

The Nuprl system acts as a program synthesizer in the following sense. Given a computationally meaningful assertion (e.g., a "program specification") and an applicable proof tactic which produces at least the executable parts of a proof of the assertion, the Nuprl extractor will produce from the proof a program. If the proof is complete then the program meets the specification. In this sense we have written a variety of program–synthesizing tactics. Several interesting comparisons can be drawn between these techniques and those based on classical methods, as in [Manna & Waldinger 80, Manna & Waldinger 85].

Artificial Intelligence

The language of Nuprl has been described as a "logic for artificial intelligence" [Turner 84]. The Nuprl system can be viewed as an intelligent system, and many of the things we do with it and to it might be characterized as improving its "intelligence". As we develop more "expert reasoners" similar to those for equality and arithmetic, for example, we certainly increase the deductive power of the system, especially as we allow them to cooperate [Nelson & Oppen 79]. The natural growth path for a system like Nuprl tends toward increased "intelligence". This is obvious when we think about

adding definitions, theorems and methods to the library. It is less obvious and more noteworthy that other changes move it in this direction as well. For example, it is helpful if the system is aware of what is in the library and what users are doing with it. It is good if the user knows when to invoke certain tactics, but once we see a pattern to this activity, it is easy and natural to inform the system about it. Hence there is an impetus to give the system more knowledge about itself. This is related to recent themes in the field of artificial intelligence [Bundy 83, Charniak & McDermott 85, Davis & Lenat 82].

1.11 Related Work

Close Relatives

The literature most closely related to this comes from Martin-Löf [Martin-Löf 73, Martin-Löf 82] and the Swedish computer scientists at Goteborg who are concerned with making type theory a usable formalism [Nordstrom 81, Nordstrom & Petersson 83, Nordstrom & Smith 84]. The AUTOMATH work at Eindhoven in The Netherlands is also closely related [deBruijn 80, Jutting 79, van Daalen 80, Zucker 75], as is the work at INRIA [Coquand & Huet 85]. In fact, deBruijn's ideas are direct predecessors of those in Martin-Löf's writings with intermediate refinements from Scott [Scott 70]. One can see in deBruijn's work the influence of his fellow mathematician and countryman, L. E. J. Brouwer, the founder of Intuitionism [Brouwer 23]. We will examine some of these influences below as well as trace the influence of more recent research such as the work of J. L. Bates [Bates 79] and Constable [Constable 71, Constable & Zlatin 84, Constable 85].

Intuitionism and Type Theory

In his 1907 doctoral thesis [Brouwer 23] Brouwer advanced the view that the basis of mathematics and of logic is found in the capacity of human beings to carry out mental constructions. On the method of proof by contradiction he says, "The words of your mathematical demonstration merely accompany a mathematical construction that is effected without words. At the point where you enounce the contradiction, I simply perceive that the construction no longer *goes*, that the required structure cannot be imbedded in the given basic structure. And when I make this observation, I do not think of a principium contradictionis."

In the 1960's the mathematician E. Bishop carried out a sweeping development of mathematics along the lines conceived by Brouwer. Bishop's works, and those of his followers [Bishop 67, Bishop & Bridges 85, Bishop & Cheng 72, Bridges 79, Chan 74], have started a modern school of constructive mathematics that we believe is in harmony with the influence of computing in

mathematics, an influence seen in many quarters [Nepeivoda 82, Goad 80, Goto 79, Takasu 78]. In Bishop's words [Bishop 67], "The positive integers and their arithmetic are presupposed by the very nature of our intelligence.... Every mathematical statement ultimately expresses the fact that if we perform certain computations within the set of positive integers, we shall get certain results.... Thus even the most abstract mathematical statement has a computational basis."

Very different views of mathematics were put forward by G. Frege [Frege 1879] and by Russell [Russell 08, Whitehead & Russell 25], as they attempted to reduce "mathematical" ideas to "logical" ideas. We shall not consider the philosophical points here, but we note that Frege gave us the predicate calculus in *Begriffssschrift* and that Russell and Whitehead gave us *Principia Mathematica*, a formal system of mathematics based on type theory.

Nuprl can be seen as expressing the philosophical ideas of Brouwer and Bishop in a language closely related to the type theory of Russell. Types prescribe constructions; some of these constructions are recognized as the meaning of propositions.

Theoretical Background

In 1930 the Intuitionist mathematician A. Heyting [Heyting 30] presented a set of rules that characterized Intuitionistic logic and differentiated it from classical logic. In addition he formulated Intuitionistic first–order number theory (based on Peano's famous axioms) as the classical theory without the law of the excluded middle. (The Intuitionistic theory is often called *Heyting arithmetic*.)

In 1945 the logician S. C. Kleene proposed his well-known *realizability* interpretation of Intuitionistic logic and number theory [Kleene 45]. He showed that any function definable in Heyting arithmetic is Turing computable. This raised the possibility that formal systems of Intuitionistic logic could be used as programming languages. A specific proposal for this was made by Constable [Constable 71] as part of a study of very high–level programming languages and by Bishop [Bishop 70] as part of his study of constructive analysis [Bishop 67]. In 1976 Constable together with his students M. J. O'Donnell, S. D. Johnson, D.B. Krafft and D.R. Zlatin developed a new kind of formal system called *programming logic* [Constable 77, Constable & O'Donnell 78, Constable, Johnson, & Eichenlaub 82] which interpreted programs as constructive proofs and allowed execution of these proofs. This system is called PL/CV.

The propositions–as–types principle [Curry, Feys, & Craig 58, Howard 80] offered a new way to interpret constructive logic and extract its constructive content. These ideas came to us from [Stenlund 72, Martin-Löf 73] and were applied in the PL/CV setting in version V3 [Constable & Zlatin 84,

Stansifer 85]. They were further refined by Bates [Bates 79] and became the basis for the actual Nuprl system [Bates & Constable 85, Bates & Constable 83].

Much of our recent theoretical work has focused on the Nuprl system. In his Ph.D. thesis R. W. Harper [Harper 85] investigated the relative consistency of Nuprl with Martin-Löf's theories, and he established the theoretical basis for automating equality reasoning in Nuprl. S. F. Allen has studied a rigorous semantic account of type theory; some of his ideas appear in chapter 8. Mendler and Constable have studied recursive types and partial functions[Constable & Mendler 85]; some of these results appear in chapter 12.

Systems Background

In 1979 Bates [Bates 79] proposed the notion of *refinement logic* and an interactive style of doing proofs in such a logic (see also [Kay & Goldberg 77, Smullyan 68, Toledo 75]). At the same time our experience with the Cornell Program Synthesizer of R. Teitelbaum and T. Reps [Teitelbaum & Reps 81] directly influenced our thinking about proof editors. Some of these ideas were first tested by Krafft for the PL/CV logic in a system called AVID [Krafft 81]. Bates also proposed specific ways to extract executable code from constructive proofs that would be efficient enough for use in a programming system. In 1982 a specific system of this kind was designed and implemented by the Prl group at Cornell [Bates & Constable 85, Nuprl Staff 83]; that system is Lambda–prl.

The Lambda–prl system was designed to interact with the metalanguage ML from the Edinburgh LCF project [Gordon, Milner, & Wadsworth 79].[4] The Lambda–prl system also incorporated decision procedures from the PL/CV project; indeed the early Lambda–prl style of automated reasoning is a synthesis of the ideas from AUTOMATH, PL/CV and LCF. Now a distinctive new style is emerging, a style which will be reported in the work of some of the authors of this monograph.

The Nuprl logic itself was designed as an attempt to present the Cornell version of constructive type theory in the setting a of refinement–style logic. The result of this design is embodied in the system and in this document.

[4] This is one result of a joint Cornell/Edinburgh study of programming logics and automated reasoning begun in 1981-82 and continuing with joint studies of type theory and polymorphism.

Chapter 2

Introduction to Type Theory

Sections 2.1 to 2.4 introduce a sequence of approximations to Nuprl, starting with a familiar formalism, the *typed lambda calculus*. These approximations are not exactly subsets of Nuprl, but the differences between these theories and subtheories of Nuprl are minor. These subsections take small steps toward the full theory and relate each step to familiar ideas. Section 2.5 summarizes the main ideas and can be used as a starting point for readers who want a brief introduction. The last two sections relate the idea of a type in Nuprl to the concept of a set and to the concept of a data type in programming languages.

2.1 The Typed Lambda Calculus

A *type* is a collection of objects having similar structure. For instance, integers, pairs of integers and functions over integers are at least three distinct types. In both mathematics and programming the collection of functions is further subdivided based on the kind of input for which the function makes sense, and these divisons are also called types, following the vocabulary of the very first type theories [Whitehead & Russell 25]. For example, we say that the integer successor function is a function from integers to integers, that inversion is a function from invertible functions to functions (as in "subtraction is the inverse of addition"), and that the operation of functional composition is a function from two functions to their composition. One modern notation for the type of functions from type A into type B is $A \rightarrow B$ (read as A "arrow" B). Thus integer successor has type $int \rightarrow int$, and the curried form of the composition of integer functions has type $(int \rightarrow int) \rightarrow ((int \rightarrow int) \rightarrow (int \rightarrow int))$. For our first look at types we

will consider only those built from type variables $A, B, C, \ldots$ using arrow. Hence we will have $(A \rightarrow B), (A \rightarrow B) \rightarrow C, ((A \rightarrow B) \rightarrow (A \rightarrow B))$, etc., as types, but not $int \rightarrow int$. This will allow us to examine the general properties of functions without being concerned with the details of concrete types such as integers.

One of the necessary steps in defining a type is choosing a notation for its elements. In the case of the integers, for instance, we use notations such as $0, +1, -2, +3, \ldots$. These are the defining notations, or *canonical forms*, of the type. Other notations, such as $1 + 1$, $2 * 3$ and $2 - 1$, are not defining notations but are derived notations or *noncanonical* forms.

Informally, functions are named in a variety of ways. We sometimes use special symbols like $+$ or $*$. Sometimes in informal mathematics one might abuse the notation and say that $x + 1$ is the successor function. Formally, however, we regard $x + 1$ as an ambiguous value of the successor function and adopt a notation for the function which distinguishes it from its values. Betrand Russell [Whitehead & Russell 25] wrote $\hat{x} + 1$ for the successor function, while Church [Church 51] wrote $\lambda x.x + 1$. Sometimes one sees the notation $() + 1$, where $()$ is used as a "hole" for the argument. In Nuprl we adopt the lambda notation, using $\backslash x.x + 1$ as a printable approximation to $\lambda x.x + 1$. This notation is not suitable for all of our needs, but it is an adequate and familiar place to start.

A Formal System

We will now define a small formal system for deriving *typing relations* such as $(\backslash x.x)$ *in* $(A \rightarrow A)$. To this end we have in mind the following two classes of expression. A *type expression* has the form of a type variable $A, B, C, \ldots$ (an example of an *atomic* type) or the form $(T_1 \rightarrow T_2)$, where T_1 and T_2 are type expressions. If we omit parentheses then arrow associates to the right; thus $A \rightarrow B \rightarrow C$ is $A \rightarrow (B \rightarrow C)$. An *object expression* has the form of a variable, $x, y, z, \ldots$, an abstraction, $\backslash x.b$ or of an application, $a(b)$, where a and b are object expressions. We say that b is the *body* of $\backslash x.b$ and the *scope* of $\backslash x$, a binding operator.

In general, a variable y is *bound* in a term t if t has a subterm of the form $\backslash y.b$. Any occurrence of y in b is bound. A variable occurrence in t which is not bound is called *free*. We say that a term a is *free for a variable x in a term t* as long as no free variable of a becomes bound when a is substituted for each free occurrence of x. For example, z is free for x in $\backslash y.x$, but y is not. If a has a free variable which becomes bound as a result of a substitution then we say that the variable has been *captured*. Thus $\backslash y$ "captures" y if we try to substitute y for x in $\backslash y.x$. If t is a term then $t[a/x]$ denotes the term which results from replacing each free occurrence of x in t by a, provided that a is free for x in t. If a is not free for x then $t[a/x]$ denotes t with a replacing each free x and the bound variables of t which would capture free variables

of a being renamed to prevent capture. $t[a_1, \ldots, a_n/x_1, \ldots, x_n]$ denotes the *simultaneous substitution* of a_i for x_i. We agree that two terms which differ only in their bound variable names will be treated as equal everywhere in the theory, so $t[a/x]$ will denote the same term inside the theory regardless of capture. Thus, for example, $(\backslash y.x(y))[t/x] = \backslash y.t(y)$ and $(\backslash x.x(y))[t/x] = \backslash x.x(y)$ and $(\backslash y.x(y))[y/x] = \backslash z.y(z)$.

When we write $(\backslash x.x)$ *in* $(A \rightarrow A)$ we mean that $\backslash x.x$ names a function whose type is $A \rightarrow A$. To be more explicit about the role of A, namely that it is a type variable, we *declare* A in a context or environment. The environment has the single declaration $A{:}U_1$, which is read "A is a type."[1]

For T a type expression and t an object expression, t *in* T will be called a *typing*. To separate the context from the typing we use >>. To continue the example above, the full expression of our goal is $A{:}U_1$ >> $(\backslash x.x)$ *in* $(A \rightarrow A)$.

In general we use the following terminology. *Declarations* are either *type declarations*, in which case they have the form $A{:}U_1$, or *object declarations*, in which case they have the form $x{:}T$ for T a type expression.[2] A *hypothesis list* has the form of a sequence of declarations; thus, for instance, $A{:}U_1, B{:}U_1, x{:}A$ is a hypothesis list. In a *proper* hypothesis list the types referenced in object declarations are declared first (i.e., to the left of the object declaration). A *typing* has the form t *in* T, where t is an object expression and T is a type expression. A *goal* has the form H >> t *in* T, where H is a hypothesis list and t *in* T is a typing.

We will now give rules for proving goals. The rules specify a finite number of subgoals needed to achieve the goal. The rules are stated in a "top–down" form or, following Bates [Bates 79], *refinement* rules. The general shape of a refinement rule is:

goal by *rule name*

1. subgoal 1

.

.

.

n. subgoal n.

Here is a sample rule.

H >> $(\backslash x.b)$ *in* $(S \rightarrow T)$ by intro
1.$H, x{:}S$ >> b *in* T
2.H >> S *in* $U1$

[1] Actually $A{:}U_1$ reads as "A is a type in universe U_1." We discuss *universes* later.
[2] In full Nuprl the distinction between types and other objects is dropped.

(1) $H, x{:}A, H' >> x\ in\ A$ by hyp x

(2) $H >> (S \rightarrow T)\ in\ U1$ by intro
  1. $H >> S\ in\ U1$
  2. $H >> T\ in\ U1$

(3) $H >> (\backslash x.b)\ in\ (S \rightarrow T)$ by intro [new y]
  1. $H, y{:}S >> b[y/x]\ in\ T$
  2. $H >> S\ in\ U1$

(4) $H >> f(a)\ in\ T$ by intro using $(S \rightarrow T)$
  1. $H >> f\ in\ (S \rightarrow T)$
  2. $H >> a\ in\ S$

Figure 2.1: Rules for the Typed Lambda Calculus

It reads as follows: to prove that $(\backslash x.b)$ is a function in $(S \rightarrow T)$ in the context H (or from hypothesis list H) we must achieve the subgoals $H, x{:}S >> b\ in\ T$ and $H >> S\ in\ U_1$. That is, we must show that the body of the function has the type T under the assumption that the free variable in the body has type S (a proof of this will demonstrate that T is a type expression), and that S is a type expression.

A *proof* is a finite tree whose nodes are pairs consisting of a subgoal and a rule name or a placeholder for a rule name. The subgoal part of a child is determined by the rule name of the parent. The leaves of a tree have rule parts that generate no subgoals or have placeholders instead of rule names. A tree in which there are no placeholders is *complete*. We will use the term *proof* to refer to both complete and incomplete proofs.

Figure 2.1 gives the rules for the small theory. Note that in rule (3) the square brackets indicate an optional part of the rule name; if the new y part is missing then the variable `x` is used, so that subgoal 1 is

1. $H, x{:}A >> b\ in\ T$.

The "new variable" part of a rule name allows the user to rename variables so as to prevent capture.

We say that an *initial* goal has the form

$$A_1{:}U_1, A_2{:}U_1, ..., A_n{:}U_1 >> t\ in\ T,$$

where the A_i are exactly the free type variables of T. The Nuprl system allows only initial goals with empty hypothesis lists. Also, in full Nuprl we do not distinguish type variables from any other kind of variable. We have introduced these special notions here as pedagogical devices.[3]

[3] In Nuprl the goal $A{:}U1 >> (\backslash x.x)\ in\ (A \rightarrow A)$ would be expressed initially as

$A{:}U1$ >> $(\backslash y.\backslash x.y(x))(\backslash v.v)$ *in* $(A \rightarrow A)$
by intro using $(A \rightarrow A) \rightarrow (A \rightarrow A)$
1. $A{:}U1$ >> $(\backslash y.\backslash x.y(x))$ *in* $(A \rightarrow A) \rightarrow (A \rightarrow A)$
by intro
1. $A{:}U1, y{:}(A \rightarrow A)$ >> $(\backslash x.y(x))$ *in* $(A \rightarrow A)$
by intro
1. $A{:}U1, y{:}(A \rightarrow A), x{:}A$ >> $y(x)$ *in* A
by intro using $(A \rightarrow A)$
1. $A{:}U1, y{:}(A \rightarrow A), x{:}A$ >> y *in* $(A \rightarrow A)$
by hyp y
2. $A{:}U1, y{:}(A \rightarrow A), x{:}A$ >> x *in* A
by hyp x
2. $A{:}U1, y{:}(A \rightarrow A)$ >> A *in* $U1$ by hyp A
2. $A{:}U1$ >> $(A \rightarrow A)$ *in* $U1$
by intro
1. $A{:}U1$ >> A *in* $U1$ by hyp A
2. $A{:}U1$ >> A *in* $U1$ by hyp A
2. $A{:}U1$ >> $\backslash v.v$ *in* $(A \rightarrow A)$ by intro
1. $A{:}U1, v{:}A$ >> v *in* A by hyp v
2. $A{:}U1$ >> A *in* $U1$ by hyp A

Figure 2.2: A Sample Proof in the Small Type Theory

Figure 2.2 describes a complete proof of a simple fact. This proof provides simultaneously a derivation of $(A \rightarrow A)$ *in* U_1, showing that $(A \rightarrow A)$ is a type expression; a derivation of $(\backslash x.\backslash y.y(x))(\backslash v.v)$, showing that this is an object expression; and type information about all of the subterms, e.g.,

$\backslash v.v$ is in $(A \rightarrow A)$;
$\backslash x.y(x)$ is in $(A \rightarrow A)$ given $y{:}A$;
$\backslash y.\backslash x.y(x)$ is in $(A \rightarrow A) \rightarrow (A \rightarrow A)$;
$(\backslash y.\backslash x.y(x))(\backslash v.v)$ is in $(A \rightarrow A)$.

There is a certain conceptual economy in providing all of this information in one format, but the price is that some of the information is repeated unnecessarily. For example, we show that A *in* U_1 three separate times. This is an inherent difficulty with the style of "simultaneous proof" adopted in Nuprl. In chapter 10 we discuss ways of minimizing its negative effects; for example, one could prove $A{:}U_1$ >> $(A \rightarrow A)$ once as a lemma and cite it as necessary, thereby sparing some repetition.

>> $A{:}U1 \rightarrow ((\backslash x.x)$ *in* $(A \rightarrow A))$. A rule of introduction would then create the context $A{:}U1$.

It is noteworthy that from a complete proof from an initial goal of the form $H >> t\ in\ T$ we know that t is a closed object expression (one with no free variables) and T is a type expression whose free variables are declared in H. Also, in all hypotheses lists in all subgoals any expression appearing on the right side of a declaration is either U_1 or a type expression whose variables are declared to the left. Moreover, all free variables of the conclusion in any subgoal are declared exactly once in the corresponding hypothesis list. In fact, no variable is declared at a subgoal unless it is free in the conclusion. Furthermore, every subterm t' receives a type in a subproof $H' >> t'\ in\ T'$, and in an application, $f(a)$, f will receive a type $(T_1 \rightarrow T_2)$ and a will receive the type T_1. Properties of this variety can be proved by induction on the construction of a complete proof. For full Nuprl many properties like these are proved in the Ph.D. theses of R. W. Harper [Harper 85] and S. F. Allen [Allen 86].

Computation System

The meaning of lambda terms $\backslash x.b$ is given by computation rules. The basic rule, called *beta reduction*, is that $(\backslash x.b)(a)$ reduces to $b[a/x]$; for example, $(\backslash x.\backslash y.x)(\backslash v.v)$ reduces to $\backslash y.(\backslash v.v)$. The strategy for computing applications $f(a)$ is involves reducing f until it has the form $\backslash x.b$, then computing $(\backslash x.b)(a)$. This method of computing with noncanonical forms $f(a)$ is called *head reduction* or *lazy evaluation*, and it is not the only possible way to compute. For example, we might reduce f to $\backslash x.b$ and then continue to perform reductions in the body b. Such steps might constitute computational optimizations of functions. Another possibility is to reduce a first until it reaches canonical form before performing the beta reductions. This corresponds to call–by–value computation in a programming language.

In Nuprl we use lazy evaluation, although for the simple calculus of typed lambda terms it is immaterial how we reduce. Any reduction sequence will terminate—this is the strong normalization result [Tait 67, Stenlund 72]—and any sequence results in the same value according to the Church–Rosser theorem [Church 51, Stenlund 72]. Of course, the number of steps taken to reach this form may vary considerably depending on the order of reduction.

2.2 Extending the Typed Lambda Calculus

Dependent Function Space

It is very useful to be able to describe functions whose range type depends on the input. For example, we can imagine a function on integers of the form $\backslash x.if\ even(x)\ then\ 2\ else\ (\backslash x.2)$. The type of this function on input x is $if\ even(x)\ then\ int\ else\ (int \rightarrow int)$. Call this type expression $F(x)$;

(0) $H >> U1\ in\ U2$ by intro

(1) $H, x{:}A, H' >> x\ in\ A$ by hyp x

(2′) $H >> (x{:}A \rightarrow T)\ in\ Ui$ by intro for $i = 1, 2$
1. $H >> A\ in\ Ui$
2. $H, x{:}A >> T\ in\ Ui$

(3′) $H >> (\backslash x.b)\ in\ (y{:}S \rightarrow T)$ by intro at Ui for $i = 1, 2$
1. $H, x{:}S >> b\ in\ T[x/y]$
2. $H >> S\ in\ Ui$

(4′) $H >> f(a)\ in\ T[a/x]$ by intro using $(x{:}S \rightarrow T)$
1. $H >> f\ in\ (x{:}S \rightarrow T)$
2. $H >> a\ in\ S$

(5) $H >> A\ in\ U2$ by cumulativity
$H >> A\ in\ U1$

Figure 2.3: Rules for Dependent Functions

then the function type we want is written $x{:}int \rightarrow F(x)$ and denotes those functions f whose value on input n belongs to $F(n)$ $(f(n)\ in\ F(n))$.

In the general case of pure functions we can introduce such types by allowing declarations of parameterized types or, equivalently, type–valued functions. These are declared as $B{:}(A \rightarrow U_1)$. To introduce these properly we must think of U_1 itself as a type, but a large type. We do not want to say $U_1\ in\ U_1$ to express that U_1 is a type because this leads to paradox in the full theory. It is in the spirit of type theory to introduce another layer of object, or in our terminology, another "universe", called U_2. In addition to the types in U_1, U_2 contains so–called *large types*, namely U_1 and types built from it such as $A \rightarrow U_1$, $U_1 \rightarrow U_1$, $A \rightarrow (B \rightarrow U_1)$ and so forth. To say that U_1 is a large type we write $U_1\ in\ U_2$. The new formal system allows the same class of object expressions but a wider class of types. Now a variable $A, B, C, \ldots$ is a type expression, the constant U_1 is a type expression, if T is a type expression (possibly containing a free occurrence of the variable x of type S) then $x{:}S \rightarrow T$ is a type expression, and if F is an object expression of type $S \rightarrow U_1$ then $F(x)$ is a type expression. The old form of function space results when T does not depend on x; in this case we still write $S \rightarrow T$.

The new rules are listed in figure 2.3. With these rules we can prove the following goals.

>> $(\backslash A.\backslash x.x)\ in\ (A{:}U1 \rightarrow (A \rightarrow A))$

>> $\backslash A.\backslash B.\backslash g.\backslash b.\backslash x.b(g(x))\ in$
$(A{:}U1 \rightarrow (B{:}(A \rightarrow U1) \rightarrow$
$(g{:}(A \rightarrow A) \rightarrow (b{:}(x{:}A \rightarrow B(x)) \rightarrow (x{:}A \rightarrow B(g(x)))))))$

With the new degree of expressiveness permitted by the dependent arrow we are able to dispense with the hypothesis list in the initial goal in the above examples. We now say that an initial goal has the form >> $t\ in\ T$, where t is an object expression and T is a type expression. One might expect that it would be more convenient to allow a hypothesis list such as $A{:}U_1, B{:}(A \rightarrow U_1)$, but such a list would have to be checked to guarantee well–formedness of the types. Such checks become elaborate with types of the form $c{:}(x{:}A \rightarrow y{:}B(x) \rightarrow U_1)$, and the hypothesis–checking methods would become as complex as the proof system itself. As the theory is enlarged it will become impossible to provide an algorithm which will guarantee the well–formedness of hypotheses. Using the proof system to show well–formedness will guarantee that the hypothesis list is well–formed.

Hidden in the explanation above is a subtle point which affects the basic design of Nuprl. The definition of a type expression involves the clause "F is an expression of type $S \rightarrow U_1$." Thus, in order to know that $t\ in\ T$ is an allowable initial goal, we may have to determine that a subterm of T is of a certain type; in the example above, we must show that B is of type $A \rightarrow U_1$. To define this concept precisely we would need some precise definition of the relation that B is of type $S \rightarrow U_1$. This could be given by a type–checking algorithm or by an inductive definition, but in either case the definition would be as complex as the proof system that it is used to define.

Another approach to this situation is to take a simpler definition of an initial goal and let the proof system take care of ensuring that only type expressions can appear on the right–hand side of a typing. To this end, we define the syntactically simple concept of a *readable expression* and then state that an initial goal has the form $e_1\ in\ e_2$, where e_1 and e_2 are these simple expressions. Using this approach, an expression is either:

a variable: $A, B, C, ..., x, y, z$;

a constant: U_1;

an application: $f(a)$;

an abstraction: $\backslash x.b$; or

an arrow: $x{:}a \rightarrow b$,

where a, b and f are expressions. This allows expressions such as $\backslash x.U_1$ or $y{:}A \rightarrow \backslash x.x$, which do not make sense in this theory. However, the proof

rules are organized so that if the initial goal $>> t\ in\ T$ is proved then T will be a type expression and t will be an object expression of type T.

Cartesian Product

One of the most basic ways of building new objects in mathematics and programming involves the ordered pairing constructor. For example, in mathematics one builds rational numbers as pairs of integers and complex numbers as pairs of reals. In programming, a data processing record might consist of a name paired with basic information such as age, social security number, account number and value of the account, e.g., $< bloog, 37, 396-54-3900, 12268, .01 >$. This item might be thought of as a single 5–tuple or as compound pair $< bloog, < 37, < 396-54-3900, < 12268, .01 >>>>$. In Nuprl we write $< a, b >$ for the pair consisting of a and b; n–tuples are built from pairs.

The rules for pairs are simpler than those for functions because the canonical notations are built in a simple way from components. We say that $< a, b >$ is a canonical value for elements of the type of pairs; the name $< a, b >$ is canonical even if a and b are not canonical. If a is in type A and b is in type B then the type of pairs is written $A\#B$ and is called the *cartesian product*. The Nuprl notation is very similar to the set–theoretic notation, where a cartesian product is written $A \times B$; we choose $\#$ as the operator because it is a standard ASCII character while $\times$ is not. In programming languages one might denote the cartesian product as $RECORD(A, B)$, as in the Pascal record type, or as $struct(A, B)$, as in Algol 68 structures.

The pair decomposition rule is the only Nuprl rule for products that is not as one might expect from cartesian products in set theory or from record types in programming. One might expect operations, say $1of()$ and $2of()$, obeying

$1of(< a, b >) = a$ and
$2of(< a, b >) = b$.

Instead of this notation we use a single form that generalizes both forms. One reason for this is that it allows a single form which is the inverse of pairing. Another more technical reason will appear when we discuss dependent products below. The form is

$$spread(p; u, v.b),$$

where p is an expression denoting a pair and where b is any expression in u and v. We think of u and v as names of the elements of the pair; these names are bound in b. Using *spread* we can define the selectors $1of()$ and $2of()$ as

$1of(p) = spread(p; u, v.u)$ and
$2of(p) = spread(p; u, v.v)$.

(1) $H >> A\#B\ in\ U1$ by intro
$H >> A\ in\ U1$
$H >> B\ in\ U1$

(2) $H >> <a, b>\ in\ A\#B$ by intro
$H >> a\ in\ A$
$H >> b\ in\ B$

(3) $H >> spread(p; u, v.b)\ in\ T$ by intro using $A\#B$
$H >> p\ in\ A\#B$
$H, u{:}A, v{:}B >> b\ in\ T$

Figure 2.4: Rules for Cartesian Product

Figure 2.4 lists the rules for cartesian product. These rules allow us to assign types to pairs and to the spread terms. We will see later that Nuprl allows variations on these rules.

Dependent Products

Just as the function space constructor is generalized from $A \rightarrow B$ to $x{:}A \rightarrow B$, so too can the product constructor be generalized to $x{:}A\#B$, where B can depend on x. For example, given the declarations $A{:}U_1$ and $F{:}A \rightarrow U_1$, $x{:}A\#F(x)$ is a type in U_1. The formation rule for dependent types becomes the following.

$(1')H >> (x{:}A\#B)\ in\ U1$ by intro
$H >> A\ in\ U1$
$H, x{:}A >> B\ in\ U1$

The introduction rules change as follows.

$(2')H >> <a, b>\ in\ (x{:}A\#B)$ by intro
$H >> a\ in\ A$
$H >> b\ in\ B[a/x]$

$(3')H >> spread(p; u, v.b)\ in\ T[p/z]$
by intro over $z.T$ using $x{:}A\#B$
$H >> p\ in\ x{:}A\#B$
$H, u{:}A, v{:}B[u/x] >> b\ in\ T[< u, v > /z]$

The term "over $z.T$" is needed in order to specify the substitution of $< u, v >$ in T.

Disjoint Union

A union operator represents another basic way of combining concepts. For example, if T represents the type of triangles, R the type of rectangles and C the type of circles, then we can say that an object is a triangle or a rectangle or a circle by saying that it belongs to the type T or R or C. In Nuprl this type is written $T|R|C$.

In general if A and B are types, then so is their disjoint union, $A|B$. Semantically, not only is the union disjoint, but given an element of $A|B$, it must be possible to decide which component it is in. Accordingly, Nuprl uses the canonical forms $inl(a)$ and $inr(b)$ to denote elements of the union; for $a\ in\ A$ $inl(a)$ is in $A|B$, and for $b\ in\ B$ $inr(b)$ is in $A|B$.

To discriminate on disjuncts, Nuprl uses the form $decide(d; u.e; v.f)$. The interpretation is that if d denotes terms of the form $inl(a)$ then

$$decide(inl(a); u.e.; v.f) = e[a/u],$$

and if it denotes terms of the form $inr(b)$ then

$$decide(inr(b); u.e; v.f) = f[b/v].$$

The variable u is bound in e and v is bound in f. It is noteworthy that the type $A|B$ can be defined in terms of # and a two–element type such as $\{0, 1\}$.

Integers

The type of integers, int, is built into Nuprl. The canonical members of this type are $0, +1, -1, +2, -2, \ldots$. The operations of addition, $+$, subtraction, $-$, multiplication, $*$, and division, $/$, are built into the theory along with the modulus operation, $a\ mod\ b$, which gives the positive remainder of dividing a by b. Thus $-5\ mod\ 2 = 1$. Division of two integers produces the integer part of real number division, so $5/2 = 2$. For nonnegative integers a and b we have $a = b * (a/b) + a\ mod\ b$.

There are three noncanonical forms associated with the integers. The first form captures the fact that integer equality is decidable; $int\_eq(a; b; s; t)$ denotes s if $a = b\ in\ int$ and denotes t otherwise. The second form captures the computational meaning of less than; $less(a; b; s; t)$ denotes s if $a < b$ and t otherwise. The third form provides a mechanism for definition and proof by induction and is written $ind(a; x, y.s; b; u, v.t)$. It is easiest to see this form as a combination of two simple induction forms over the nonnegative and nonpositive integers. Over the nonnegative integers $(0, +1, +2, +3, \ldots)$ the form denotes an inductive definition satisfying the following equations:

$$ind(0; x, y.s; b; u, v.t) = b$$
$$ind(n+1; x, y.s; b; u, v.t) = t[(n+1), ind(n; x, y.s; b; u, v.t)/u, v].$$

Over the nonpositive integers $(0, -1, -2, \ldots)$ the form denotes an inductive definition satisfying these equations:

$$ind(0; x, y.s; b; u, v.t) = b$$
$$ind(n-1; x, y.s; b; u, v.t) = s[(n-1), ind(n; x, y.s; b; u, v.t)/x, y].$$

For example, this form could be used to define $n!$ as $ind(n; x, y.1; 1; u, v.u * v)$ if we assume that for $n < 0, n! = 1$.

In the form $ind(a; x, y.s; b; u, v.t)$ a represents the integer argument, b represents the value of the form if $a = 0$, $x, y.s$ represents the inductive case for negative integers, and $u, v.t$ represents the inductive case for positive integers. The variables x and y are bound in s, while u and v are bound in t.

Atoms and Lists

The type of *atoms* is provided in order to model character strings. The canonical elements of the type *atom* are "...", where ... is any character string. Equality on atoms is decidable using the noncanonical form $atom\_eq(a; b; s; t)$, which denotes s when $a = b$ *in atom* and t otherwise.

Nuprl also provides the type of lists over any other type A; it is denoted $A\ list$. The canonical elements of the type $A\ list$ are nil , which corresponds to the empty list, and $a.b$, where a is in A and b is in $A\ list$. For example, the list of the first three positive integers in descending order is denoted $3.(2.(1.nil))$.

It is customary in the theory of lists to have *head* and *tail* functions such that

$head(a.b) = a$ and
$tail(a.b) = b$.

These and all other functions on lists that are built inductively are defined in terms of the list induction form $list\_ind(a; b; h, t, v.t)$. The meaning of this form is given by the following equations.

$$list\_ind(nil; b; h, t, v.t) = b$$
$$list\_ind(a.r; b; h, t, v.t) = t[a, r, list\_ind(r; b; h, t, v.t)/h, t, v]$$

With this form the tail function can be defined as $list\_ind(a; nil; h, t, v.t)$. The basic definitions and facts from list theory appear in chapter 11.

2.3 Equality and Propositions as Types

So far we have talked exclusively about types and their members. We now want to talk about simple declarative statements. In the case of the integers many interesting facts can be expressed as equations between terms. For example, we can say that a number n is even by writing $n \; mod \; 2 = 0 \; in \; int$. In Nuprl the equality relation on int is built–in; we write $x = y \; in \; int$. In fact, each type A comes with an equality relation written $x = y \; in \; A$. The idea that types come equipped with an equality relation is very explicit in the writings of Bishop [Bishop 67]. For example, in *The Foundation of Constructive Analysis*, he says, "A set is defined by describing what must be done to construct an element of the set, and what must be done to show that two elements of the set are equal." The notion that types come with an equality is central to Martin-Löf's type theories as well.

The equality relations $x = y \; in \; A$ play a dual role in Nuprl in that they can be used to express type membership relations as well as equality relations within a type. Since each type comes with such a relation, and since $a = b \; in \; A$ is a sensible relation only if a and b are members of A, it is possible to express the idea that a belongs to A by saying that $a = a \; in \; A$ is true. In fact, in Nuprl the form $a \; in \; A$ is really shorthand for $a = a \; in \; A$.

The equality statement $a = a \; in \; A$ has the curious property that it is either true or nonsense. If a has type A then $a = a \; in \; A$ is true; otherwise, $a = a \; in \; A$ is not a sensible statement because $a = b \; in \; A$ is sensible only if a and b belong to A. Another way to organize type theory is to use a separate form of judgement to say that a is in a type, that is, to regard $a \; in \; A$ as distinct from $a = a \; in \; A$. That is the approach taken by Martin-Löf. It is also possible to organize type theory without built–in equalities at all except for the most primitive kind. We only need equality on some two–element type, say a type of booleans, $\{\text{true}, \text{false}\}$; we could then define equality on int as a function from int into $\{\text{true}, \text{false}\}$ The fact that each type comes equipped with equality complicates an understanding of the rules, as we see when we look at functions. If we define a function $f \; in \; (A \rightarrow B)$ then we expect that if $a_1 = a_2 \; in \; A$ then $f(a_1) = f(a_2) \; in \; B$. This is a key property of functions, that they respect equality. In order to guarantee this property there are a host of rules of the form that if part of an expression is replaced by an equal part then the results are equal. For example, the following are rules.

$$H >> spread(a; x, y.t) = spread(b, x, y.t) \; in \; T$$
$$H >> a = b \; in \; A \# B$$

$$H >> decide(a; x.s; y.t) = decide(b; x.s; y.t) \; in \; T$$
$$H >> a = b \; in \; A.$$

Propositions as Types

An equality form such as $a = b\ in\ A$ makes sense only if A is a type and a and b are elements of that type. How should we express the idea that $a = b\ in\ A$ is well–formed? One possibility is to use the same format as in the case of types. We could imagine a rule of the following form.

$H >> (a = b\ in\ A)\ in\ U1$ by intro
  $H >> A\ in\ U1$
  $H >> a\ in\ A$
  $H >> b\ in\ A$

This rule expresses the right ideas, and it allows well–formedness to be treated through the proof mechanism in the same way that well–formedness is treated for types. In fact, it is clear that such an approach will be necessary for equality forms if it is necessary for types because it is essential to know that the A in $a = b\ in\ A$ is well–formed.

Thus an adequate deductive apparatus is at hand for treating the well–formedness of equalities, provided that we treat $a = b\ in\ A$ as a type. Does this make sense on other grounds as well? Can we imagine an equality as denoting a type? Or should we introduce a new category, called *Prop* for proposition, and prove $H >> (a = b\ in\ A)$ in Prop? The constructive interpretation of truth of any proposition P is that P is provable. Thus it is perfectly sensible to regard a proposition P as the type of its proofs. For the case of an equality we make the simplifying assumption that we are not interested in the details of such proofs because those details do not convey any more computational information than is already contained in the equality form itself. It may be true that there are many ways to prove $a = b\ in\ A$, and some of these may involve complex inductive arguments. However, these arguments carry only "equality information," not computational information, so for simplicity we agree that equalities considered as types are either empty if they are not true or contain a single element, called *axiom*, if they are true.[4]

Once we agree to treat equalities as types (and over *int*, to treat $a < b$ as a type also) then a remarkable economy in the logic is possible. For instance, we notice that the cartesian product of equalities, say $(a = b\ in\ A)\#(c = d\ in\ B)$, acts precisely as the conjunction $(a = b\ in\ A)\ \&\ (c = d\ in\ B)$. Likewise the disjoint union, $(a = b\ in\ A)|(c = d\ in\ B)$, acts exactly like the constructive disjunction. Even more noteworthy is the fact that the dependent product, say $x{:}int\#(x = 0\ in\ int)$, acts exactly like the constructive existential quantifier, $\exists x{:}int.x = 0\ in\ int$. Less obvious, but also

[4] A better term to use here might be the token "yes", which can be thought of as a summary of the proof. The term *axiom* suggests that the facts are somehow basic or atomic, but in fact they may require considerable work to prove.

valid, is the interpretation of $x{:}A \rightarrow (x = x\ in\ A)$ as the universal statement, $\forall x{:}A.x = x\ in\ A$.

We can think of the types built up from equalities (and inequalities in the case of integer) using $\#$, $|$ and $\rightarrow$ as propositions, for the meaning of the type constructors corresponds exactly to that of the logical operators considered constructively. As another example of this, if A and B are propositions then $A \rightarrow B$ corresponds exactly to the constructive interpretation of $A\ implies\ B$. That is, proposition A implies proposition B constructively if and only if there is a method of building a proof of B from a proof of A, which is the case if and only if there is a function f mapping proofs of A to proofs of B. However, given that A and B are types such an f exists exactly when the type $A \rightarrow B$ is inhabited, i.e., when there is an element of type $A \rightarrow B$.

It is therefore sensible to treat propositions as types. Further discussion of this principle appears in chapters 3 and 11.

Is it sensible to consider any type, say *int* or *int list*, as a proposition? Does it make sense to assert *int*? We can present the logic and the type theory in a uniform way if we agree to take the basic form of assertion as "type A is inhabited." Therefore, when we write the goal H >> A we are asserting that given that the types in H are inhabited, we can build an element of A. When we want to mention the inhabiting object directly we say that it is *extracted* from the proof, and we write H >> $A\ ext\ a$. This means that A is inhabited by the object a. We write the form H >> A instead of H >> $a\ in\ A$ when we want to suppress the details of how A is inhabited, perhaps leaving them to be determined by a computer system as in the case of Nuprl.

When we write $A{:}U_1$ >> $(\backslash x.x)\ in\ (A \rightarrow A)$ we are really asserting the equality

$$A{:}U_1 >> ((\backslash x.x) = (\backslash x.x)\ in\ (A \rightarrow A)).$$

This equality is a type. If it is true it is inhabited by *axiom*. The full statement is therefore

$$A{:}U_1 >> ((\backslash x.x) = (\backslash x.x)\ in\ A \rightarrow A)\ ext\ axiom.$$

As another example of this interpretation, consider the goal

$$>> int.$$

This can be proved by introducing 0, and from such a proof we would extract 0 as the inhabiting witness. Compare this to the goal

$$>> 0\ in\ int.$$

This is proved by introduction, and the inhabiting witness is *axiom*.

2.4 Sets and Quotients

We conclude the introduction of the type theory with some remarks about two, more complex type constructors, the subtype constructor and the quotient type constructor. Informal reasoning about functions and types involves the concept of subtypes. A general way to specify subtypes uses a concept similar to the *set comprehension* idea in set theory; that is, $\{x{:}A|B\}$ is the type of all x of type A satisfying the predicate B. For instance, the nonnegative integers can be defined from the integers as $\{z{:}int|0 <= z\}$. In Nuprl this is one of two ways to specify a subtype. Another way is to use the type $z{:}int\#0 <= z$. Consider now two functions on the nonnegative integers constructed in the following two ways.

$$f{:}\{z{:}int|0 <= z\} \rightarrow int$$
$$g{:}(z{:}int\#0 <= z) \rightarrow int$$

The function g takes a pair $< x, p >$ as an argument, where x is an integer and p is a proof that the integer is nonnegative. The set construct is defined in such a way that f takes only integers as arguments to the computation; the information that the argument is nonnegative can only be used noncomputationally in proofs.

The difference between these notions of subset is more pronounced with a more involved example. Suppose that we consider the following two types defining integer functions having zeros.

$$F_1 = \{f{:}int \rightarrow int|\ some\ y{:}int.f(y) = 0\ in\ int\}$$
$$F_2 = (f{:}int \rightarrow int\#\ some\ y{:}int.f(y) = 0\ in\ int)$$

It is easy to define a function g mapping F_2 into int such that for all p in F_2, $1of(p)(g(p)) = 0\ in\ int$. (Notice that p is a pair $< f, e >$, where $f{:}int \rightarrow int$ and e is a proof that f has a zero, so $1of(p) = f$.) That is, the function g simply picks out the witness for the quantifier in $some\ y{:}int f(y) = 0\ in\ int$. There is no such function h from F_1 because the only input to h is the function f, so in order to find a zero value h would need to search through int for a zero. In the language described so far there is no unbounded search operator to use in defining h.[5]

One can think of the set constructor, $\{x{:}A|B\}$, as serving two purposes. One is to provide a subtype concept; this purpose is shared with $(x{:}A\#B)$. The other is to provide a mechanism for hiding information to simplify computation.

The quotient operator builds a new type from a given base type, A, and an equivalence relation, E, on A. The syntax for the quotient is $(x, y){:}A//E$.

[5] In chapter 12 we introduce a concept of *partial function* that will allow us to define $h{:}F_1 \rightarrow int$ such that $f(h(f)) = 0\ in\ int$ using an unbounded search. The search is guaranteed to terminate because of the information about f.

In this type the equality relation is E, so the quotient operator is a way of redefining equality in a type.

In order to define a function $f{:}(x,y){:}A//E \rightarrow B$ one must show that the operation respects E, that is, $E(x,y)$ implies $f(x) = f(y)$ *in* B. Although the details of showing f is well–defined may be tedious, we are guaranteed that concepts defined in terms of f and the other operators of the theory respect equality on $(x,y){:}A//E$. As an example of quotienting changing the behavior of functions, consider defining the integers modulo 2 as a quotient type.

$$N_2 = (x,y){:}int//(x \; mod \; 2 = y \; mod \; 2 \; in \; int)$$

We can now show that successor is well–defined on N_2 by showing that if $x \; mod \; 2 = y \; mod \; 2 \; in \; int$ then $x + 1 \; mod \; 2 = y + 1 \; mod \; 2 \; in \; int$. On the other hand, the maximum function is not well–defined on N_2 because $0 = 2 \; in \; N_2$ but $max(1,0) = 1$ and $max(1,2) = 2$, meaning that it is not the case that $max(1,0) = max(1,2) \; in \; N_2$.

2.5 Semantics

This section is included for technical completeness; the beginning reader may wish to skip this section on a first reading. Here we shall consider only briefly the Nuprl semantics. The complete introduction appears in section 8.1. The semantics of Nuprl are given in terms of a system of computation and in terms of criteria for something being a *type*, for *equality* of types, for something being a *member* of a given type and for *equality* between members *in* a given type.

The basic objects of Nuprl are called *terms*. They are built using variables and operators, some of which bind variables in the usual sense. Each occurrence of a variable in a term is either free or bound. Examples of free and bound variables from other contexts are:

- Formulas of predicate logic, where the quantifiers $(\forall, \exists)$ are the binding operators. In $\forall x.\,(P(x) \& Q(y))$ the two occurrences of x are bound, and the occurrence of y is free.

- Definite integral notation. In $\int_x^y \sin x \, dx$ the occurrence of y is free, the first occurrence of x is free, and the other two occurrences are bound.

- Function declarations in Pascal. In

```
function Q(y:integer):integer;
  function P(x:integer):integer; begin P:=x+y end ;
  begin Q:=P(y) end ;
```

all occurrences of `x` and `y` are bound, but in the declaration of `P x` is bound and `y` is free.

By a *closed* term we mean a term in which no variables are free. Central to the definitions of computation in the system is a procedure for *evaluating* closed terms. For some terms this procedure will not halt, and for some it will halt without specifying a result. When evaluation of a term does specify a result, this value will be a closed term called a *canonical* term. Each closed term is either canonical or *noncanonical*, and each canonical term has itself as value.

Certain closed terms are designated as *types*; we may write "T type" to mean that T is a type. Types always evaluate to canonical types. Each type may have associated with it closed terms which are called its *members*; we may write "$t \in T$" to mean that t is a member of T. The members of a type are the (closed) terms that have as values the canonical members of the type, so it is enough when specifiying the membership of a type to specify its canonical members. Also associated with each type is an equivalence relation on its members called the *equality in* (or *on*) that type; we write "$t = s \in T$" to mean that t and s are members of T which satisfy equality in T. Members of a type are equal (in that type) if and only if their values are equal (in that type).

There is also an equivalence relation $T = S$ on types called *type equality*. Two types are equal if and only if they evaluate to equal types. Although equal types have the same membership and equality, in Nuprl some *un*equal types also have the same membership and equality.

We shall want to have simultaneous substitution of terms, perhaps containing free variables, for free variables. The result of such a substitution is indicated thus:

$$t[t_1, \ldots, t_n/x_1, \ldots, x_n],$$

where $0 \leq n$, $x_1, \ldots, x_n$ are variables, and $t_1, \ldots, t_n$ are the terms substituted for them in t.

What follows describes inductively the type terms in Nuprl and their canonical members. We use `typewriter` font to signify actual Nuprl syntax. The integers are the canonical members of the type `int`. There are denumerably many atom constants (written as character strings enclosed in quotes) which are the canonical members of the type `atom`. The type `void` is empty. The type $A \mid B$ is a disjoint union of types A and B. The terms `inl(`a`)` and `inr(`b`)` are canonical members of $A \mid B$ so long as $a \in A$ and $b \in B$. (The operator names `inl` and `inr` are mnemonic for "inject left" and "inject right".) The canonical members of the cartesian product type $A\#B$ are the terms `<`a`,`b`>` with $a \in A$ and $b \in B$. If $x{:}A\#B$ is a type then A is closed (all types are closed) and only x is free in B. The canonical members of a type $x{:}A\#B$ ("dependent product") are the terms `<`a`,`b`>` with $a \in A$

and $b \in B[a/x]$. Note that the type from which the second component is selected may depend on the first component. The occurrences of x in B *become bound* in $x:A\texttt{\#}B$. Any free variables of A, however, remain free in $x:A\texttt{\#}B$. The x in front of the colon is also bound, and indeed it is this position in the term which determines which variable in B becomes bound. The canonical members of the type A `list` represent lists of members of A. The empty list is represented by `nil`, while a nonempty list with head a and tail b is represented by $a\texttt{.}b$, where b evaluates to a member of the type A `list`.

A term of the form $t\texttt{(}a\texttt{)}$ is called an *application* of t to a, and a is called its *argument*. The members of type $A\texttt{->}B$ are called *functions*, and each canonical member is a *lambda term*, $\backslash x\texttt{.}b$, whose application to any member of A is a member of B. The canonical members of a type $x:A\texttt{->}B$, also called *functions*, are lambda terms whose applications to any member a of A are members of $B[a/x]$. In the term $x:A\texttt{->}B$ the occurrences of x free in B become bound, as does the x in front of the colon. For these function types it is required that applications of a member to equal members of A be equal in the appropriate type.

The significance of some constructors derives from the representation of propositions as types, where the proposition represented by a type is true if and only if the type is inhabited. The term $a\texttt{<}b$ is a type if a and b are members of `int`, and it is inhabited if and only if the value of a is less than the value of b. The term $\texttt{(}a\texttt{=}b\ \texttt{in}\ A\texttt{)}$ is a type if a and b are members of A, and it is inhabited if and only if $a = b \in A$. The term $\texttt{(}a\texttt{=}a\ \texttt{in}\ A\texttt{)}$ is also written $\texttt{(}a\ \texttt{in}\ A\texttt{)}$; this term is a type and is inhabited if and only if $a \in A$.

Types of form $\{A \mid B\}$ or $\{x:A \mid B\}$ are called *set types*. The set constructor provides a device for specifying subtypes; for example, $\{\texttt{x:int|0<x}\}$ has just the positive integers as canonical members. The type $\{A \mid B\}$ is inhabited if and only if the types A and B are, and if it is inhabited it has the same membership as A. The members of a type $\{x:A \mid B\}$ are the members a of A such that $B[a/x]$ is inhabited. In $\{x:A \mid B\}$, the x before the colon and the free x's of B become bound.

Terms of the form $A\texttt{//}B$ and $\texttt{(}x\texttt{,}y\texttt{)}:A\texttt{//}B$ are called *quotient types*. $A\texttt{//}B$ is a type only if B is inhabited, in which case $a = a' \in A\texttt{//}B$ exactly when a and a' are members of A. Now consider $\texttt{(}x\texttt{,}y\texttt{)}:A\texttt{//}B$. This term denotes a type exactly when A is a type, $B[a, a'/x, y]$ is a type for a and a' in A, and the relation $\exists b.b \in B[a, a'/x, y]$ is an equivalence relation over A in a and a'. If $\texttt{(}x\texttt{,}y\texttt{)}:A\texttt{//}B$ is a type then its members are the members of A; the difference between this type and A only arises in the equality between elements. Briefly, $a = a' \in x, y{:}A\texttt{//}B$ if and only if a and a' are members of A and $B[a, a'/x, y]$ is inhabited. In $(x, y):A\texttt{//}B$ the x and y before the colon and the free occurrences of x and y in B become bound.

Now consider equality on the types already discussed. Members of `int` are equal (in `int`) if and only if they have the same value. The same goes for

type `atom`. Canonical members of $A \mid B$, $A\#B$, $x:A\#B$ and A `list` are equal if and only if they have the same outermost operator and their corresponding immediate subterms are equal (in the corresponding types). Members of A`->`B or $x:A$`->`B are equal if and only if their applications to any member a of A are equal in $B[a/x]$. The types a`<`b and (`(`a`=`b `in` A`)` have at most one canonical member, namely `axiom`, so equality is trivial. Equality in $\{x:A \mid B\}$ is just the restriction of equality in A to $\{x:A \mid B\}$, as is the equality for $\{A \mid B\}$.

Now consider the so–called *universes*, `U`k (k positive). The members of `U`k are types. The universes are cumulative; that is, if j is less than k then membership and equality in `U`j are just restrictions of membership and equality in `U`k. `U`k is closed under all the type–forming operations except formation of `U`i for i greater than or equal to k. Equality in `U`k is the restriction of type equality to members of `U`k.

With the type theory in hand we now turn to the Nuprl proof theory. The assertions that one tries to prove in the Nuprl system are called *judgements*. They have the form

$$x_1:T_1\,,\ldots,x_n:T_n \texttt{ >> } S \texttt{ [ext } s\texttt{]},$$

where $x_1,\ldots,x_n$ are distinct variables and $T_1,\ldots,T_n,S$ and s are terms (n may be 0), every free variable of T_i is one of $x_1,\ldots,x_{i-1}$, and every free variable of S or of s is one of $x_1,\ldots,x_n$. The list $x_1:T_1\,,\ldots,x_n:T_n$ is called the *hypothesis list* or *assumption list*, each $x_i:T_i$ is called a *declaration* (of x_i), each T_i is called a *hypothesis* or *assumption*, S is called the *consequent* or *conclusion*, s the *extract term* (the reason will be seen later), and the whole thing is called a *sequent*.

The criterion for a judgement being *true* is to be found in the complete introduction to the semantics.[6] Here we shall say a judgement

$x_1:T_1\,,\ldots,x_n:T_n$ `>>` S `[ext` s`]` is *almost true* if and only if
$\forall t_1,\ldots,t_n.\ s[t_1,\ldots,t_n/x_1,\ldots,x_n] \in S[t_1,\ldots,t_n/x_1,\ldots,x_n]$
if $\forall i < n.\ t_{i+1}[t_1,\ldots,t_i/x_1,\ldots,x_i] \in T_{i+1}[t_1,\ldots,t_i/x_1,\ldots,x_i]$

That is, a sequent like the one above is almost true exactly when substituting terms t_i of type T_i (where t_i and T_i may depend on t_j and T_j for $j < i$) for the corrseponding free variables in s and S results in a true membership relation between s and S.

It is not always necessary to declare a variable with every hypothesis in a hypothesis list. If a declared variable does not occur free in the conclusion, the extract term or any hypothesis, then the variable (and the colon following it) may be omitted.

[6] Section 8.1, page 141.

In Nuprl it is not possible for the user to enter a complete sequent directly; the extract term must be omitted. In fact, a sequent is never displayed with its extract term. The system has been designed so that upon completion of a proof, the system automatically provides, or *extracts*, the extract term. This is because in the standard mode of use the user tries to prove that a certain type is inhabited without regard to the identity of any member. In this mode the user thinks of the type (that is to be shown inhabited) as a proposition and assumes that it is merely the truth of this proposition that the user wants to show. When one does wish to show explicitly that $a = b \in A$ or that $a \in A$, one instead shows the type `(`a `=` b `in` A`)` or the type `(`a `in` A`)` to be inhabited.[7]

The system can often extract a term from an incomplete proof when the extraction is independent of the extract terms of any unproven claims within the proof body. Of course, such unproven claims may still contribute to the truth of the proof's main claim. For example, it is possible to provide an incomplete proof of the untrue sequent `>> 1<1 [ext axiom]`, the extract term `axiom` being provided automatically.

Although the term extracted from a proof of a sequent is not displayed in the sequent, the term is accessible by other means through the name assigned to the proof in the user's library. In the current system proofs named in the user's library cannot be proofs of sequents with hypotheses.

2.6 Relationship to Set Theory

Type theory is similar to set theory in many ways, and one who is unfamiliar with the subject may not see readily how to distinguish the two. This section is intended to help. A type is like a set in these respects: it has elements, there are subtypes, and we can form products, unions and function spaces of types. A type is unlike a set in many ways too; for instance, two sets are considered equal exactly when they have the same elements, whereas in Nuprl types are equal only when they have the same structure. For example, `void` and `{x:int|x<x}` are both types with no members, but they are not equal.

The major differences between type theory and set theory emerge at a global level. That is, one cannot say much about the difference between the type `int` of integers and the set Z of integers, but one can notice that in type theory the concept of equality is given with each type, so we write `x=y in int` and `x=y in int#atom`. In set theory, on the other hand, equality is an absolute concept defined once for all sets. Moreover, set theory can be organized so that all objects of the theory are sets, while type theory

[7] Recall that the term `(`a `=` b `in` A`)` is a type whenever $a \in A$ and $b \in A$ and is inhabited just when $a = b \in A$. As a special case the term `(`a `in` A`)`, which is shorthand for `(`a `=` a `in` A`)`, is a type and is inhabited just when $a \in A$.

requires certain primitive elements, such as individual integers, pairs, and functions, which are not types. Another major global difference between the theories concerns the method for building large types and large sets. In set theory one can use the union and power set axioms to build progressively larger sets. In fact, given any indexed family of sets, $\{ S(x) \mid x \in A \}$, the union of these sets exists. In type theory there are no union and power type operators. Given a family of types `S(x)` indexed by `A`, they can be put together into a disjoint union, `x:A#S(x)`, or into a product, `x:A->S(x)`, but there is no way to collect only the members of the `S(x)`. Large unstructured collections of types can be obtained only from the universes, `U1`,`U2`,... .

Another global difference between the two theories is that set theory typically allows so–called impredicative set formation in that a set can be defined in terms of a collection which contains the set being defined. For instance, the subgroup H of a group G generated by elements $h_1, \ldots, h_n$ is often defined to be the least among all subgroups of G containing the h_i. However, this definition requires quantifying over a collection containing the set being defined. The type theory presented here depends on no such impredicative concepts.

For set theories, such as Myhill's CST [Myhill 75], which do not employ impredicative concepts, Peter Aczel [Aczel 77, Aczel 78] has shown a method of defining such theories in a type theory similar to Nuprl.

Both type theory and set theory can play the role of a foundational theory. That is, the concepts used in these theories are fundamental. They can be taken as irreducible primitive ideas which are explained by a mixture of intuition and appeal to defining rules. The view of the world one gets from inside each theory is quite distinct. It seems to us that the view from type theory places more of the concepts of computer science in sharp focus and proper context than does the view from set theory.

2.7 Relationship to Programming Languages

In many ways the formalism presented here will resemble a functional programming language with a rich type structure. The functions of Nuprl are denoted by lambda expressions, written `\x.t`, and correspond to programs. The function terms do not carry any type information, and they are evaluated without regard to types. This is the evaluation style of ML [Gordon, Milner, & Wadsworth 79], and it contrasts with a style in which some type correctness is checked at runtime (as in PL/I). The programs of Nuprl are rather simple in comparison to those of modern production languages; there is no concurrency, and there are few mechanisms to optimize the evaluation (such as alternative parameter passing mechanisms, pointer allocation schemes, etc.).

The type structure of Nuprl is much richer than that of any programming language; for example, no such language offers dependent products, sets,

quotients and universes. On the other hand, many of the types and type constructors familiar from languages such as Algol 68, Simula 67, Pascal and Ada are available in some form in Nuprl. We discuss this briefly below.

A typical programming language will have among its primitive types the integers, *int*, booleans, *bool*, characters, *char*, and real numbers (of finite precision), *real*. In Nuprl the type of integers, `int`, is provided; the booleans can be defined using the set type as `{x:int | x=0 in int or x=1 in int}`, the characters are given by `atom`, and various kinds of real numbers can be defined (including infinite precision), although no built–in finite precision real type is as yet provided.

Many programming languages provide ways to build tuples of values. In Algol the constructor is the *structure*; in Pascal and Ada it is the *record* and has the form `RECORD` x:A, y:B `END` for the product of types A and B. In Nuprl such a product would be written `A#B` just as it would be in ML.

In Pascal the *variant record* has the following form.

```
RECORD
    CASE kind:(RECT,TRI,CIRC) of
        RECT:(w,h:real);
        CIRC:(r:real);
        TRI :(x,y,a:real)
END
```

The elements of this type are either pairs, triples or quadruples, depending on the first entry. If the first entry is `RECT` then there are two more components, both reals. If the first entry is `CIRC` then there is only one other which is a real; if it is `TRI` then there are three real components. One might consider this type a discriminated union rather than a *variant record*. In any case, in Nuprl it is defined as an extension of the product operator which we call a *dependent product*. If `real` denotes the type of finite precision reals, and if the Pascal type `(RECT,CIRC,TRI)` is represented by the type consisting of the three atoms `"RECT"`,`"CIRC"` and `"TRI"`, and if the function `F` is defined as

```
F("RECT") = real#real
F("CIRC") = real
F("TRI")  = real#real#real
```

then the following type represents the variant record.

```
i:("RECT","CIRC","TRI")#F(i)
```

In Nuprl, as in Algol 68, it is possible to form directly the *disjoint union*, written $A|B$, of two types A and B. This constructor could also be used to define the variant record above as `real#real|(real|real#real#real)`.

One of the major differences between Nuprl types and those of most programming languages is that the type of functions from A to B, written

A->B, denotes exactly the *total* functions. That is, for every input a of type A, a function in A->B must produce a value in B. In Algol the type of functions from `A` to `B`, say `PROC(x:A)B`, includes those procedures which may not be well–defined on all inputs of `A`; that is, they may diverge on some inputs.

In contrast to the usual state of affairs with programming languages the semantics of Nuprl "programs" is completely formal. There are rules to settle such issues as when two types of programs are equal, when one type is a subtype of another, when a "program" is correctly typed, etc. There are also rules for showing that "programs" meet their specifications. Thus Nuprl is related to programming languages in many of the ways that a programming logic or program verification system is.

Chapter 3

Statements and Definitions in Nuprl

This chapter will explain how to write definitions and statements with the Nuprl system, and in doing so it will introduce a minimal set of system commands to the beginning user and explain how the type theory introduced in chapter 2 can be used to express mathematical propositions. The first four sections of the chapter concentrate on discussing the system commands and some relevant syntactic features of the Nuprl logic. The rest of the chapter describes the conceptual issues involved in using type theory as a general–purpose language for expressing mathematics. These conceptual issues are discussed via examples drawn from four areas of mathematics: logic (both constructive and classical), number theory, algebra and set theory. (Chapter 11 contains a summary of definitions and theorems from other areas of mathematics.) It is intended that this chapter be read while experimenting with the system. Throughout this chapter the `typewriter font` represents text that is actually used in the Nuprl system, while the *mathematical font* is used in general discussions.

3.1 Overview of the Nuprl Environment

The Nuprl system provides an interactive medium which is accessed via a screen divided into various windows[1] and a keyboard/mouse which allows communication with the system. The windows represent regions of the system with specialized roles in the interactive process; the different kinds of windows are listed in figure 3.1. The terminal is used to enter definitions,

[1] The windows are part of Nuprl and are not part of the window mechanism that may come with the machine on which Nuprl is implemented.

- The Command Window
- The Library Window
- The Refinement Editor (red) (also called the proof editor)
- The Text Editor (ted)

Figure 3.1: Windows Used in the Nuprl System

commands, proof steps and so forth into the appropriate window of the system. Commands that allow movement between windows and modification of the *view* of the environment consist of control sequences or mouse commands. In this section these features are described in as much detail as is needed for the purposes of this chapter. Chapter 7 contains a comprehensive discussion of system features.

The Terminal

Nuprl is designed to run on a terminal with an ordinary keyboard with a control key, a keypad with nine keys arranged as in figure 3.2 and, optionally, a mouse. In the subsequent discussion we shall assume that these are available, and we shall assume that the mouse comes with three buttons called L, M and R (for left, middle and right, respectively) and is activated by clicking one of these buttons. However, one may simulate moving the mouse by using the keypad as follows. The arrow keys are used to move the cursor; if the arrow keys are used alone then the cursor moves in the direction of the arrow by either a single character or a single line. The cursor can be made to move in larger steps by pressing the key LONG before the arrow key. The COMMAND key is used for returning to the command window momentarily while in the middle of editing. For any particular terminal one may have to simulate the keypad and/or the mouse on the keyboard using control keys in combination with ordinary keys.

The mouse is used for moving the cursor on the screen, for opening windows and for moving windows. An arrow on the screen called the *mouse cursor* responds directly to movements of the mouse. We shall also have occasion to refer to the cursor which responds to the keyboard and keypad commands as *the* cursor. To move the cursor, one places the mouse cursor at the desired position and clicks button R.

DIAG	UP	JUMP
LEFT	LONG	RIGHT
SEL	DOWN	MOUSE

Figure 3.2: The Keypad

Windows

Users may control the placement and size of all windows. To alter a window, one places the cursor in it and invokes a menu by either clicking M on the mouse or simulating the mouse actions with the keypad in the following fashion. First, one enters mouse mode by pressing the MOUSE key (see figure 3.2). Next, using the arrow keys and, optionally, the LONG key, the user positions the mouse in the desired window and presses MOUSE again, thereby causing a menu of operations to appear. The option desired is selected by positioning the cursor next to the option in the displayed menu and pressing the MOUSE key; when a menu item is selected an asterisk appears on the left of the item. To leave mouse mode one uses the DIAG key. In subsequent discussion, the commands described are mouse commands rather than keypad commands. The translation to keypad commands is exactly as described in the discussion above.

The basic window manipulating commands available in the menu are `size` and `move`. The position of a window is determined by its upper left corner and the size by its lower right corner. To change position, for example, one selects the `move` option in the menu (by placing the cursor on it and clicking M) and then positions the cursor at the new location for the upper left corner and exits mouse mode by clicking M. Similarly, the size of the window can also be altered. Windows are closed by holding down the control key and pressing D. (The notation ↑D is used to represent this key combination.) When a window is closed the cursor returns to the window it had been in when the new window was opened. Several other window management commands are available in mouse mode; these commands are typically needed when the user has several edit windows in use concurrently and needs to hide some of them temporarily. A complete discussion of these commands appears in chapter 7.

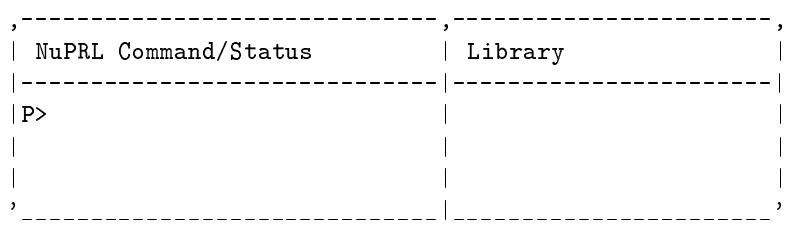

Figure 3.3: The Initial Display

Commands and Status

Commands are used to enter objects in the library, to invoke editors, to check definitions, to save a session, to load back a saved session and to exit the system.[2] Commands are entered into the command window by typing the text of the command at the prompt `P>` and ending with a carriage return. The system responds to commands by printing messages in the command window.

Starting Up

Nuprl is implemented in Lisp as a function; therefore, entering the Nuprl environment entails first invoking Lisp. Currently there are two implementations, one running on a Symbolics Lisp Machine and one written in Franz Lisp and running on Unix. One invokes the Nuprl environment using commands which load the system in the appropriate Lisp environment. The system comes up with a two–window display as in figure 3.3.

`P>` is the top–level prompt from the command module. Depending on its *mode*, the window may display one of several prompts The modes and their corresponding prompts are:

- `P>` Top–level Command Interpreter
- `I>` Definition Facility
- `M>` Metalanguage Interpreter
- `E>` Evaluation Mechanism

In this chapter we discuss only the first two modes. The third mode is discussed in chapters 6 and 9, the fourth is discussed in chapters 5 and 7.

[2] Other commands needed for proving theorems, evaluating terms, etc., will be discussed in later chapters.

Finishing Up

After a session with Nuprl the results can be saved in the file system using the `dump` command as illustrated below. These files can then be reloaded with the `load` command. To end a session one uses the command `exit`.

3.2 Libraries

The library is the part of the system where objects of various kinds are stored. The system presents the user with a view of a portion of the library through the library window; the library window shows objects by name, *kind* (theorem, definition or ML), and status. For theorems, the statement of the theorem is also visible.

The `scroll` command alters the user's view of the library. The form of the command is `scroll` *number*. This command moves the view *down* by the indicated number of objects; to move the view upward one uses the command `scroll up` *number*.[3]

Entry of Objects

One creates definitions and theorems in two stages. First, one creates a slot in the library where the new object is to be stored by issuing the `create` command followed by the name of the object, the kind of the object and a place for the name (`top`, `bottom` or `before` or `after` another name). The general form of the "create" command is `create` *name kind place*. An example of such a command follows.[4]

```
create t1 thm top
```

The result is the display of

```
? t1
THM
```

as a new library entry. The system does not recognize this as an object but rather as a place where the object $t1$ will eventually reside. The mark `?` is called a *status mark*. The status of an object can be any one of the following:

- Incomplete: This applies only to theorems and means that the proof of the theorem is not completed. The status mark is `#`.
- Good: This applies to any object and signifies that the object has been checked and found correct. In the case of a theorem, for example, it means that it has been proven. The status mark is $*$.

[3] The keyword `scroll` can be abbreviated by `scr`.
[4] The keyword `create` can be abbreviated by `cr`.

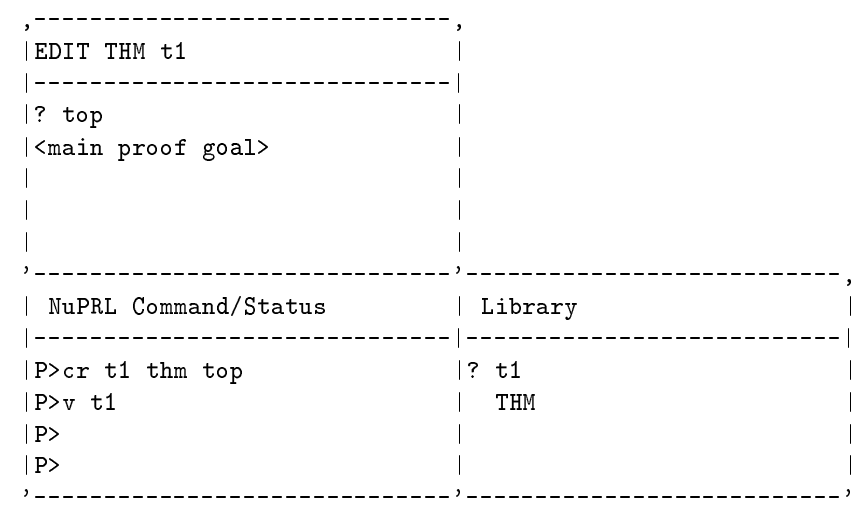

Figure 3.4: Appearance of the Screen after the Proof Editor Is Invoked

- Raw: The object has not yet been checked. The status mark is `?`.
- Bad: The object has been checked and been found incorrect. The status mark is `-`.

The second stage in creating an object involves invoking the proof and text editors on the library object created in the first stage described above. One does so by asking to *view* the object: `view` *name*.[5] Issuing `view t1` results in a new window as shown in figure 3.4. If an object were present, it would appear in the window; at this stage, however, no object has been created, so one sees `? top`. Since the object being edited is a theorem, the new window produced is a view of the proof editor, or *refinement* editor; if the object were a definition, then the system invokes the text editor. Definitions will be discussed at greater length in the next section. The refinement editor is a editor for tree–structured objects and is used for building proof trees. It is discussed in detail in chapter 4; at present it may be viewed as the place where theorems are displayed. To enter a theorem statement at this stage, one invokes the text editor (ted) by pressing the select key, SEL, or by clicking the left key on the mouse with the mouse cursor in `<main proof goal>`. This opens a new window as shown in figure 3.5.[6] These windows

[5] The keyword `view` can be abbreviated by `v`.

[6] The new window is dependent on the first window, so the first window cannot be deleted until the text editor window is closed. The first window can, however, be moved

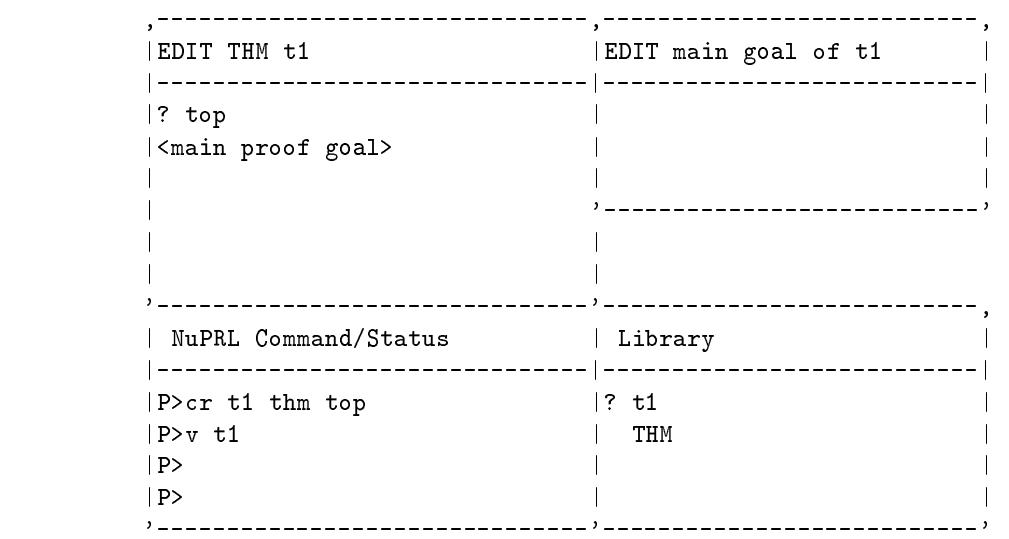

Figure 3.5: Appearance of the Screen after the Text Editor Is Invoked

can of course be repositioned, resized or *elided* (moved off to the edge of the screen).

One can now enter text into the text editor. As an example, suppose one were to type `>> 0=0 in int`. This becomes the statement of the theorem `t1` and makes the trivial assertion that $0 = 0$. The >> symbol is our notation for the logical turnstile,[7] and the phrase `in int` appears because every statement of equality must be with respect to some type. One can now leave the text editor by pressing ↑D; the statement now appears in the proof–editing window. If one were trying to prove this theorem, one would at this stage start using the proof facilities to build the proof; this process is discussed in detail in chapter 4. For the present, one can simply exit from this window with ↑D. The result is a new library entry for `t1`. The screen now looks like figure 3.6. The status mark `#` indicates that this is an incomplete but syntactically correct object.

Saving Results

The library facility in Nuprl also allows one to save theorems, definitions and other objects produced in a session. For example, if it were desired to

off the screen by using the window–moving commands. Thus windows can be temporarily hidden.

[7] The notation >> is used instead of the more usual turnstile $\vdash$ since the former is available on most terminals.

```
,-------------------------------,--------------------,
| NuPRL Command/Status          | Library            |
|-------------------------------|--------------------|
|P>cr t1 thm top                |# t1                |
|P>v t1                         |  THM >>0=0 in int  |
|P>                             |                    |
|                               |                    |
'-------------------------------'--------------------'
```

Figure 3.6: Appearance of Screen after Creating a Theorem

save the theorem `t1` of the previous example in a file named `library-name`, one would issue the command

```
dump t1 to library-name.
```

The theorem (together with its proof, if it has been proved) is stored in the file named in the `dump` command. If there is a library of several objects, say

```
#t1
THM ...

#t2
THM
.
.
#tn
THM
```

then the entire library can be saved by the command

```
dump t1-tn to library-name
```

or by the command

```
dump first-last to library-name.
```

Any segment of a library, say `t5` to `t9`, can be saved by the command

```
dump t5-t9 to library-name.
```

If one later wishes to retrieve a stored library, one issues the command

```
load place from library-name,
```

where *place* refers to a place in the active system library and may be either `top`, `bottom`, `before` *name* or `after` *name*.

3.3 The Text Editor

Entering Text

The text editor is the basic tool which one uses to express anything in the Nuprl system. Even when using the proof editor to construct tree–structured proofs the text editor is constantly needed to communicate with the system; for instance, one uses it to tell Nuprl which proof rule to use. Entry of text in this editor is modeless; one types characters and deletes them as if using an ordinary keyboard. There are no special commands to "save" text or to "write" text. Text that exists in a window will be read and put in the library automatically when the window is closed.

Within the text editor one moves the cursor by using the mouse or the arrow keys on the keypad. Long moves of the cursor can be made by striking LONG once or twice before the arrow keys. One LONG preceding a vertical arrow key moves the cursor up or down four lines; if used before a horizontal arrow key LONG moves the cursor left or right four positions. Two LONG's move the cursor to the end of the line if they precede a horizontal arrow key; if they precede a vertical arrow key they move the cursor up or down by a screenful. To move the cursor from one edit window to another the JUMP key is used.

Text is deleted using the delete key on the terminal or the `kill` key, KILL. The delete key removes the character to the left of the cursor, while the kill key removes the character at which the cursor is positioned. An entire region of text can be erased using the KILL key on a selected region. To select a region, the cursor is positioned at the beginning of the region and the SEL key is pressed or L is clicked. The cursor is then positioned at the end of the region and SEL is pressed again or L is clicked.[8] If the KILL key is used now the entire region is deleted and stored in a special buffer called the *kill buffer*. At most one region can be selected at a time; if three endpoints are selected only the last two are remembered. A kill operation causes the system to "forget" both the endpoints.

Copying Text

Text can be copied from one place to another in an edit window by selecting the region of text to be copied using the SEL key as before, placing the cursor where the selected text is to appear, and striking ↑C. One may also copy text between edit windows by selecting text, jumping to the appropriate window using the JUMP key, placing the cursor at the appropriate spot and striking ↑C.[9] As with a kill operation, when a copy is performed the endpoints of the selected region are forgotten. If ↑C is pressed while no region is selected

[8] Actually one can select the endpoints in either order.

[9] One must *not* be in mouse mode while doing this.

then the contents of the kill buffer are copied at the cursor position. Using the copy operation in conjunction with the kill buffer does not destroy the contents of the kill buffer, so this is a convenient way to make multiple copies of a given piece of text.

Definitions

Nuprl provides a powerful and flexible definition facility which allows users to define their own notation. The general form of a definition follows:

$$text\langle formal\ name : description\rangle text == text.$$

The left–hand side of a definition gives the form in which the definition is displayed when invoked; it gives the syntax of the extended notation. The right–hand side expresses the meaning of the new notation in terms of existing notation. The occurrences of `<formal name:description>` are parameters of the new notation. It is essential to use the angle brackets to enclose the parameters of a definition; otherwise, the symbol intended as a parameter will be used as a literal symbol every time the definition is instantiated. If the left angle bracket, <, is used as the *less than* symbol of number theory then it must be preceded by an escape character, \ in the text editor, to prevent the system from interpreting the < as the start of a parameter to the definition. Most of the definitions shown in this chapter have the escape character removed in order to make the definitions more readable.

A definition is created in much the same way as a theorem is created. One opens a place in the library using the command `create` *name* `def` *place*. The object is then viewed by the command `view` *name*, which in the case of a definition invokes the text editor rather than the proof editor. The form of the definition is now entered directly in the text editor. To illustrate this process suppose one wishes to define the logical connective "and" in terms of the type constructor `#`. A place in the library is first created using the `create` command; this command would include a name, say `and`, for the definition being created. The object `and` would then be viewed, and the text editor would thereby be invoked. In the text editor one would type

```
<p:prop> & <q:prop> == ( (<p>) # (<q>) ).
```

Upon exit with ↑D the system places the definition in the library.

Before a definition can be used it must be *checked*; this is done by issuing the command `check` $name-of-def$. The system updates the status of the definition to either *good*, indicated by the status mark $*$, or *bad*, indicated by the status mark –. Returning to our example, after `and` is checked, it extends the notation by introducing the new symbol `&`, which is defined in terms of the existing type constructor `#`.

To use a definition in creating another object, one types ↑I in the text editor while editing the new object; the cursor will be put in the command window with the prompt being `I>`. The definition name is typed followed by a RETURN; the left–hand side of the definition appears where the cursor is positioned. Angle brackets containing the description will appear in the positions where a parameter is expected. In the example above, using the definition would result in the appearance of `[<prop> & <prop> ]`. The parameters, `p` and `q` in the example above, can now be entered by moving the cursor to the places indicated by angle brackets and typing the desired parameters. As the instantiations for the parameters are typed the angle brackets vanish. It is possible to use another definition while instantiating the parameters of a definition.

If one wishes to delete a definition then the kill command must be used. To delete an instance of a definition in another object, one positions the cursor at the beginning of the definition and hits the kill key. The easiest way to ensure that the cursor is correctly positioned is to use *bracket mode*. This is a display mode in which instantiated definitions are shown with brackets surrounding them and is normally the default mode. If the bracket display is not desired, bracket mode can be turned off by using ↑B. This key serves as a toggle between the two display modes.

3.4 Syntactic Issues

The Nuprl system reads text from an edit window when that window is closed by typing ↑D. The text is read by a parsing routine which will accept text that conforms to syntactic rules described in detail in section 8.1. These syntactic rules do not completely characterize the meaningful entities; thus the parser may accept text which cannot be viewed as a term. Essentially this means that the parser does not perform any type checking. Text which is accepted by the parser is called *readable text*. Text which is meaningful in the sense of the semantic account in chapter 2 is called readable and well–formed. Each well–formed piece of text is called a *term*. Note that our notion of well–formedness is *semantic*, unlike the traditional usage of the word "well-formed". The rest of this section contains a brief discussion of syntactic issues that are of immediate relevance.

Theorems

In Nuprl all theorems consist of goal statements, where a goal statement is similar to sequents used in natural deduction style inference systems. The form of a goal statement is H >> T, where H is a hypothesis list and T is a formula in the Nuprl logic. The symbols >> separate the hypotheses from the conclusion and thus correspond to the $\vdash$ symbol used in logic. Top–level goals

must have empty hypothesis lists; this prevents the user from introducing inconsistent hypotheses. Theorems also cannot contain free variables; in the terminology of chapter 2, this means that all top–level goals must contain closed terms to the right of the >> symbol.

Expressions Formed by Using Type Constructors

The basic components of the type theory employed by the Nuprl system are the *atomic types* `int`, `atom` and `void`. In addition to the atomic types, there is an integer–indexed family of predefined types called *universes*, $U_1, U_2, \ldots$. Universes are types whose canonical elements are types. All the atomic types are members of U_1, and U_i is a member of U_{i+1}. More complex types are built from the atomic types and the universes by using *type constructors*, operators which take types and build new types from them. The type constructors used in Nuprl are[10]

`#`	product type constructor;
`->`	function type constructor;
`|`	disjoint sum type constructor;
`x:A#B`	dependent product type constructor;
`x:A->B`	dependent function type constructor;
`A list`	list constructor;
`{x:A|B}`	subset constructor;
`x,y:A//B`	quotient type constructor.

The meaning of these type constructors has been briefly discussed in chapter 2 and will be discussed in detail in chapter 8. The first three type constructors are binary, and all have equal precedence. They all associate to the right. We can also form types by using the forms "*element* `in` *type*" and "*element* `=` *element* `in` *type*". As explained in chapter 2, these are two of the basic judgement forms. The symbols `in` and `=` can be viewed as a type constructor for the purposes of the present discussion. Viewed in this way, `in` associates to the left. The symbol `=` binds more tightly than the symbol `in`. The type constructors in the list above also bind more tightly than does `in`. The binding and scope conventions are described in chapter 2.

3.5 Stating Theorems

The discussion so far has focused on the mechanics of using Nuprl. The rest of this chapter will concentrate on the conceptual issues that arise when one uses Nuprl to express formal mathematics. The discussion will continue to make reference to the mechanics of using Nuprl so that interested readers can take the opportunity to experiment with the system while familiarizing

[10] Other type constructors are being actively investigated. Some of these will be discussed in chapter 12.

themselves with the logic of Nuprl. No attempt will be made to discuss proof techniques in this chapter; these will be covered in chapter 4.

All theorems in Nuprl have the form "`>>` *term*", where `>>` separates assumptions from the goal. One can read `>>int in U1` as "prove that type int is in U_1 from no assumptions."[11] In the next chapter we will see how various rules generate assumptions.

The most important aspect of the Nuprl system is the constructive nature of the type theory that underlies the system. A beginner should therefore be careful to make certain that the theorems being expressed are indeed constructively true. Even if a particular theorem is constructively true, the reader should remember that the proof will involve carrying out the actual constructions needed to exhibit the truth of the theorem. Thus several classically trivial theorems (or axioms) become relatively difficult. The benefit of the constructive viewpoint, of course, is that the resulting proofs become computationally significant. The heart of the constructive interpretation is contained in the propositions–as–types principle, which is discussed later in this section. It should be noted here that the constructive nature of the Nuprl type system in no way inhibits the possibility of modeling classical logic, and in fact we do so in the next section. The use of a constructive type theory for the Nuprl system is in sharp contrast to other systems for theory development, notably the LCF system.

Expressing Concepts in the Language of Types

The first example of formal mathematics that we shall consider is type theory itself. Since this is the "primitive language" of Nuprl it will not be necessary to write definitions; the basic constructors and atomic types are already available.

As we have already mentioned, the three atomic types are `int`, `atom` and `void`,[12] and the types are classified into a cumulative hierarchy of universes, U_1, U_2, ...[13] A type in universe U_i is said to be at *level* i. The atomic types all exist at level 1, i.e., in U_1. We can express this for the type of integers with the statement `int in U1`. We now illustrate the mechanics of stating theorems one more time using this theorem as an example. Figure 3.7 shows actual snapshots of the screen as the theorem is produced. First, a position is created in the library using the command `cr t1 thm bot`. The proof editor is entered using `v t1`, and the text editor is entered by positioning

[11] Although `>>` must be present in every theorem, the system does not generate it because we want to allow the possibility of stating theorems with hypotheses, theorems such as `x in int >> x+1 in int`.

[12] This is not a mathematical fact but a fact about Nuprl, so we would not expect to be able to prove in Nuprl a theorem of the form "if A is an atomic type, then A = atom or A = void".

[13] The notation U_i is used in general discussion about the logic of Nuprl, whereas the notation `Ui` is used in discussing text used by Nuprl.

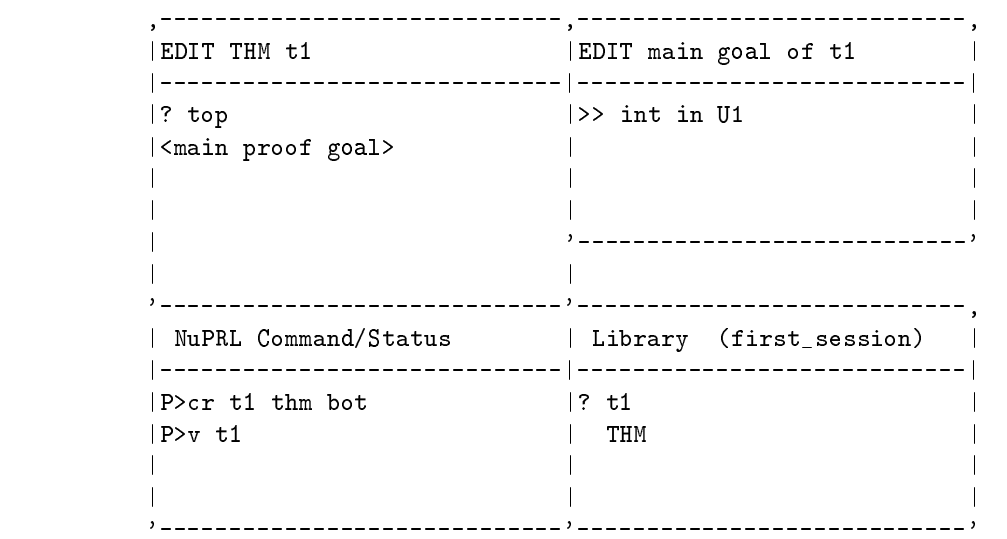

Figure 3.7: Stating a Simple Theorem

the cursor on `<main proof goal>` and hitting SEL. We now type `int in U1` in the text editor and then hit ↑D.

Propositions as Types

The theorems we have seen so far have appeared rather ordinary, but their semantics suggests another kind of statement which is not. The interpretation we discuss below may seem problematic to the reader who is seeing constructive mathematics for the first time, so it may be useful for the new user to contrast the ideas below with the traditional, truth–functional view of the semantics of logic. In constructive logic a proposition is identified with the evidence we can give for it. Specifically, a proposition is a type consisting of formal objects called proofs.[14] For example, the proofs corresponding to `0=0 in int` and to `int in U1` consist solely of the object `axiom`. It is important to note that although we think of statements of membership (e.g., `1 in int`) or of equality (e.g., `0=0 in int`) as being propositions, they are in fact types and *they are combined with other types using type constructors.* Later on we show how to define logical connectives in terms of type constructors so that we can combine these kinds of statements in the familiar fashion of sentential logic. As far as the Nuprl system is concerned, however, propositions are just types.

[14]The proofs we are referring to in this discussion are not formal Nuprl proofs but abstract proofs.

With this view of propositions, we lose some of the equivalences of classical logic. Thus, for example, if p is a proposition and $\sim p$ represents its negation then p and $\sim\sim p$ are no longer equivalent. As types they are different, although from a classical, truth–functional point of view they are the same.

Not only can we assert types which correspond to propositions by exhibiting (either explicitly or implicitly) members of the type, but it makes sense to assert any type. To do so means that the type is inhabited. To prove the assertion is to produce an object, either implicitly or explicitly.[15]

3.6 Defining Logics

In this section we shall show how one can use the type theory described above to define various logics. The general pattern of presentation will be as follows:

- A series of definitions is introduced so that one can work directly with familiar logical notation instead of the underlying type–theoretic notation.
- Theorems are stated using the defined notation.

Constructive Logic

The first example of a logic that we express in type theory is constructive logic. This logic is very close in spirit to the constructive principles underlying type theory, so expressing constructive logic amounts to interpreting the type constructors of the Nuprl logic as logical connectives using the informal semantics of evidence that was employed in the discussion of propositions as types.

The definitions in figure 3.8[16] express the correspondence between logical connectives and type constructors.

We treat $A\ or\ B$ as $A \mid B$, so we do not need an explicit definition for the usual logical "or" in terms of the type constructors. We could also include notations `all3`, `some3`, `all4`, etc., for higher numbers of variables of the same type, but in practice two or three suffice most of the time.

The rules for the logical operators defined in this way are exactly the rules for constructive logic; see for example [Dummet 77]. We will discuss

[15] Every object can be constructed in two ways, either explicitly as in the example above or implicitly. Every proof builds an object implicitly. We will see in chapter 4 how to prove implicitly that the integers are an object in U_1.

[16] It is important to enclose the right side of the definition in parentheses to guarantee proper precedence and scope. Also, it is important not to include a carriage return in the definition. The carriage return will sometimes be read as a bad character when parsing the right side, but it will not be visible on the screen.

this at length in chapter 4 when we study proofs and rules; however, one can see the intentions behind these definitions in terms of an informal semantics of evidence of the kind mentioned in chapter 1. Consider first the definition `and`; there is evidence for A & B precisely when there is a pair, $< a, b >$, such that a is evidence for A and b is evidence for B. Thus treating A & B as $A \# B$ is semantically correct from this viewpoint.

Consider A *or* B next. According to the constructive interpretation, A *or* B is true when we have evidence for A or for B and when we can tell from the evidence which of A or B it supports. This is precisely the information that is available in the type $A \mid B$. An element of the disjoint union is either $inl(a)$ or $inr(b)$, where a is in A and b is in B and inl and inr are the canonical injections into the disjoint sum type.

Next consider $A \Rightarrow B$. According to Brouwer [Brouwer 23], the founder of Intuitionism, an implication is true when there is a method of transforming any evidence for the antecedent, A, into evidence for the consequent, B. If A and B are represented by types, then the function space, $A \rightarrow B$, consists of objects which are in fact evidence for $A \Rightarrow B$. Thus our definition is semantically correct. The definition of the biconditional, $A \Leftrightarrow B$, is correct given that the definition of implication is correct.

The constructive meaning of negation is delicate. To say *not* A is to say that there is no evidence for A or that A will never be proved. In a type–theoretic context one way to know this is to see that there are no members in A. One might be tempted to translate *not* A as $A = Void$, but type equality means more than simply having the same members. To say A is empty, we can say A is equivalent to void, i.e., that $A \rightarrow Void \# Void \rightarrow A$. However, $Void \rightarrow A$ is always inhabited, so the only new content is $A \rightarrow Void$, which we take as our definition of *not*.[17]

Now consider *some* $x{:}A.B$, which is read "we can find an x of type A such that B is true." Intuitively there is evidence for an assertion of existence when we have a witness, a in A, and evidence, b, that B is true with a substituted for free variable x in B. We write this substitution as $B[a/x]$. The dependent sum type $x{:}A \# B$ consists of pairs $< a, b >$ with exactly these properties, so the definition is correct.

A universal statement, *all* $x{:}A.B$, means constructively that given any element a of A we can find evidence for $B[a/x]$. This means that there is a procedure mapping elements a of A into $B[a/x]$. This is precisely what the dependent function space $x{:}A \rightarrow B$ provides, so this definition is also correct.

Finally, there should be no evidence for $false$, so the empty type, $Void$, is the right definition for it.

These definitions can be applied to atomic propositional forms such as the types `x = y in int` or `x < y` to produce statements such as `all`

[17] We will see other definitions later which will take account of the structure of A.

Definition Name	Definition Body
prop	`prop == U1`
false	`false == void`
not	`~<p:prop> == ((<p>) → void)`
and	`<p:prop> & <q:prop> == ((<p>) # (<q>))`
imp	`<p:prop> ⇒ <q:prop> == ((<p>) → (<q>))`
equiv	`<p:prop> ⇔ <q:prop> == (((<p>) → (<q>)) & ((<q>) → (<p>)))`
all	`all <x:var>:<t:type>.<p:prop> == (<x>:(<t>)→(<p>))`
all2	`all <x1:var>,<x2:var>:<t:type>.<p:prop> == (<x1>:(<t>) → <x2>:(<t>) → (<p>))`
some	`some <x:var> : <t:type> . <p:prop> == (<x>:(<t>) # (<p>))`
some2	`some <x1:var>, <x2:var> : <t:type> . <p:prop> == (<x1> : (<t>) # <x2> : (<t>) # (<p>))`

Figure 3.8: Constructive Logic Connectives Expressed as Types

`x:int.some y:int. x<y`, which we read as "for all integers x we can find some integer y such that x is less than y". It is also possible to make more general statements, such as `all A:U1.all x:A.x=x in A`. This statement says that equality on every type is reflexive and involves quantifying over all types in U_1; without the hierarchy of universes it would not be possible to make such a statement predicatively.

The logical operators can be used to state general results about constructive predicate logic. Give the identification of propositions and types, which we can make more conspicuous by the definition `prop == U1`, we can assert theorems about constructive propositional logic. Figure 3.9 gives a list of theorems that were expressed and proved in Nuprl.

Statements in the predicate calculus are usually analyzed, following Frege [Frege 1879], as operators applied to *propositional functions*, where the term propositional function describes a function that maps objects to propositions. Given a type A, the propositional functions (of one argument) on A are of the type $A \rightarrow U_1$. With this reading some basic facts from the predicate calculus are shown in figure 3.10.

Classical Logic

It is possible to express *classical* logic in the Nuprl system by introducing appropriate definitions. More specifically, one can introduce a logical con-

```
>> P:U1->Q:U1->
   P=>Q=>P

>> P:U1->Q:U1->R:U1->
  (P=>Q)=>( P=>Q=>R)=>(P=>R)

>> P:U1->Q:U1->
  (P=>Q=>P&Q)

>> P:U1->Q:U1->
   (P&Q=>P)

>> P:U1->Q:U1->
   (P=>P|Q)

>> P:U1->Q:U1->
   ((P=>Q)=>(P=>~Q)=>~P)

>> P:U1->Q:U1->
   (P&(P=>Q)=>Q)
```

Figure 3.9: Some Theorems of Constructive Propositional Logic

nective to correspond to the classical "or" by defining the new connective via the classical equivalence between "or" and a suitable combination of "not" and "and". Consider the additional logical operators defined in figure 3.11. The "vel" operator represents the classical connective "or".[18] The classical notion of implication is usually called *material implication* and is defined in terms of `vel` just as shown in figure 3.11. Likewise, the classical idea of existence is far different from the constructive notion of existence in that we can prove something exists classically by merely showing that it is contradictory for it not to exist. This is precisely the meaning captured by the given definition.

Figure 3.12 shows some of the laws of classical logic in our notation.

From a formal point of view the situation is clear. We have defined logical operators that obey the laws of classical logic and are thus complete for the classical concept of logical truth in the first–order case. We have given an explanation of these laws in terms of type–theoretic concepts. While this

[18] The usual sign for classical or is $\vee$, which abbreviates the Latin *vel*, meaning or. We have chosen not to use the word "or" at all among the logical operators because it has so many meanings in natural language.

```
>> all A:U1.all P:(A->U1).all Q:(A->U1).
     all x:A.P(x)&Q(x)  <=>
       all x:A.P(x) & all x:A.Q(x)

>> all A:U1.all P:(A->U1).all Q:(A->U1).
     ( (all x:A.P(x)) | (all x:A.Q(x)) ) =>
         all x:A.P(x) | Q(x)

>> all A:U1.all P:(A->U1).all Q:(A->U1).
      some y:A.P(y)&Q(y)  =>
          some y:A.P(y) & some y:A.Q(y)
```

Figure 3.10: Some Predicate Calculus Theorems

Definition Name	Definition Body
vel	`<p:prop> vel <q:prop> == (~(~<p> & ~<q>))`
mat	`<p:prop> imp <q:prop> == (~<p> vel <q>)`
bicond	`<p:prop> equiv <q:prop> == ((<p> imp <q>) & (<q> imp<p>))`
exist	`exists <x:var>:<t:type>.<p:prop> == (~all <x>:<t>.~(<p>))`

Figure 3.11: Definitions for Classical Logic

is not the usual explanation in terms of Tarski's truth semantics, it is an adequate explanation for first–order logic.

Other Logical Operators

A number of other logical concepts can be expressed in Nuprl. For example, the *conditional and* is sometimes used when dealing with partial functions. This connective can be defined as `<p:prop> cand <q:prop> == (x:<p> # <q>)`. This results in an and connective which forces the evaluation of its first argument before the second argument.

As another example, sometimes it is useful to simplify the structure of a type considered as a proposition by "squashing" all of the proofs into one object. This squash operator is defined as

```
|| <p:prop> || == {0 in int | <p> }.
```

```
>>P:U1->Q:U1->
  ((~P)&(P vel Q) imp Q)

>>P:U1->
  (P vel (~P))

>>all A:U1.all P:(A->U1).
  exists x:A.P(x) <=> ~all x:A.~P(x)
```

Figure 3.12: Some Laws of Classical Logic

Thus if <p> is true then || <p> || consists of a single object; all the different proofs have been identified. Note that this operator is built using the subset type constructor.

3.7 Elementary Number Theory

One of the atomic types available in Nuprl is the type integer, called `int`. The availability of integers as an atomic type allows one to express a great deal of elementary number theory without much effort. The type `int` comes equipped with an *induction form* so that one can inductively define functions from integers to integers. In this section we illustrate how some number–theoretic predicates and functions can be defined in Nuprl and how number–theoretic theorems are stated.

We begin this section with an example of a simple theorem which will be proved in chapter 4; we use this theorem to review how one enters theorems in the system. The theorem asserts that for every pair of integers one can find an integer whose value is 0 if the first of the original pair of integers is less than the second and whose value is 1 if the first of the original pair of integers is not less that the second. In this statement there are a pair of quantifiers; the first is universal and the second is existential. Accordingly we need the definitions for universal quantifiers and existential quantifiers from figure 3.8. To state the theorm in Nuprl, one would first create a name and place for the theorem using the commands discussed in section 3.2. Then the theorem would be viewed and finally the text editor would be invoked by clicking R on the mouse in `<main proof goal>`. Once in the text editor the first symbols typed would be >>; after this the text of the theorem can be typed. At this stage the definition `all2` is needed. We use it by first typing ↑I; this will put the cursor in the command window with the `I>` prompt. Now one types `all2` followed by a carriage return. The definition will be instantiated in the text editor. The appearance of the definition

template will be `all <var>,<var>:<type>.<prop>`. The variables `x` and `y` can now be entered as the first two parameters, and `int` can be entered as the third parameter. The body of this universally quantified statement is an existentially quantified statement. The last parameter, `<prop>`, in the definition is instantiated by invoking the definition for `some`. The first two arguments of this quantifier can be filled in as before. The body of the inner quantified statement is a conjunction of the two conditions appearing in the statement of the theorem and can be expressed using the built–in relation `<` between integers and the definitions for conjunction and negation. The final appearance of the theorem is as follows.

```
>>all x,y:int. some z:int.
    (x<y => z=0) & (~x<y => z= 1)
```

This is a theorem whose proof will embody some computational content; the existence proof will involve a procedure which computes z, given x and y.

The following definitions suggest what some standard number theory looks like in Nuprl. We first define the predicate *divides* using the (built–in) mod function.

```
divides(<n>:int,<m>:int) == ( <m> mod <n> = 0 in int)
```

The next definition defines the predicate *prime* on integers.

```
prime(<n:int>) ==
          (1 < <n>) &
          all k:int. ( (0 < k) &
            (<n> mod k = 0  in int) ) =>
              (k=1 in int) | (k=<n> in int)
```

Now these number–theoretic predicates can be used in conjunction with the logic definitions to state theorems in number theory in the following fashion.

```
>> all n:int. prime(n) | ~prime(n)

>> all n:int. some p:int. n<p & prime(p)
```

In stating number–theoretic propositions we come across our first examples of Nuprl theorems that have manifestly interesting computational content. The Nuprl system can extract the procedure from the proof. If we call the theorem `t`, then

```
\n.(term_of(t)(n))
```

is a function which takes a natural number to a prime which is greater than it, where the function `term_of` is the system function that performs function extraction. Similarly the proof of the first theorem above will involve giving a decision procedure for primality.

Another interesting theorem one can state in Nuprl is the remainder theorem:

```
>>all x:int. all y: int. some q:int. some r:int.
 (y = (q*x + r) in int) & (0 <= r < x).
```

We assume that the term `(0 <= r < x)` has been defined to have its usual meaning via a definition.[19] Once again a proof would give rise to an algorithm for finding the quotient and remainder resulting from dividing two integers, and the corresponding function would be extracted by the system. This illustrates, in a very simple setting, the use of Nuprl as a program synthesis tool.

3.8 Set Theory

The Subset Constructor

The expression of concepts from set theory in Nuprl is facilitated by the availability of the set type constructor as one of the primitive constructors of the underlying Nuprl logic. The notation for this constructor is {x:A|B}. Intuitively the set type constructor first forms a dependent sum of the two component types and then discards the second member of each pair in the dependent sum. The members of this type are those members a of the type A such that $B[a/x]$ is inhabited. Recalling the propositions–as–types principle, we may think of B as a propositional function and $B[a/x]$ as a proposition. The proposition $B[a/x]$ is true when the corresponding type is inhabited; thus the notation {x:A|B} defines the type whose members are members a of A that make the proposition $B[a/x]$ true. This is essentially what the classical set formation construct means.

The simplest statements about sets that one can make involve subsets of a specific type such as `int`. In Nuprl we may represent such sets as set types over `int` in the following fashion. Given a predicate P on the integers, the set corresponding to P is,

```
{x:int | P(x)}.
```

This set type denotes those elements of `int` which satisfy P. Two important aspects of this type are that its members are elements of `int` and that the proof of $P(a)$ for an element a of `int` is not available in computations. For example, to prove that 9 belongs to

```
{x:int | some y:int. ~(y=x in int) & x mod y=0 in int}
```

we must prove that some number, such as 3, satisfies the defining predicate, but we do not keep this factor as part of the membership condition. This

[19] If the symbol $<$ is being used in a definition to mean "less than" then it must be preceded by a backslash; otherwise, the system will try to interpret it as the start of a new parameter.

means that from an assumption such as `y in {x:int | P(x)}` we do not have access to the proof of $P(y)$. This aspect of the subset constructor comes out as a restriction on the way hypotheses can be used in the set–elimination rule.

Intervals

Subsets of the integers of the form $n, n+1, \ldots, n+p$ are especially useful. We call them *intervals*, and they are defined as follows. First we define the symbol `<=` to mean "less than or equal to" with the following definition:

```
<x:int>\<=<y:int>==((<y><<x>)->void)}.
```

Now the concept of interval can be defined by

```
{<n:int>,...,<m:int>}=={x:int| <n><=x # x<=<m>}.
```

Using intervals we can define the concept of an *array a of type A* as

```
a:{1,...,n}->A.
```

A very important concept in set theory is the concept of *equal cardinality*. We shall use the term *equipollent* to signify the type–theoretic analogue of equal cardinality. As in classical set theory, two types are equipollent if there are a pair of invertible functions between them. This is captured by the following definition.

```
<A:type> is equipollent with <B:type>==
(some ab:<A>-><B>.some ba:<B>-><A>.all x:<A>.all y:<B>.
ba(ab(x))=x in <A> &  ab(ba(y))=y in <B>)
```

In the constructive account, however, we must exhibit the two functions that establish equipollence, so it is not permissible to use the Schroeder–Bernstein theorem, for example, to establish equipollence. Thus equipollence is not identical to the classical notion of equal cardinality.

The following three Nuprl theorems express basic facts about equipollence.

```
>>all A:U1.  A is equipollent with A

>>all A:U1.all B:U1. A is equipollent with B =>
  B is equipollent with A

>>all A:U1.all B:U1.all C:U1.
      A is equipollent with B &
        B is equipollent with C  =>
           A is equipollent with C
```

Using the notions of intervals and equipollence we can define the concept of a *finite type*. We say that a type A is finite if and only if it is equipollent with some interval.

```
<A:type> is finite== some n:int.{1,...,n} is
equipollent with <A>
```

We say that n is the *cardinality* of A.

A nontrivial statement that can be made with the definitions introduced in this section is the pigeonhole principle. The following Nuprl theorem expresses the principle.

```
>>all n:int.all m:int. all f:{1,...,m}->{1,...,n}.
      0<n<m => some i,j:int.i<j & f(i)=f(j) in int
```

The proof of such a proposition in Nuprl will include a procedure which, given a function f from the interval $\{1,\ldots,m\}$ to the interval $\{1,\ldots,n\}$ with $m > n$, produces the values i and j for which $f(i) = f(j)$. The constructive proof is considerably harder than the (trivial) proof of the classically interpreted pigeon–hole principle, but the information contained in the constructive proof is correspondingly much greater.

3.9 Algebra

In this section we examine a fragment of group theory and discuss how the definition of algebraic *structures* can be cast in the Nuprl logic. Defining an algebraic structure involves identifying certain distinguished functions (the *operators*) and constants and stating the properties of the functions and constants. The properties one wishes to state are usually *equational* and thus correspond to Nuprl terms. Expressing *inequations* is slightly more tedious but does not present any fundamental difficulty.

We begin by discussing how some typical phrases that one might see in an algebra text are expressed in Nuprl. Suppose one wishes to make a statement of the following general form: "...x... = ... *where* x = ...". In the first equation of such a statement the variable x serves as an abbreviation for the right–hand side of the second equation. The entire statement thus expresses whatever the first equation expresses with x appropriately instantiated. A natural way to express instantiation involves the application of a lambda term to another term. Thus we are led to the following Nuprl definition:[20]

```
<t:term> where <x:var>=<tt:term> == (\ <x>.<t>)(<tt>).
```

[20] The space after the "\" in the definition is to prevent the parser from thinking that the following < is being quoted.

In algebra it is customary to write binary operators in infix form. In the Nuprl notation operators are just functions (lambda terms) expressed in curried form. The following definition thus allows one to use traditional algebraic syntax in discussing operators:

```
<x:arg> <op:operation> <y:arg> == <op>(<x>)(<y>).
```

An important property of operators that one might wish to state is the property of *associativity*. Associativity is an equationally stated property; accordingly in Nuprl it is necessary to state associativity with respect to a particular type. One can use the definition for the universal quantifier in writing the definition of associativity to obtain the following definition.

```
<op:A->A->A> associative over <A:type>==
All x,y,z:<A>.
  x <op> (y <op> z) = (x <op> y) <op> z  in <A>
```

During an instantiation of the last definition the template displayed in the library window will look like the following:

```
<A->A->A> associative over <A:type>.
```

With the definitions above, one can state the definition of a group. A group is just a type; accordingly the definition of a group will be a type expression in which the various components of the definition are pieced together. The complete definition is as follows:

```
Group ==
  A:U1
# o:(A->A->A)
# e:A
# All x:A.  x o e = e o x = x  in A
# All x:A. Some y:A.  x o y = y o x = e  in A
# o associative over A
```

In the definition above, `o` is the group composition operator, `e` is the group identity, and the last three lines express the standard properties of a group. There are a number of different ways in which the definition can be used in stating theorems. One could, for example, state the type `Group` as a top–level goal. This would then assert that there exist groups. One could make a statement of the form `B in Group`; this would assert that `B` is a group. A proof of such a statement would of course be constructive and would involve displaying a procedure for computing inverses in `B`. A third way of using the definition of a group would be to state properties true of all groups with statements of the form

```
All G:Group. ... .
```

To be able to state interesting theorems about group theory it is necessary to be able to pull apart the constituents of a group. This is done using the *spread* operator, the elimination form of the product constructor. The following two definitions provide mnemonic access functions.

```
fst(<x:term>) == spread(<x>;u,v.u)

snd(<x:term>) == spread(<x>;u,v.v)
```

Now we can define notation that accesses the definition of a group and produces the underlying carrier set and operations. For example, the following definition defines the carrier set of a group.

```
|<G:Group>| == fst(<G>)
```

We can access the composition operator and the identity of the group by using the following definitions.

```
op_of <G:Group> == fst(snd(<G>))

id_of <G:Group> == fst(snd(snd(<G>)))
```

We conclude this section by stating a typical elementary theorem of group theory, namely that every equation of the form $a \circ x = b$ can be uniquely solved for x. In Nuprl this theorem would read

```
>> All G:Group. All a,b:|G|.
   Some x:|G|.
     (a o x = b in |G|
      & All c:|G|. a o c = b in |G| => c = x in |G|
      where o=op_of G )
```

In the statement above, the fourth line states the uniqueness of the solution.

Chapter 4

Proofs

This chapter contains an informal introduction to Nuprl proofs. It describes the structure of proofs, shows how to use the refinement editor to move around in and add to proofs, and gives enough description and examples of rule usage to enable the reader to prove theorems of his own. Examples are drawn from logic (section 4.3) and elementary number theory (section 4.4). The reader may at times wish to refer to the complete presentation of the rules; this is contained in chapter 8.

4.1 Structure of Proofs

A proof in Nuprl may be thought of as a tree. Associated with each node of the tree is a sequent and, if the node is not a leaf, a rule. A sequent is in turn composed of a numbered list of hypotheses and a goal. The children of a node are uniquely determined by the sequent and rule of that node. If no rule is present, or if the rule is incorrect or not applicable, the node has no children; otherwise, the children correspond to the subgoals generated by the application of the rule to the sequent. Note the departure from conventional usage; a proof need not be complete.

The following example from a Nuprl session was obtained by using the command `view example` to create a window for the theorem object `example` and then using the refinement editor, or *red* for short, to navigate through the proof tree.

```
,-------------------------------------,
|EDIT THM example                     |
|-------------------------------------|
|# top 1 1                            |
|1. x:int                             |
|2. y:int                             |
|>> z:int->(x*(y+z)=x*y+x*z in int)   |
|                                     |
|BY intro                             |
|                                     |
|1# 1. x:int                          |
|   2. y:int                          |
|   3. z:int                          |
|   >> (x*(y+z)=x*y+x*z in int)       |
|                                     |
|2# 1. x:int                          |
|   2. y:int                          |
|   >> int in U1                      |
'-------------------------------------'
```

Displayed in the window above is the sequent to be proven, or *main goal*, associated with the current node, the rule of the node (preceded by `BY`), and the sequents corresponding to the two children of the node. This is the way that the system displays a proof tree in a theorem window; it shows a sequent, the rule applied to it (if any), and the immediate subgoals. The location of the node in the tree is given by the header, `top 1 1` in this case; this means that the node is the first child of the first child of the root of the proof tree. The sign "#" indicates that the *status* of the subproof is *incomplete*. Other possible statuses are: "-" for *bad*, meaning that some descendant (leaf) has an incorrect rule application, or that the main goal was not acceptable; "?" for *raw*, which is the status of the proof before the main goal has been parsed; and "*" for *complete*, which indicates that the subproof has been finished.

One usually visualizes a tree as having its root at the top (or bottom) with the branches going downward (upward). In Nuprl, however, a proof tree should be viewed as lying on its side, with the root at the left and with the first child of a given node being at the same height as its parent, and having its siblings below it. Keeping this in mind will make the commands for moving around in proof trees easier to remember.

4.2 Commands Needed for Proofs

In the preceding chapter we described how to create theorem objects and how to enter the main goal of such an object; in this section we will talk about the

Nuprl commands for viewing and changing proof trees. One builds proofs of theorems using *red*, the refinement editor. *Red* is a syntax–directed and rule–driven editor; the proof is developed by specifying instances of rules, such as `elim` and `arith`, and the editor ensures that the rule instances are appropriate and calculates the new subgoals. The remainder of this section is a tutorial which develops most of the proof of[1]

```
>> All x,y:int.Some z:int.
   (x<y => z=0 in int) & (~x<y => z=1 in int)
```

Starting in the command window, type `create t thm top`; this command creates the theorem object and places it in the library. Now type `ml`. This activates the ML subsystem; Nuprl indicates this by giving the prompt `ML>`. Type "`set_auto_tactic ‘COMPLETE immediate‘;;`" (note that the quotes are back quotes) and then hit RETURN. This has Nuprl show the steps of any proof that it automatically constructs using the *auto–tactic* `immediate`, an ML program which performs simple reasoning automatically. The ML subsystem will be discussed later. Now return to the command mode by typing ↑D and type `view t` to bring up the window:

```
,----------------------------------------,
|EDIT THM t                              |
|----------------------------------------|
|? top                                   |
|<main proof goal>                       |
'----------------------------------------'
```

As described in the preceding chapter, enter the main goal; this will result in a window that looks like the following.

```
,------------------------------------------------------,
|EDIT THM t                                            |
|------------------------------------------------------|
|# top                                                 |
|>> All x,y:int.Some z:int.                            |
|     x<y => (z=0 in int) & ~x<y => (z=1 in int)       |
|                                                      |
|BY <refinement rule>                                  |
'------------------------------------------------------'
```

Note the status; it indicates that the main goal has been successfully parsed and that the proof can proceed. Also note that the statement of the theorem now appears in the library window. Now, the first step of the proof will be to apply an introduction rule. Since a universally quantified formula is, by definition, a dependent function type, the function–introduction rule is required. It suffices, however, to enter `intro` as the rule; the system

[1] The definitions of the logical connectives, which were presented in the preceding chapter, are used here.

will determine from the context exactly which type of introduction is to be applied. The rule is entered into an edit window, which can be brought up in two ways. The first way is to type LONG ↓ in order to move the cursor from the main goal to "`BY <refinement rule>`" and to then type SEL. The other way is to position the mouse pointer over "`BY <refinement rule>`" and then do a mouse SEL, i.e., a left–click on the mouse.[2] The system displays the following windows.

```
,-------------------------------,--------------,
|EDIT THM t                     |EDIT rule of t|
|-------------------------------|--------------|
|# top                          | _            |
|>> All x,y:int.Some z:int.     |              |
|     x<y => (z=0 in int) &     |              |
|     ~x<y => (z=1 in int)      |              |
|                               |              |
|BY <refinement rule>           |              |
'-------------------------------'--------------'
```

Now type the rule name `intro`, finishing with a ↑D. In general, a ↑D causes the entered text to be processed and, unless certain errors have occurred, removes the rule window. If the first word of the text was a rule name, then the text which follows, if any, is checked to make sure that it is appropriate to the rule, and if it is, the rule is applied and the resulting subgoals, if any, are displayed. The proof window is now:

```
,-------------------------------------------------------,
|EDIT THM t                                             |
|-------------------------------------------------------|
|# top                                                  |
|>> All x,y:int.Some z:int.                             |
|     x<y => (z=0 in int) & ~x<y => (z=1 in int)        |
|                                                       |
|BY intro                                               |
|                                                       |
|1# 1. x:(int)                                          |
|   >> (All y:int.Some z:int.                           |
|        x<y => (z=0 in int) & ~x<y => (z=1 in int))     |
|                                                       |
|2* >> (int) in U1                                      |
|                                                       |
'-------------------------------------------------------'
```

Nuprl has generated the two subgoals specified by the function–introduction rule. Now try moving the cursor around using LONG ↑ and LONG ↓, and

[2] In general, any action involving the cursor and SEL key can also can be done using the mouse pointer and a left–click.

note that it stops at the four major entries of the window. If a proof window is not big enough to display all the subgoals, moving the cursor down will force the window to be scrolled. Note that the status of subgoal 2 above is "complete". This indicates that the auto–tactic has completely proven the second subgoal. Now position the cursor over the first subgoal and type $\rightarrow$[3], or position the mouse pointer over it and right–click; either action has the effect of moving the theorem window down the proof tree to the indicated subgoal. Alternatively, one may type LONG $\rightarrow$; this will always move the window to the next (in an order to be described later) unproved leaf of the proof tree. The window should now look like:

```
,-------------------------------------------------------,
|EDIT THM t                                             |
|-------------------------------------------------------|
|# top 1                                                |
|1. x:(int)                                             |
|>> (All y:int.Some z:int.                              |
|     x<y => (z=0 in int) & ~x<y => (z=1 in int))       |
|                                                       |
|BY <refinement rule>                                   |
'-------------------------------------------------------'
```

To strip off another quantifier, proceed as before (i.e., type LONG $\downarrow$ SEL or position the mouse pointer and left–click, then enter the rule `intro` followed by $\uparrow$D). Move to the first of the subgoals by typing LONG $\rightarrow$. The current goal is now:

```
,-------------------------------------------------------,
|EDIT THM t                                             |
|-------------------------------------------------------|
|# top 1 1                                              |
|1. x:(int)                                             |
|2. y:(int)                                             |
|>> (Some z:int.                                        |
|     x<y => (z=0 in int) & ~x<y => (z=1 in int))       |
|                                                       |
|BY <refinement rule>                                   |
'-------------------------------------------------------'
```

Using an introduction rule here would require that a value for `z` be supplied. Since the choice of that value depends on whether or not `x<y`, a case analysis is required. Enter the following as the rule:

```
seq x<y | ~x<y
```

In general, this rule, called the *sequence* rule, introduces a new fact into the proof. The window below shows the two new subgoals; one is to prove

[3] Recall that we think of the proof tree as lying on its side.

the new fact, and the other is to prove the previous goal assuming the new fact.

```
,-------------------------------------------------------,
|EDIT THM t                                             |
|-------------------------------------------------------|
|# top 1 1                                              |
|1. x:(int)                                             |
|2. y:(int)                                             |
|>> (Some z:int.                                        |
|     x<y => (z=0 in int) & ~x<y => (z=1 in int))       |
|                                                       |
|BY seq x<y | ~x<y                                      |
|                                                       |
|1* 1. x:(int)                                          |
|   2. y:(int)                                          |
|   >> x<y | ~x<y                                       |
|                                                       |
|2# 1. x:(int)                                          |
|   2. y:(int)                                          |
|   3. x<y | ~x<y                                       |
|   >> (Some z:int.                                     |
|        x<y => (z=0 in int) & ~x<y => (z=1 in int))     |
'-------------------------------------------------------'
```

Note that the auto–tactic has proven the first subgoal. To see the proof it constructed, move the window to the first subgoal (LONG ↓ LONG ↓ →).

```
,-------------------------------------------------------,
|EDIT THM t                                             |
|-------------------------------------------------------|
|* top 1 1 1                                            |
|1. x:(int)                                             |
|2. y:(int)                                             |
|>> x<y | ~x<y                                          |
|                                                       |
|BY arith  at U1                                        |
|                                                       |
|1* 1. x:(int)                                          |
|   2. y:(int)                                          |
|   >> x in int                                         |
|                                                       |
|2* 1. x:(int)                                          |
|   2. y:(int)                                          |
|   >> y in int                                         |
'-------------------------------------------------------'
```

The auto–tactic applied the `arith` rule to the subgoal and then proved the subgoals (using `intro`). Now go back to the previous node (i.e., the parent of the current one) by typing ←, or by positioning the mouse pointer over the main goal of the window and right–clicking, and proceed to the second subgoal of that node. The window now becomes:

```
,-----------------------------------------------------------,
|EDIT THM t                                                 |
|-----------------------------------------------------------|
|# top 1 1 2                                                |
|1. x:(int)                                                 |
|2. y:(int)                                                 |
|3. x<y | ~x<y                                              |
|>> (Some z:int.                                            |
|      x<y => (z=0 in int) & ~x<y => (z=1 in int))           |
|                                                           |
|BY <refinement rule>                                       |
'-----------------------------------------------------------'
```

A case analysis on hypothesis 3 above is now required; this is done by entering the rule `elim 3`.

```
,-------------------------------------------------------,
|EDIT THM t                                             |
|-------------------------------------------------------|
|# top 1 1 2                                            |
|1. x:(int)                                             |
|2. y:(int)                                             |
|3. x<y | ~x<y                                          |
|>> (Some z:int.                                        |
|     x<y => (z=0 in int) & ~x<y => (z=1 in int))       |
|                                                       |
|BY elim 3                                              |
|                                                       |
|1# 1. x:(int)                                          |
|   2. y:(int)                                          |
|   3. x<y | ~x<y                                       |
|   4. x<y                                              |
|   >> (Some z:int.                                     |
|        x<y => (z=0 in int) & ~x<y => (z=1 in int))     |
|                                                       |
|2# 1. x:(int)                                          |
|   2. y:(int)                                          |
|   3. x<y | ~x<y                                       |
|   4. ~x<y                                             |
|   >> (Some z:int.                                     |
|        x<y => (z=0 in int) & ~x<y => (z=1 in int))     |
'-------------------------------------------------------'
```

Note the pattern of rule usage—to break down the conclusion of the goal we use `intro`, and to break down a hypothesis we use `elim`. We continue by proving the first subgoal. The second, which is very similar, is left to the reader. Type LONG → to get:

```
,-------------------------------------------------------,
|EDIT THM t                                             |
|-------------------------------------------------------|
|# top 1 1 2 1                                          |
|1. x:(int)                                             |
|2. y:(int)                                             |
|3. x<y | ~x<y                                          |
|4. x<y                                                 |
|>> (Some z:int.                                        |
|     x<y => (z=0 in int) & ~x<y => (z=1 in int))       |
|                                                       |
|BY <refinement rule>                                   |
'-------------------------------------------------------'
```

It is now possible to specify a value for `z`, since `x<y` is a hypothesis. Entering "`intro 0`" as the rule causes `z` to be instantiated with 0:

```
,-------------------------------------------------------,
|EDIT THM t                                             |
|-------------------------------------------------------|
|* top 1 1 2 1                                          |
|1. x:(int)                                             |
|2. y:(int)                                             |
|3. x<y | ~x<y                                          |
|4. x<y                                                 |
|>> (Some z:int.                                        |
|     x<y => (z=0 in int) & ~x<y => (z=1 in int))       |
|                                                       |
|BY intro 0                                             |
|                                                       |
|1* 1. x:(int)                                          |
|   2. y:(int)                                          |
|   3. x<y | ~x<y                                       |
|   4. x<y                                              |
|   >> 0 in (int)                                       |
|                                                       |
|2* 1. x:(int)                                          |
|   2. y:(int)                                          |
|   3. x<y | ~x<y                                       |
|   4. x<y                                              |
|   >> ((x<y )->(0=0 in int))#((~x<y ))->(0=1 in int)   |
|                                                       |
|3* 1. x:(int)                                          |
|   2. y:(int)                                          |
|   3. x<y | ~x<y                                       |
|   4. x<y                                              |
|   5. z:(int)                                          |
|   >> (x<y => (z=0 in int) & ~x<y => (z=1 in int)) in U1 |
'-------------------------------------------------------'
```

The auto–tactic now finishes the proof of this subgoal. The rest of the proof is left as an exercise. (Use LONG → to get to the remaining unproved subgoal.)

4.3 Examples from Introductory Logic

Constructive Propositional Logic

Many people have seen formal proofs only during courses in logic. Most such courses begin with rules in which the propositional connectives $\&, |, \sim$ and $\Rightarrow$ correspond to the natural language connectives "and","or","not" and "implies", respectively. If the nature of propositions is left unanalyzed except for their propositional structure, and if the unanalyzed parts are denoted by variables such as P, Q and R, then the resulting forms are called *propositional formulas*. For example, $P\&Q \Rightarrow P$ is an instance of a propositional formula.

Let us take a propositional formula which we recognize to be true and analyze why we believe it. We will then translate the argument into Nuprl. Consider the formula $(P| \sim P) \Rightarrow ((\sim P \Rightarrow \sim Q) \Rightarrow (Q \Rightarrow P))$. We argue for its truth in the following fashion. If we assume $P| \sim P$ then we need to show $(\sim P \Rightarrow \sim Q) \Rightarrow (Q \Rightarrow P)$. Supposing $(\sim P \Rightarrow \sim Q)$ is also true, we need to show $Q \Rightarrow P$; that is, we must show that P is true if Q is. Therefore assume Q is true. Now we know $P| \sim P$, so we can consider two cases; if we establish the truth of P in each case then we have finished the proof. In the first case, where P is true, we are done immediately. If $\sim P$ is true, then because $\sim P \Rightarrow \sim Q$ holds we know that $\sim Q$ is true, for that is the meaning of implication. Therefore, both Q and $\sim Q$ follow from our assumptions. This, of course, is a contradiction, and since from a contradiction we can derive anything, P again is true. This finishes the argument.

To formalize this argument we use the definitions given in chapter 3 of the constructive propositional connectives and of `all`. We also recognize that to say P and Q are arbitrary propositions can be taken to mean that they are arbitrary members of the universe U_1.[4] A reasonable formalization of the assertion to be proved, then, is

```
all P:U1. all Q:U1. (P|~P) => (~P=>~Q) => (Q=>P).
```

Following are a few snapshots from a session in which this theorem is proved. Note the similarity between the steps of the formal proof and the steps of the informal proof given above. In the first two windows below, we see the use of the `intro` rule as the formal analogue of the "assume" or "suppose" of the informal argument.

[4] It might seem natural to be more general and say that P and Q belong to any U_i whatsoever. This is something that we will be able to do in Nuprl shortly, but we do not want to discuss such matters at this point.

```
,-------------------------------------,
|EDIT THM prop1                       |
|-------------------------------------|
|# top 1                              |
|1. P:(U1)                            |
|>> (all Q:U1.                        |
|   (P|~P)=>(~P=>~Q)=>(Q=>P))         |
|                                     |
|BY intro at U2                       |
|                                     |
|1# 1. P:(U1)                         |
|   2. Q:(U1)                         |
|   >> ( (P|~P)=>(~P=>~Q)=>(Q=>P) )   |
|                                     |
|2* 1. P:(U1)                         |
|   >> (U1) in U2                     |
'-------------------------------------'
,-------------------------------------,
|EDIT THM prop1                       |
|-------------------------------------|
|# top 1 1 1                          |
|1. P:(U1)                            |
|2. Q:(U1)                            |
|3. (P|~P)                            |
|>> (~P=>~Q)=>(Q=>P)                  |
|                                     |
|BY intro  at U1                      |
|                                     |
|1# 1. P:(U1)                         |
|   2. Q:(U1)                         |
|   3. (P|~P)                         |
|   4. (~P=>~Q)                       |
|   >> (Q=>P)                         |
|                                     |
|2* 1. P:(U1)                         |
|   2. Q:(U1)                         |
|   3. (P|~P)                         |
|   >> (~P=>~Q) in U1                 |
'-------------------------------------'
```

In the main goal of the next window is the result of five consecutive `intros`. The step shown below uses `elim` to do the case analysis of the informal argument.

```
,-----------------------------------------------,
|EDIT THM prop1                                 |
|-----------------------------------------------|
|# top 1 1 1 1 1                                |
|1. P:(U1)                                      |
|2. Q:(U1)                                      |
|3. (P|~P)                                      |
|4. (~P=>~Q)                                    |
|5. Q                                           |
|>> P                                           |
|                                               |
|BY elim 3                                      |
|                                               |
|1* 1. P:(U1)                                   |
|   2. Q:(U1)                                   |
|   3. (P|~P)                                   |
|   4. (~P=>~Q)                                 |
|   5. Q                                        |
|   6. P                                        |
|   >> P                                        |
|                                               |
|2# 1. P:(U1)                                   |
|   2. Q:(U1)                                   |
|   3. (P|~P)                                   |
|   4. (~P=>~Q)                                 |
|   5. Q                                        |
|   6. ~P                                       |
|   >> P                                        |
'-----------------------------------------------'
```

In the main goal of the next window we see that the conclusion follows from hypotheses 4, 5 and 6. To use hypothesis 4 we `elim` it.

```
,---------------------------,--------------------------,
|EDIT THM prop1             |EDIT rule of prop1        |
|---------------------------|--------------------------|
|# top 1 1 1 1 1 2          |elim 4                    |
|1. P:(U1)                  |                          |
|2. Q:(U1)                  |                          |
|3. (P|~P)                  |                          |
|4. (~P=>~Q)                |                          |
|5. Q                       |                          |
|6. ~P                      |                          |
|>> P                       '--------------------------'
|                                          |
|BY <refinement rule>                      |
'------------------------------------------'
```

Nuprl will then require us to prove `~P`, which is trivial since it is a hypothesis, and then to prove `P` given the new hypothesis `~Q`, which is also trivial, since we will have two contradictory hypotheses. The auto–tactic will handle both of these subgoals. This completes the proof.

The preceding proof gave a few examples of the application of `intro` and `elim` rules to propositional formulas. A more complete account is given in figure 4.1, which shows a set of rules for the constructive propositional calculus in a format suitable to a refinement logic, with the premises (subgoals) *below* the conclusion (goal). If a goal has the form of one of the goals in the table, then specifying `intro` or `elim` as the Nuprl rule will generate the subgoals shown with the exception that `=> intro` will have a second subgoal, `P in U1`, which will usually be proved by the auto–tactic. The hypotheses shown in the goals of the elim rules will in general be surrounded by other hypotheses, and in all of the rules the entire hypothesis list is copied to the subgoals with any new hypotheses added to the end. Recall that `~P` is really `P => false`, so rules dealing with negation can be derived from the given rules.

Classical Propositional Logic

As we saw in chapter 3, the operators associated with classical logic are easily defined in Nuprl. Now we will see how to prove a typical fact about them. The most characteristic fact about the classical operators is the "law of the excluded middle," which says that for every proposition P, $P\ vel\ {\sim}P$, where *vel* is usually read as the English "or". Recall that in Nuprl, `P vel ~P` is defined to mean `~(~P & ~~P)`. The following snapshots show how this proposition is proved. It is interesting to notice that there is not much computational content to this result; thus, while it is a true fact in Nuprl, it is not a very interesting one.

Connective	Introduction	Elimination
&	$\frac{\vdash P\&Q}{\vdash P,\ \vdash Q}$	$\frac{P\&Q \vdash R}{P, Q \vdash R}$
$\vee$	$\frac{\vdash P \vee Q}{\vdash P}$	$\frac{P \vee Q \vdash R}{P \vdash R,\ Q \vdash R}$
	$\frac{\vdash P \vee Q}{\vdash Q}$	
$\Rightarrow$	$\frac{\vdash P \Rightarrow Q}{P \vdash Q}$	$\frac{P \Rightarrow Q \vdash R}{\vdash P,\ Q \vdash R}$
false	—	$\mathit{false} \vdash P$

Figure 4.1: Refinement Rules for the Constructive Propositional Logic

The next window shows the hypotheses and subgoals after introducing the arbitrary propositional variable.

```
,-----------------------------------------,
|EDIT THM t2                              |
|-----------------------------------------|
|# top                                    |
|>> all P:U1.  P vel ~P                   |
|                                         |
|BY intro at U2                           |
|                                         |
|1# 1. P:(U1)                             |
|    >> (P vel ~P)                        |
|                                         |
|2* >> (U1) in U2                         |
'-----------------------------------------'
```

Next we deal with the first subgoal. (The system has dispatched the second one.) Since `P vel ~P` is defined as a negation, and since negation is defined to be an implication, we prove it via an introduction rule. This results in the two subgoals shown below.

```
,--------------------------------------,
|EDIT THM t2                           |
|--------------------------------------|
|# top 1                               |
|1. P:(U1)                             |
|>> (P vel ~P)                         |
|                                      |
|BY intro  at U1                       |
|                                      |
|1# 1. P:(U1)                          |
|   2. ~P&~~P                          |
|   >> void                            |
|                                      |
|2* 1. P:(U1)                          |
|   >> ~P&~~P in U1                    |
'--------------------------------------'
```

The system has proven the second subgoal here. We handle the first subgoal by first breaking down the conjunction in the hypothesis list.

```
,----------------------------------------,
|EDIT THM t2                             |
|----------------------------------------|
|# top 1 1                               |
|1. P:(U1)                               |
|2. ~P&~~P                               |
|>> void                                 |
|                                        |
|BY elim 2                               |
|                                        |
|1# 1. P:(U1)                            |
|   2. ~P&~~P                            |
|   3. ~P                                |
|   4. ~~P                               |
|   >> void                              |
'----------------------------------------'
```

Now we recognize that `~~P` is just `~P=>void`, so if we eliminate hypothesis 4 the proof will be complete. The result of this command is displayed in the next snapshot.

```
,-----------------------------------------,
|EDIT THM t2                              |
|-----------------------------------------|
|* top 1 1 1                              |
|1. P:(U1)                                |
|2. ~P&~~P                                |
|3. ~P                                    |
|4. ~~P                                   |
|>> void                                  |
|                                         |
|BY elim 4                                |
|                                         |
|1* 1. P:(U1)                             |
|   2. ~P&~~P                             |
|   3. ~P                                 |
|   4. ~~P                                |
|   >> ~P                                 |
|                                         |
|2* 1. P:(U1)                             |
|   2. ~P&~~P                             |
|   3. ~P                                 |
|   4. ~~P                                |
|   5. void                               |
|   >> void                               |
'-----------------------------------------'
```

Predicate Logic

Beyond propositional calculus lies an analysis of phrases such as "for all", "for every", "there exists" and "there is". As we saw in chapter 3, in Nuprl we treat these phrases as operations on propositional functions. Thus the sentence "for every integer y there is an integer z such that z is less than y" is analyzed first as the operator `all y:int` applied to the propositional function "there is an integer z such that z is less than `y`". This in turn is analyzed as the operator `some z:int` applied to the propositional form `z<y`. Nuprl provides two propositional forms on the integers: `x=y in int` and `x<y`.

Once we have seen how to express arbitrary propositions, we can state all the formulas of the predicate calculus over an arbitrary domain or over an arbitrary nonempty domain as is common in classical logic textbooks. Chapter 3 gave several examples of these statements; we now show how to prove one of them. The snapshots below should be almost self–explanatory, so little further comment will be given.

```
,----------------------,-------------------------------------,
|EDIT THM t3           |EDIT main goal of t3                 |
|----------------------|-------------------------------------|
|? top                 |>>all A:U1.all P:A->A->U1.           |
|<main proof goal>     |      some y:A.all x:A.P(x)(y) =>    |
|                      |      all x:A.some y:A.P(x)(y)       |
|                      '-------------------------------------'
|                                                            |
'------------------------------------------------------------'

,----------------------------------------------------,
|EDIT THM t3                                         |
|----------------------------------------------------|
|# top 1 1                                           |
|1. A:(U1)                                           |
|2. P:(A->A->U1)                                     |
|>> some y:A.all x:A.P(x)(y) => all x:A.some y:A.P(x)(y)|
|                                                    |
|BY intro  at U1                                     |
|                                                    |
|1# 1. A:(U1)                                        |
|   2. P:(A->A->U1)                                  |
|   3. some y:A.all x:A.P(x)(y)                      |
|   >> all x:A.some y:A.P(x)(y)                      |
|                                                    |
|2# 1. A:(U1)                                        |
|   2. P:(A->A->U1)                                  |
|   >> some y:A.all x:A.P(x)(y) in U1                |
'----------------------------------------------------'

,-------------------------------------,-----------------,
|EDIT THM t3                          |EDIT rule of t3  |
|-------------------------------------|-----------------|
|# top 1 1 1 1                        |elim 3           |
|1. A:(U1)                            |                 |
|2. P:(A->A->U1)                      |                 |
|3. some y:A.all x:A.P(x)(y)          |                 |
|4. x:(A)                             |                 |
|>> (some y:A.P(x)(y))                |                 |
|                                     |                 |
|BY <refinement rule>                 '-----------------|
|                                                       |
'-------------------------------------------------------'
```

Since `some` $x{:}T.P$ is defined to be a dependent product, an inhabitant of the third hypothesis above is a pair. The next elimination step thus

requires two new variables, y0 and h in this case, which will each contain a component of the pair.

```
,-----------------------------------------------,
|EDIT THM t3                                    |
|-----------------------------------------------|
|# top 1 1 1 1                                  |
|1. A:(U1)                                      |
|2. P:(A->A->U1)                                |
|3. some y:A.all x:A.P(x)(y)                    |
|4. x:(A)                                       |
|>> (some y:A.P(x)(y))                          |
|                                               |
|BY elim 3 new y0, h                            |
|                                               |
|1# 1. A:(U1)                                   |
|   2. P:(A->A->U1)                             |
|   3. some y:A.all x:A.P(x)(y)                 |
|   4. x:(A)                                    |
|   5. y0:(A)                                   |
|   6. h:(all x:A.P(x)(y0))                     |
|   >> (some y:A.P(x)(y))                       |
'-----------------------------------------------'
```

```
,-----------------------------------------------,
|EDIT THM t3                                    |
|-----------------------------------------------|
|# top 1 1 1 1 1                                |
|1. A:(U1)                                      |
|2. P:(A->A->U1)                                |
|3. some y:A.all x:A.P(x)(y)                    |
|4. x:(A)                                       |
|5. y0:(A)                                      |
|6. h:(all x:A.P(x)(y0))                        |
|>> (some y:A.P(x)(y))                          |
|                                               |
|BY intro y0                                    |
|                                               |
|1* 1. A:(U1)                                   |
|   2. P:(A->A->U1)                             |
|   3. some y:A.all x:A.P(x)(y)                 |
|   4. x:(A)                                    |
|   5. y0:(A)                                   |
|   6. h:(all x:A.P(x)(y0))                     |
|   >> y0 in (A)                                |
|                                               |
|2# 1. A:(U1)                                   |
|   2. P:(A->A->U1)                             |
|   3. some y:A.all x:A.P(x)(y)                 |
|   4. x:(A)                                    |
|   5. y0:(A)                                   |
|   6. h:(all x:A.P(x)(y0))                     |
|   >> (P(x)(y0))                               |
'-----------------------------------------------'
```

```
,-------------------------------------------------,
|EDIT THM t3                                      |
|-------------------------------------------------|
|* top 1 1 1 1 1 2                                |
|1. A:(U1)                                        |
|2. P:(A->A->U1)                                  |
|3. some y:A.all x:A.P(x)(y)                      |
|4. x:(A)                                         |
|5. y0:(A)                                        |
|6. h:(all x:A.P(x)(y0))                          |
|>> (P(x)(y0))                                    |
|                                                 |
|BY elim h on x                                   |
|                                                 |
|1* 1. A:(U1)                                     |
|   2. P:(A->A->U1)                               |
|   3. some y:A.all x:A.P(x)(y)                   |
|   4. x:(A)                                      |
|   5. y0:(A)                                     |
|   6. h:(all x:A.P(x)(y0))                       |
|   >> x in (A)                                   |
|                                                 |
|2* 1. A:(U1)                                     |
|   2. P:(A->A->U1)                               |
|   3. some y:A.all x:A.P(x)(y)                   |
|   4. x:(A)                                      |
|   5. y0:(A)                                     |
|   6. h:(all x:A.P(x)(y0))                       |
|   7. (P(x)(y0))                                 |
|   >> (P(x)(y0))                                 |
'-------------------------------------------------'
```

4.4 Example from Elementary Number Theory

In chapter 3 simple results about integers were stated; now a simple number–theoretic argument will be developed. Our goal is to prove that every nonnegative integer x has an integer square root, which we define to be that integer y such that $y^2 \leq x < (y+1)^2$. From the proof we will derive an integer square root function called `sqrt`.

The following Nuprl statement poses the computational problem that we want to solve.

```
>> all x:int. x>=0 => some y:int. y*y <= x < (y+1)*(y+1)
```

A proof of this statement in Nuprl must be constructive because the constructive connectives and quantifiers defined in chapter 3 are used to state the problem; hence the proof will implicitly contain a method for computing square roots. We will use the proof to define the square root function. Before looking at the formal proof, however, let us consider an informal constructive proof.

We clearly want to proceed by induction, because one can build the root of x from a root $y0$ of $x-1$ by testing whether $x < (y0+1)^2$. If this inequality holds then $y0$ is also the root of x; if it does not then $(y0+1)$ is the root of x. Therefore we introduce x and then perform induction on x (the Nuprl rule for this being `elim x`). Integer induction generally involves three proof obligations: one for downward induction, to establish the result for negative numbers, one for the base case, $x = 0$, and one for the upward case of the positive integers. For this proof the downward case is trivial because x is assumed to be nonnegative. The base case is also trivial because for $x = 0$ we must take y to be 0. We are left with the upward induction case, for which the induction assumption is

$$x - 1 \geq 0 \Rightarrow \textit{some } y{:}\textit{int}.\ y^2 \leq x - 1 < (y+1)^2.$$

We know $0 < x$ because we are in the upward case, so we can prove $x-1 \geq 0$. That allows us to conclude that

$$\textit{some } y{:}\textit{int}.\ y^2 \leq x - 1 < (y+1)^2.$$

We want to examine this y explicitly so that we can compare y^2 to x, so we choose $y0$ as a witness for the *some* quantifier. (This will be done in Nuprl by using `elim`.)

Once we have $y0$ we can do a case analysis on whether or not $x < (y0+1)^2$. We would like to invoke the "trichotomy" rule,

$$x > (y0+1)^2 | x = (y0+1)^2 | x < (y0+1)^2,$$

and then observe that the first case is impossible so that we only need to consider

$$x = (y0+1)^2 | x < (y0+1)^2.$$

In the first case `y0` is the root and in the last case `y0+1` is.

In the snapshots which follow the preceding informal argument is formalized in Nuprl. The first snapshot shows the result of the first introduction rule applied to the statement of the theorem.

```
,-------------------------------------------------,
|EDIT THM root                                    |
|-------------------------------------------------|
|# top                                            |
|>> all x:int.  x>=0 => some y:int. y*y<= x <(y+1)*  |
|   (y+1)                                         |
|                                                 |
|BY intro                                         |
|                                                 |
|1# 1. x:(int)                                    |
|   >> ( x>=0 => some y:int. y*y<= x <(y+1)*(y+1) )  |
|                                                 |
|2* >> (int) in U1                                |
`-------------------------------------------------'
```

The `new h` part of the rule shown in the next snapshot directs the refinement editor to label the new hypothesis with `h`. Recall that when `x` is an integer `elim x` is the refinement rule for induction.

```
,-------------------------------------------------,
|EDIT THM root                                    |
|-------------------------------------------------|
|# top 1                                          |
|1. x:(int)                                       |
|>> ( x>=0 => some y:int. y*y<= x <(y+1)*(y+1) )     |
|                                                 |
|BY elim x new h                                  |
|                                                 |
|1# 1. x:(int)                                    |
|   2. x<0                                        |
|   3. h:(  x+1>=0 => some y:int. y*y<= x+1 <(y+1)*  |
|      (y+1))                                     |
|   >> ( x>=0 => some y:int. y*y<= x <(y+1)*(y+1) )  |
|                                                 |
|2# 1. x:(int)                                    |
|   >> ( 0>=0 => some y:int. y*y<= 0 <(y+1)*(y+1) )  |
|                                                 |
|3# 1. x:(int)                                    |
|   2. 0<x                                        |
|   3. h:(  x-1>=0 => some y:int. y*y<= x-1 <(y+1)*  |
|      (y+1))                                     |
|   >> ( x>=0 => some y:int. y*y<= x <(y+1)*(y+1) )  |
`-------------------------------------------------'
```

We now introduce the fact that `x-1>=0` in order to use the induction hypothesis.

```
,--------------------------------------------------,
|EDIT THM root                                     |
|--------------------------------------------------|
|# top 1 3 1                                       |
|1. x:(int)                                        |
|2. 0<x                                            |
|3. h:( x-1>=0 => some y:int. y*y<= x-1 <(y+1)*(y+1))|
|4. (x>=0)                                         |
|>> (some y:int. y*y<= x <(y+1)*(y+1))             |
|                                                  |
|BY seq x-1>=0                                     |
|                                                  |
|1* 1. x:(int)                                     |
|   2. 0<x                                         |
|   3. h:(  x-1>=0 => some y:int. y*y<= x-1 <(y+1)*  |
|      (y+1))                                      |
|   4. (x>=0)                                      |
|   >> x-1>=0                                      |
|                                                  |
|2# 1. x:(int)                                     |
|   2. 0<x                                         |
|   3. h:(  x-1>=0 => some y:int. y*y<= x-1 <(y+1)*  |
|      (y+1))                                      |
|   4. (x>=0)                                      |
|   5. x-1>=0                                      |
|   >> (some y:int. y*y<= x <(y+1)*(y+1))          |
`--------------------------------------------------'
```

In the next step we want to obtain the consequent of hypothesis 3, the induction hypothesis. Since we know `x-1>=0` the first subgoal will be immediate. The interesting part is subgoal 2, in which the consequent is now available. (From now on we will elide some of the hypotheses by substituting "..." for them.[5])

[5] Soon the system will allow the user to suppress the display of any hypotheses he is not interested in.

```
,--------------------------------------------------,
|EDIT THM root                                     |
|--------------------------------------------------|
|# top 1 3 1 2                                     |
|1. x:(int)                                        |
|2. 0<x                                            |
|3. h:( x-1>=0 => some y:int. y*y<= x-1 <(y+1)*(y+1))|
|4. (x>=0)                                         |
|5. x-1>=0                                         |
|>> (some y:int. y*y<= x <(y+1)*(y+1))             |
|                                                  |
|BY elim h new r                                   |
|                                                  |
|1* 1...5.                                         |
|   >> (x-1>=0)                                    |
|                                                  |
|2# 1...5.                                         |
|   6. r:(some y:int. y*y<= x-1 <(y+1)*(y+1))      |
|   >> (some y:int. y*y<= x <(y+1)*(y+1))          |
`--------------------------------------------------'
```

Now we choose a name, y0, for the witness of the existential quantifier in line 6. As a result we are also required to supply a name for the fact known about y0, namely hh.

```
,--------------------------------------------------,
|EDIT THM root                                     |
|--------------------------------------------------|
|* top 1 3 1 2 2                                   |
|1. x:(int)                                        |
|2. 0<x                                            |
|3. h:( x-1>=0 => some y:int. y*y<= x-1 <(y+1)*(y+1))|
|4. (x>=0)                                         |
|5. x-1>=0                                         |
|6. r:(some y:int. y*y<= x-1 <(y+1)*(y+1))         |
|>> (some y:int. y*y<= x <(y+1)*(y+1))             |
|                                                  |
|BY elim r new y0,hh                               |
|                                                  |
|1# 1...6.                                         |
|   7. y0:(int)                                    |
|   8. hh:( y0*y0<= x-1 <(y0+1)*(y0+1))            |
|   9. r=<y0,hh> in (some y:int. y*y<= x-1 <(y+1)*(y+|
|      1))                                         |
|   >> (some y:int. y*y<= x <(y+1)*(y+1))          |
`--------------------------------------------------'
```

Now the sequence rule is used to introduce the formula needed for the case analysis.

```
,-------------------------------------------------,
|EDIT THM root                                    |
|-------------------------------------------------|
|# top 1 3 1 2 2 1                                |
|1...7.                                           |
|8. hh:( y0*y0<= x-1 <(y0+1)*(y0+1))              |
|9. r=<y0,hh> in (some y:int. y*y<= x-1 <(y+1)*(y+1))|
|>> (some y:int. y*y<= x <(y+1)*(y+1))            |
|                                                 |
|BY seq (y0+1)*(y0+1)<x | ~(y0+1)*(y0+1)<x         |
|                                                 |
|1* 1...7.                                        |
|   8. hh:( y0*y0<= x-1 <(y0+1)*(y0+1))           |
|   9. r=<y0,hh> in (some y:int. y*y<= x-1 <(y+1)*(y+|
|      1))                                        |
|   >> (y0+1)*(y0+1)<x | ~(y0+1)*(y0+1)<x         |
`-------------------------------------------------'
```

The contradictory case follows.

```
,-------------------------------------------------,
|EDIT THM root                                    |
|-------------------------------------------------|
|* top 1 3 1 2 2 1 2 1 1                          |
|1...9.                                           |
|10. (y0+1)*(y0+1)<x | ~(y0+1)*(y0+1)<x            |
|11. (y0+1)*(y0+1)<x                              |
|12. (y0*y0<= x-1 )                               |
|13. ( x-1 <(y0+1)*(y0+1))                        |
|>> (some y:int. y*y<= x <(y+1)*(y+1))            |
|                                                 |
|BY arith  at U1                                  |
`-------------------------------------------------'
```

Now comes the other part of the trichotomy analysis.

```
,--------------------------------------------------,
|EDIT THM root                                     |
|--------------------------------------------------|
|# top 1 3 1 2 2 1 2 2                             |
|1...9.                                            |
|10. (y0+1)*(y0+1)<x | ~(y0+1)*(y0+1)<x             |
|11. ~(y0+1)*(y0+1)<x                              |
|>> (some y:int. y*y<= x <(y+1)*(y+1))             |
|                                                  |
|BY seq ( (y0+1)*(y0+1)=x in int ) | x<(y0+1)*(y0+1) |
`--------------------------------------------------'
```

Now we conduct a case analysis. Only a few steps of each case are shown.

```
,--------------------------------------------------,
|EDIT THM root                                     |
|--------------------------------------------------|
|# top 1 3 1 2 2 1 2 2 2 1                         |
|13. ( (y0+1)*(y0+1)=x in int )                    |
|>> (some y:int. y*y<= x <(y+1)*(y+1))             |
|                                                  |
|BY intro (y0+1)                                   |
|                                                  |
|1* 1...10.                                        |
|   11. ~(y0+1)*(y0+1)<x                           |
|   12. ( (y0+1)*(y0+1)=x in int ) | x<(y0+1)*(y0+1) |
|   13. ( (y0+1)*(y0+1)=x in int )                 |
|   >> (y0+1) in (int)                             |
`--------------------------------------------------'
```

```
,--------------------------------------------------,
|EDIT THM root                                     |
|--------------------------------------------------|
|* top 1 3 1 2 2 1 2 2 2 1 2 1                     |
|1...10.                                           |
|11. ~(y0+1)*(y0+1)<x                              |
|12. ( (y0+1)*(y0+1)=x in int ) | x<(y0+1)*(y0+1)  |
|13. ( (y0+1)*(y0+1)=x in int )                    |
|>> ((y0+1)*(y0+1)<= x )                           |
|                                                  |
|BY arith  at U1                                   |
`--------------------------------------------------'
```

Now we consider the other case, given by hypothesis 13 above.

```
,-------------------------------------------------,
|EDIT THM root                                    |
|-------------------------------------------------|
|* top 1 3 1 2 2 1 2 2 2                          |
|   1...9.                                        |
|   10. (y0+1)*(y0+1)<x | ~(y0+1)*(y0+1)<x         |
|   11. ~(y0+1)*(y0+1)<x                          |
|   12. ( (y0+1)*(y0+1)=x in int ) | x<(y0+1)*(y0+1) |
|   13. x<(y0+1)*(y0+1)                           |
|   >> (some y:int. y*y<= x <(y+1)*(y+1))         |
|                                                 |
|BY intro y0                                      |
`-------------------------------------------------'
,-------------------------------------------------,
|EDIT THM root                                    |
|-------------------------------------------------|
|* top 1 3 1 2 2 1 2 2 2 2 2                      |
|1...11.                                          |
|12. ( (y0+1)*(y0+1)=x in int ) | x<(y0+1)*(y0+1)  |
|13. x<(y0+1)*(y0+1)                              |
|>> ( y0*y0<= x <(y0+1)*(y0+1))                   |
|                                                 |
|BY intro                                         |
`-------------------------------------------------'
```

Assuming that the theorem has been proved and is named `root`, the term `term_of(root)` will evaluate[6] to a function which takes an integer `x` and a proof of `x>=0` and produces a proof of `some y:int.(...)`. A proof of the latter is a pair, the first element of which is the integer square root of `x`. Thus the definition below of `rootf`, when instantiated with a nonnegative integer as its parameter, will evaluate to the integer square root of the supplied integer. The definition uses a defined function `1of` which returns the first element of a pair. Also, we supply `\v.v` as the proof of `x>=0`; since by definition `x>=0` is `x<0=>void`, a proof of it, if it is true, can be any lambda term.

```
rootf(<x:int>) == 1of( term_of(root)(<x>)(\v.v) )
```

This is a *partial function* on `int` because its domain is a subset of `int`. It is a total function from nonnegative integers to integers.

If we want a function which will return an integer value on all integer inputs, then the following theorem describes a reasonable choice.

```
>> all x:int. some y:int.
   (x<0 => y=0 in int) & (x>=0 => y*y<=x<(y+1)*(y+1))
```

[6] Evaluation in Nuprl will be discussed in the next chapter.

If this theorem is called `troot` then a (total) square root function on `int` can be defined by

```
sqrt(<x:int>)== 1of(term_of(troot)(x)).
```

Chapter 5

Computation

We may view Nuprl as a programming language in which the proofs are programs which are "translated" into terms and "run" via an evaluator. Proving the truth of a statement in the system is equivalent to showing that the type corresponding to the statement is inhabited, and proving that a type is inhabited in a constructive setting requires that the user specify how an object of the type be built. Implicitly associated with each Nuprl proof, then, is a term whose type is specified by the main assertion being proved. This term exhibits the properties specified by the assertion it corresponds to; if we think of a programming problem as being a list of specifications, then a proof that the specifications can be met defines an algorithm which solves the problem, and the associated term becomes the computational realization of the algorithm. The system also supplies a means for evaluating these terms. Given a term the evaluator attempts to find a value corresponding to the term. For more on the correspondence between proofs and programs see [Bates & Constable 85] and [Sasaki 85].

In this chapter we consider Nuprl as a programming tool. We describe generally the way in which the system extracts terms from proofs and computes the values of terms, and we conclude the chapter with an example which demonstrates the workings of the extractor and the evaluator.

5.1 Term Extraction

A proof of a theorem in the Nuprl system implicitly provides directions for constructing a witness for the truth of the theorem. These directions arise from the fact that every Nuprl rule has associated with it an *extraction form*, a term template which yields a term when various parameters to the form are instantiated with terms. These forms may give rise either to canonical or noncanonical terms depending on the nature of the rule corresponding

to the form. Figure 5.1 lists the noncanonical forms, while the canonical forms and the extraction form corresponding to each proof rule are listed in chapter 8.

A proof gives rise to a term in that each rule used in the proof produces an extract form whose parameters are instantiated with the terms inductively extracted from the subgoals generated by the rule invocation. For example, if we wish to prove a theorem of the form $A|B$ using the `intro` rule for disjoint union we must prove either A or B as a subgoal. Assuming we prove B and assuming b is the term inductively extracted from the proof of B then `inr(`b`)` becomes the term extracted from the proof of $A|B$, for `inr(...)` is the extraction form for the right `intro` rule for disjoint unions. Note that if b is in type B then `inr(`b`)` is in $A|B$, so the extraction makes sense. Similarly, if we prove something of the form `x:`$A|B$ `>>` T using the `elim` rule for disjoint union on the hypothesis, then Nuprl generates two subgoals. The first requires us to show that if A is true (i.e., y:A appears in the hypothesis list) then T is true. The second requires us to show that if B is true (i.e., z:B appears in the hypothesis list) then T is true. If $y.ta$ is the term extracted from the the first subgoal ($y.ta$ is a term with y a free variable whose type is A; recall that y represents our assumption of the truth of A in the first subgoal) and $z.tb$ is the term extracted from the second subgoal, then `decide(`x; $y.ta$; $z.tb$`)` is the term extracted from the proof of `x:`$A|B$ `>>` T. Note that if x corresponds to `inl(`a`)` for some a in A then from the computation rules the extracted term is equivalent to $ta[a/y]$; since $y.ta$ proves T under the assumption of A and A holds (since a is in A) the extracted term works as desired. Similarly, the term behaves properly if x has value `inr(`b`)` for some b in B; thus this extraction form is justified.

One should note that certain standard programming constructs have analogs as Nuprl terms. In particular, recursive definition corresponds to the `ind(...)` form, which is extracted from proofs which use induction via the `elim` rule on integers. `decide(...)` represents a kind of if–then–else construct, while the functional terms extracted from proofs using functional `intro` correspond to function constructs in a standard programming language.

To display the term corresponding to a theorem t one evaluates a special Nuprl term, `term_of(`t`)`, which constructs the extracted term of t in a recursive fashion. Briefly, the extraction form of the top refinement rule becomes the outermost form of the term being constructed. The parameters of the form then become the terms constructed from the subgoals generated by the refinement rule.

Many Nuprl rules require the user to prove that a type is well–formed, i.e., that the type resides in some universe U_i. These subgoals, when proved, yield the extraction term `axiom` and are usually ignored by `term_of` as it builds the term for a theorem.

We should note here that one can manipulate canonical and noncanonical

terms explicitly in the system. For example, the following definition defines an absolute value function.

```
abs(<x:int>) == less (<x>; 0; -<x>; <x>)
```

We may now use `abs` as a definition in the statement and proof of theorems. This capability adds a great deal of flexibility to the system.

5.2 Evaluation

The Nuprl terms define a simple functional programming language whose reduction rules are given by the computation rules of the Nuprl theory. By definition canonical terms have outermost forms which cannot be reduced and, as such, represent the values of the theory. On the other hand the outermost forms of noncanonical terms can be reduced. The Nuprl *evaluator* gives the user the means to compute the values of terms. Given a closed term[1] t the evaluator attempts to find a canonical term k such that t and k denote the same value. The form of the term guides this search process. Briefly, the evaluator successively chooses a noncanonical subterm in appropriate form and replaces it with a term closer to canonical form. It is this process of replacing such a term with another which we call *reducing* the term; the form of the replacement is given by the *reduction rules*, which are in turn derived from the computation rules of the Nuprl logic.

A given term may contain many noncanonical subterms, so some strategy for choosing the subterm to be reduced is essential. It may not be necessary to reduce all the noncanonical subterms, as a canonical term can contain noncanonical subterms. The strategy chosen is a *lazy* strategy in that it chooses a minimal number of reductions needed to reduce the term to canonical form. The evaluator cannot always succeed since there are terms which have no canonical form. However, the evaluator will succeed on any term which can be assigned a type.

Using the Evaluator

The EVAL mechanism may be run interactively by typing the command `eval` in the command window. This command replaces the `P>` prompt with the `E>` prompt. Two kinds of expressions may be entered after this prompt: Nuprl terms and bindings. Every expression must be terminated by ; ; and may extend for any number of lines. Entering a term results in that term being evaluated and its value being displayed. A binding has the form `let` *id* `=` *term*. Entering a binding results in the evaluation of the term and the resulting value being bound to *id* in the EVAL environment and this value

[1] A *closed* term is a term containing no free variables.

$\mathtt{e}_1\mathtt{+}\mathtt{e}_2$	$\mathtt{e}_1\mathtt{-}\mathtt{e}_2$
$\mathtt{e}_1\mathtt{*}\mathtt{e}_2$	$\mathtt{e}_1\mathtt{/}\mathtt{e}_2$
$\mathtt{e}_1$ `mod` $\mathtt{e}_2$	$\mathtt{ind}(\mathtt{e}; x, y.d; b; x, y.u)$
$\mathtt{list\_ind}(\mathtt{e}; b; x, y, z.u)$	$\mathtt{decide}(\mathtt{e}; x.l; y.r)$
$\mathtt{spread}(\mathtt{e}; x, y.t)$	$\mathtt{e}(t)$
$\mathtt{less}(\mathtt{e}_1; \mathtt{e}_2; t_1; t_2)$	$\mathtt{int\_eq}(\mathtt{e}_1; \mathtt{e}_2; t_1; t_2)$
$\mathtt{atom\_eq}(\mathtt{e}_1; \mathtt{e}_2; t_1; t_2)$	

Figure 5.1: The Noncanonical Forms

being displayed. The EVAL environment persists from one invocation of `eval` to the next. An EVAL session is terminated by ↑D.

For example, suppose we have a theorem named `bruce` with main goal

```
>> x:int -> y:int # (x=y in int)
```

proved so that the extracted term is `\x.<x,axiom>`. We could have the following session with the evaluator.

```
...
...
|P>eval                                  |
|E>let p1 = \x.spread(x;u,v.u);;          |
|\x.spread(x;u,v.u)                       |
|E>let t = term_of(bruce);;               |
|\x.<x,axiom>                             |
|E>t(17);;                                |
|<17,axiom>                               |
|E>p1(t(17));;                            |
|17                                       |
...
...
```

The Nuprl command `create` *name* `eval` *place* creates a library object that can contain any number of bindings (all terminated by ;;). The library objects created in this way are edited using ted and so may contain def refs. Checking an EVAL object augments the EVAL environment by evaluating the bindings it contains. All EVAL objects in a library are evaluated when the library is loaded.

The Reduction Rules

The reduction rules define how a noncanonical term may be reduced. Figure 5.1 gives a list of the noncanonical forms. The variables denoted by a

(possibly subscripted) **e** indicate so-called *argument places*, which must be occupied by canonical terms of the appropriate type before any sense can be made of the term. In the rules which follow we will use k_n to denote the canonical `int` term whose value is n. As an example consider the rule for addition:

$$k_n + k_m \Rightarrow k_{n+m}$$

We use $\Rightarrow$ to indicate that the term on the left (the *redex*) reduces to the term on the right (the *contractum*). Note that the form of the contractum depends on the values of the canonical terms appearing in the argument places of the redex. This is a property of all the rules. The rules for the other arithmetic operators are almost identical and can be obtained by replacing both occurrences of + with the appropriate operator. The rest of the rules are listed in figure 5.2. These rules embody the content of the

---

$$\begin{array}{lll}
\texttt{ind}(k_n; x, y.d; b; x, y.u) & \Rightarrow d[k_n, \texttt{ind}(k_{n+1}; x, y.d; b; x, y.u)/x, y] & \text{if } n < 0 \\
\texttt{ind}(0; x, y.d; b; x, y.u) & \Rightarrow b & \\
\texttt{ind}(k_n; x, y.d; b; x, y.u) & \Rightarrow u[k_n, \texttt{ind}(k_{n-1}; x, y.d; b; x, y.u)/x, y] & \text{if } n > 0 \\
\\
\texttt{list_ind}(\texttt{nil}; b; x, y, z.u) & \Rightarrow b & \\
\texttt{list_ind}(h.t; b; x, y, z.u) & \Rightarrow u[h, t, \texttt{list_ind}(t; b; x, y, z.u)/x, y, z] & \\
\\
\texttt{decide}(\texttt{inl}(t); x.l; y.r) & \Rightarrow l[t/x] & \\
\texttt{decide}(\texttt{inr}(t); x.l; y.r) & \Rightarrow r[t/y] & \\
\\
\texttt{spread}(\texttt{<}t_1, t_2\texttt{>}; x, y.t) & \Rightarrow t[t_1, t_2/x, y] & \\
\\
(\backslash x.b)(t) & \Rightarrow b[t/x] & \\
\\
\texttt{less}(k_n; k_m; t_1; t_2) & \Rightarrow t_1 & \text{if } n < m \\
\texttt{less}(k_n; k_m; t_1; t_2) & \Rightarrow t_2 & \text{if } n \geq m \\
\\
\texttt{int_eq}(k_n; k_n; t_1; t_2) & \Rightarrow t_1 & \text{if } n = m \\
\texttt{int_eq}(k_n; k_m; t_1; t_2) & \Rightarrow t_2 & \text{if } n \neq m \\
\\
\texttt{atom_eq}(a; b; t_1; t_2) & \Rightarrow t_1 & \text{if } a = b \\
\texttt{atom_eq}(a; b; t_1; t_2) & \Rightarrow t_2 & \text{if } a \neq b
\end{array}$$

Figure 5.2: The Nonarithmetic Reduction Rules

---

computation rules of the Nuprl logic. One somewhat anomalous term is the `term_of`(*theorem-name*) term. This term evaluates to the term extracted from the given theorem.

The Reduction Strategy

As noted in the introduction to this chapter the evaluation process consists of the repeated selection of a redex and the application of the appropriate reduction rule to it until the term is in canonical form. To make this process definite we need to specify which redex is chosen if there is more than one possibility. The goal of this choice should be that as few reductions as necessary are done to bring the term into canonical form. Now, whether or not a term is in canonical form depends only on the outer structure of the term. For example, the term `inl(spread(<1,2>;u,v.u))` is considered to be in canonical form even though it contains a redex. Thus, to minimize the number of reductions needed the choice is made to reduce the outermost redex.[2] Because of the position of the argument places in noncanonical forms, this amounts to choosing the leftmost redex when the term is written out in linear fashion. Under this strategy the term `(\`x`.<(\`y`.0)(`x`),3>)(4)` reduces to `<(\`y`.0)(4),3>` after a single application of the application reduction rule, whereas under an innermost or rightmost strategy it would reduce to `<0,3>`. This illustrates the lazy nature of this strategy.[3]

Properties of the Evaluator

The evaluator makes no use of type information and thus, because of the application reduction rule, has embedded in it an interpreter for the untyped lambda calculus. It is well known that there are terms in the untyped lambda calculus for which there are no terminating reduction sequences, or, in Nuprl terminology, which have no canonical form. For example, consider the term `(\`$x.x(x)$`)(\`$x.x(x)$`)`. The only applicable rule is the application rule, but an application of that rule leaves us with the same term.[4] However, we are mainly interested in terms which have some type in the system. The above term has instances of self–application and therefore cannot be typed in a predicative logic such as Nuprl. For terms which can be typed we stand on firmer ground. It is a metamathematical property of the system that for any closed term t, if one can find a closed type T such that the theorem t `in` T is provable, then the evaluator will reduce t to a canonical term k and furthermore the theorem t `=` k `in` T is provable. As each application of

[2] This corresponds to head reductions, normal–order evaluation or call–by–name parameter passing semantics.

[3] One might complain that for certain terms this strategy would cause more work than necessary to be performed. For instance, consider the term `(\x.(x+x))(`t`)`, where t is very expensive to evaluate. Under the strategy described above, the value of t would be computed twice. As the language we are dealing with is *functional* we know that this is unnecessary, and under the actual implementation of the evaluator it would in fact be evaluated only once. In general, for any substitution $t[t'/x]$ in the rules above, t' will be evaluated at most once and will be evaluated only if there is a free occurrence of x in t.

[4] The evaluator will try to reduce this term and in doing so will run until something intervenes, such as exhausting available memory.

a reduction rule corresponds to the use of a computation rule, it is not hard to imagine how such a proof would proceed, given the sequence of reduction rules applied.

5.3 Computational Content

Not all theorems are interesting computationally, of course. Recall the `rootf` function defined in the preceding chapter; `rootf(`n`)` returns the unique integer i such that if $n < 0$ then $i = 0$ and otherwise $i^2 \leq n < (i+1)^2$. Now consider the following two theorems, where `square(x)` is defined as `rootf(x)*rootf(x)=x in int`.

```
Thm1 >>all x:int. square(x) | ~ square(x).
Thm2 >>all x:int. square(x) vel ~ square(x).
```

The first theorem can be proved by introducing x and applying the `rootf` function to x. We can prove that either `rootf(x)*rootf(x)=x in int` or not by using the arithmetic rule, `arith`; the proof yields a decision procedure identifying which case holds. From a proof of `Thm1`, then, we can extract a function which will decide whether an integer is a square, namely `\x.decide(term_of(Thm1)(x); u.0;v.1)`. This function gives 0 if x is a square and 1 otherwise.

The second theorem has a trivial proof. We simply use the fact that P `vel ~`P holds for any proposition P and take `rootf(x)*rootf(x)=x in int` as P. However, there is no interesting computational content to this result; we do not obtain from it a procedure to decide whether x is a square. (The interested reader should prove this theorem and display its computational content.)

5.4 An Example

The following is a small proof in Nuprl with the extraction clauses shown explicitly. The system does not produce the comments `(* ... *)`; we include them for illustrative purposes only. In the interest of condensing the presentation we also elide hypothesis lists wherever possible.

```
,-------------------------------------------------------------,
|EDIT THM thm                                                 |
|-------------------------------------------------------------|
|* top  (* term_of(thm) extracted *)                          |
|>> all x,y:int.(x=y in int)|some z:int.~(z=0 in int)#(x+z=y  |
|   in int                                                    |
|                                                             |
|BY quantifier (* ext \x.\y.e1 (quantifier tactic does        |
|                               intro twice)     *)           |
|1* 1. x:(int)                                               |
|   2. y:(int)                                                |
|   >> ((x=y in int)|some z:int.~(z=0 in int)#(x+z=y in int))|
|                     (* ext e1 *)                            |
|2* >> (int) in U1                                            |
|                                                             |
|3* >> (int) in U1                                            |
'-------------------------------------------------------------'
```

The rule `quantifier` is a *tactic*, a user–defined rule of inference, which successively applies the `intro` rule to each universal quantifier; in this case it performs two applications. This tactic was written by a user of the system and must be loaded as an ML object to be used; in general it would not be available. See the following chapter for more on tactics. The extracted term is a function which, given any values for x and y, will give a proof of the body; this corresponds to the intuitive meaning of `all` and justifies its definition as a dependent function. Note that $e1$ is the term extracted from the first subgoal; the next window shows the proof of this subgoal.

```
,-------------------------------------------------------------,
|EDIT THM thm                                                 |
|-------------------------------------------------------------|
|* top 1  (* e1 extracted *)                                  |
|1...2.                                                       |
|>> ((x=y in int)|some z:int.~(z=0 in int)#(x+z=y in int))    |
|                                                             |
|BY seq (x=y in int)|~(x=y in int) new E (* ext \E.e3(e2) *)  |
|                                                             |
|1* 1...2.                                                    |
|   >> (x=y in int)|~(x=y in int)  (* ext e2 *)               |
|                                                             |
|2* 1...2.                                                    |
|   3. E:(x=y in int)|~(x=y in int)                           |
|   >> ((x=y in int)|some z:int.~(z=0 in int)#(x+z=y in int))|
|                      (* ext e3 *)                           |
'-------------------------------------------------------------'
```

The `seq` rule extracts a function which in effect substitutes the proof of

the `seq` term, $e2$, for each instance of its use in $e3$ (under the name `E`).

```
,------------------------------------------------------------,
|EDIT THM thm                                                |
|------------------------------------------------------------|
|* top 1 1  (* e2 extracted *)                               |
|1...2.                                                      |
|>> (x=y in int)|~(x=y in int)                               |
|                                                            |
|BY arith (* ext int_eq(x;y;inl(axiom);inr(axiom)) *)        |
|                                                            |
|1* 1...2.                                                   |
|   >> x in int                                              |
|                                                            |
|2* 1...2.                                                   |
|   >> y  in int                                             |
|                                                            |
|3* 1...2.                                                   |
|   >> x in int                                              |
|                                                            |
|4* 1...2.                                                   |
|   >> y  in int                                             |
'------------------------------------------------------------'
```

The `arith` decision procedure produces the `int_eq` term, a decision procedure for equality on the integers.

```
,------------------------------------------------------------,
|EDIT THM thm                                                |
|------------------------------------------------------------|
|* top 1 2 (* e3 extracted *)                                |
|1...2.                                                      |
|3. E:(x=y in int)|~(x=y in int)                             |
|>> ((x=y in int)|some z:int.~(z=0 in int)#(x+z=y in int))   |
|                                                            |
|BY elim 3 (* ext decide(E;l.e4;r.e5) *)                     |
|                                                            |
|1* 1...3.                                                   |
|   4. l:(x=y in int)                                        |
|   >> ((x=y in int)|some z:int.~(z=0 in int)#(x+z=y in int))|
|                               (* ext e4 *)                 |
|2* 1...3.                                                   |
|   4. r:~(x=y in int)                                       |
|   >> ((x=y in int)|some z:int.~(z=0 in int)#(x+z=y in int))|
|                               (* ext e5 *)                 |
'------------------------------------------------------------'
```

The `elim` rule for disjoint unions produces a `decide` term. In the next frame the term `l` is extracted from the first subgoal because our goal is exactly `l` (the actual proof of this subgoal is not shown but is carried out using the rule `hyp 4`).

```
,-------------------------------------------------------------,
|EDIT THM thm                                                 |
|-------------------------------------------------------------|
|* top 1 2 1  (* e4 extracted *)                              |
|1...3.                                                       |
|4. l:(x=y in int)                                            |
|>> ((x=y in int)|some z:int.~(z=0 in int)#(x+z=y in int))    |
|                                                             |
|BY intro left  (* ext inl(e6) *)                             |
|                                                             |
|1* 1...4.                                                    |
|   >> (x=y in int)  (* ext e6==l (follows by hyp 4) *)       |
|                                                             |
|2* 1...4.                                                    |
|   >> some z:int.~(z=0 in int)#(x+z=y in int) in U1          |
'-------------------------------------------------------------'
```

In the next frame `r` is extracted for the same reason as `l` is above.

```
,-------------------------------------------------------------,
|EDIT THM thm                                                 |
|-------------------------------------------------------------|
|* top 1 2 2  (* e5 extracted *)                              |
|1...3.                                                       |
|4. r:~(x=y in int)                                           |
|>> ((x=y in int)|some z:int.~(z=0 in int)#(x+z=y in int))    |
|                                                             |
|BY intro right (* ext inr(e7) *)                             |
|                                                             |
|1* 1...4.                                                    |
|   >> some z:int.~(z=0 in int)#(x+z=y in int) (ext e7 *)     |
|                                                             |
|2* 1...4.                                                    |
|   >> ((x=y in int)) in U1                                   |
'-------------------------------------------------------------'
```

In the next frame, since `some` is defined to be a dependent product we extract a pair consisting of an element of the integers, `y-x`, and a proof, *e8*, that the proposition is true for this element. This corresponds to the desired meaning of `some`.

```
,-------------------------------------------------------------,
|EDIT THM thm                                                 |
|-------------------------------------------------------------|
|* top 1 2 2 1 (* e7 extracted *)                             |
|1...3.                                                       |
|4. r:~(x=y in int)                                           |
|>> some z:int.~(z=0 in int)#(x+z=y in int)                   |
|                                                             |
|BY intro y-x  (* ext <y-x,e8> *)                             |
|                                                             |
|1* 1...4.                                                    |
|   >> y-x in (int)                                           |
|                                                             |
|2* 1...4.                                                    |
|   >> ((y-x=0  in int)->void)#(x+y-x=y  in int) (* ext e8 *)|
|                                                             |
|3* 1...4.                                                    |
|   5. z:(int)                                                |
|   >> (~(z=0 in int)#(x+z=y in int)) in U1                   |
'-------------------------------------------------------------'
```

The product `intro` produces a pair consisting of proofs of the two subgoals. Although the proof is not shown here, subgoal 2 can be seen to follow trivially by `arith`.

```
,-------------------------------------------------------------,
|EDIT THM thm                                                 |
|-------------------------------------------------------------|
|* top 1 2 2 1 2  (* e8 extracted *)                          |
|1...3.                                                       |
|4. r:~(x=y in int)                                           |
|>> ((y-x=0 in int)->void)#(x+y-x=y in int) (* ext <e9,e10>*)|
|                                                             |
|BY intro                                                     |
|                                                             |
|1* 1...4.                                                    |
|   >> (y-x=0  in int)->void  (* ext e9 *)                    |
|                                                             |
|2* 1...4.                                                    |
|   >> x+y-x=y  in int (* ext e10==axiom (follows by arith)*)|
'-------------------------------------------------------------'
```

In the next frame we see that hypotheses 4 and 5 contradict each other. The system somewhat arbitrarily extracts `(\v0.any(v0))(r(axiom))` from this (we could extract anything we desired from a contradiction).

```
,-------------------------------------------------------------,
|EDIT THM thm                                                 |
|-------------------------------------------------------------|
|* top 1 2 2 1 2 1 (* e9 extracted *)                         |
|1...3.                                                       |
|4. r:~(x=y in int)                                           |
|>> (y-x=0  in int)->void                                     |
|                                                             |
|BY intro new f (* ext \f.e11 *)                              |
|                                                             |
|1* 1...4.                                                    |
|   5. f:y-x=0  in int                                        |
|   >> void  (* ext e11==(\v0.any(v0))(r(axiom))              |
|                                      (from a contradiction)*)|
|                                                             |
|2* 1...4.                                                    |
|   >> (y-x=0  in int) in U1                                  |
'-------------------------------------------------------------'
```

We could now have the following session with the evaluator. What follows has additional *white space* (spaces and carriage returns) to improve readability.

```
,-------------------------------------------------------------,
|P>eval                                                       |
|E>term_of(thm);;                                             |
|\x.\y.(\E.decide(E;l.inl(l);                                 |
|        r.inr(<y-x,<\f.(\v0.any(v0))(r(axiom)),axiom>>)))    |
|       (int_eq(x;y ;inl(axiom);inr(axiom)))                  |
|E>term_of(thm)(7)(7);;                                       |
|inl(axiom)                                                   |
|E>term_of(thm)(7)(10);;                                      |
|inr(<10-7,<\f.(\v0.any(v0))(axiom(axiom)),axiom>>)           |
|E>let p1 = \x.spread(x;u,v.u);;                              |
|\x.spread(x;u,v.u)                                           |
|E>let d = \y.decide(y;u.p1(u);u.p1(u));;                     |
|\y.decide(y;u.p1(u);u.p1(u))                                 |
|E>d(term_of(thm)(7)(10));;                                   |
|3                                                            |
'-------------------------------------------------------------'
```

To terminate an `eval` session type ↑D.

Chapter 6

Proof Tactics

In the study of logic a distinction is made between the logic being studied, the *object language*, and the language in which the study of the logic is conducted, the *metalanguage*. In our case the object language is the type theory Nuprl. Thus far in this book we have conducted the discussion of Nuprl using a combination of English and mathematics as an informal metalanguage. In this chapter we introduce a formal metalanguage based upon the ML programming language [Gordon, Milner, & Wadsworth 79].

In ML the various parts of the object language—terms, declarations, proofs and rules—are data types. By defining a formal metalanguage we have made concrete the structure and elements of the object language. We can then write ML programs that manipulate objects of the object language. Thus, for example, we can write a program to return the subterms of a term or one that substitutes a term for a free variable in a term. More importantly, we can write ML functions which search for or transform proofs. We can then use such automated proof techniques and theorem–proving heuristics, *tactics*, while writing proofs.

A tactic is a function written in ML which partially automates the process of theorem proving in Nuprl. In any logic writing a formal proof is a combination of insightful, interesting steps and tedious ones. Being required to fill in every detail of a proof distracts one's attention from the *important* parts of the proof and can be quite tiresome. The refinement logic of Nuprl is no exception to this. Given undecidability, we cannot hope to fully automate the theorem–proving process. Ideally, then, we would want the user to enter only the *central ideas* of a proof and leave the details to be filled in by the system. Tactics offer a mechanism whereby a sketch of a proof supplied by the user can often be completed automatically and many simple proofs can be avoided altogether. Tactics can also be used to automate more difficult patterns of proofs that occur frequently so that essentially the same argument need not be repeated; a tactic for that pattern of proof would be

invoked instead. The uses of tactics are many and need not be restricted to the above. There is the potential in the tactic mechanism to express any decision procedure, semidecision procedure or theorem–proving algorithm which is valid in the Nuprl logic. As will be explained in detail below, the implementation of the metalanguage makes sure that all proofs produced by tactics are in fact valid Nuprl proofs. This makes it impossible to write or use a tactic that produces an incorrect proof.[1]

In Nuprl there are predefined tactics that are always available. These include tactics of general use in theorem proving. The user can easily add new tactics that extend the predefined tactics or are specific to the domain of theorems being proved. The next two sections describe two different kinds of tactics: refinement tactics and transformation tactics. This is followed by an introduction to the ML programming language and an explanation of writing simple tactics. Chapter 9 contains a more detailed description of the metalanguage interface and information on writing tactics and examples of more complicated tactics.

6.1 Refinement Tactics

A refinement tactic resembles a derived rule of inference. Such a tactic is invoked by typing the name of the tactic where a refinement rule is requested by the proof editor. When the tactic is applied it tries to prove the current goal. There are three possible outcomes from a tactic invocation.

1. If the tactic completely proves the current goal then the name of the tactic name is entered as the rule name, and the status of the sequent is changed to *complete*. This refinement generates no subgoals.

2. If the tactic produces part of a proof of the current goal then the tactic is entered as the rule name, and the status of the sequent is changed to *incomplete*. Each of the subgoals left unproved in the proof produced by the tactic is listed as a subgoal to the refinement.

3. If the tactic fails then an error message is displayed in the command window, and the status of the sequent is changed to *bad*.

Thus a refinement tactic may completely prove a goal, partially prove the goal, leaving some unproved subgoals, or fail.

The behavior of a refinement tactic resembles that of a primitive rule of inference or decision procedure (such as arith). However, although refinement tactics are similar to rules of inference, they should not be thought of as fixed rules; within the tactic, for instance, the goal can be examined and different refinements can be applied based upon this analysis. In addition,

[1] A tactic may fail to produce any proof at all, but it will not produce an invalid proof.

by using the failure mechanism that is primitive to the ML language a tactic may approach a goal using one technique, find that this leads to a dead–end, backtrack and try another approach. Thus while the result of a tactic may represent just a few primitive refinement steps, the method by which those few steps were arrived at may have required a substantial amount of computation.

6.2 Transformation Tactics

Whereas refinement tactics take only goals as arguments and return proofs, transformation tactics take proofs as arguments and return proofs. Transformation tactics can perform much more global analysis and change to proofs than can refinement tactics. These tactics, for instance, can complete or expand an unfinished proof, produce a new proof that is analogous to the given proof and perform various optimizations to the proof such as replacing subproofs with more elegant or concise ones, to name a few of the uses to which they have been put.

A user invokes a transformation tactic by editing the theorem of interest and traversing the proof until the goal heading the desired subproof is displayed. The user then types ↑T, and Nuprl will prompt for the name of the transformation tactic which is to be applied. The tactic is applied to the proof consisting of the current goal and anything that is below it, that is, any subgoals and proofs below the subgoals. The transformation tactic, if it succeeds, will return a proof which has the same goal as the argument proof tree. Unlike those of refinement tactics, the name of the transformation tactic is not entered into the proof; the tactic serves as an operation on proofs, *transforming* one proof to another. Upon termination the proof returned may bear little resemblance to the original proof, except that a transformation tactic cannot change the proof above the root of the argument, nor can it change the goal of the root of the subproof. The only exception to this rule is if the subproof is the whole proof—the tactic was called from the top goal. In this case the transformation tactic may replace the goal of the proof; thus a transformation tactic can map one theorem into another. These constraints are enforced by the implementation and need not be the concern of the tactic designer or user.

6.3 Writing Simple Tactics

The Metalanguage ML

Writing complicated tactics requires a thorough knowledge of the ML programming language [Gordon, Milner, & Wadsworth 79] and the information

contained in chapter 9. It is an easy matter, however, to write simple tactics and to combine existing tactics with a minimal subset of ML. ML is a functional programming language with three important characteristics which make it a good language for expressing tactics:

1. ML has an extensible, polymorphic type discipline with secure types. This allows type constraints on the arguments and results of functions to be expressed and enforced. For example, the result of a function may be constrained to be of type `proof`.

2. ML has a mechanism for raising and handling exceptions (or, in the terminology of ML, throwing and catching failures). This is a convenient way to incorporate backtracking into tactics.

3. ML is fully higher–order; functions are objects in the language. This allows tactics (which are functions) to be combined using combining forms called tacticals, all of which are written in ML.

In order to understand the example tactics presented below it is not necessary to know many of the details of ML. The following summarizes some of the more important and less obvious language constructs. Functions in ML are defined as in

```
let divides x y = ((x/y)*y = x);;
```

This function has type `int→int→bool`; that is, it maps integers to functions from integers to boolean values. There is also an explicit abstraction operator, \, which stands for the more conventional λ. The previous function could have been defined as

```
let divides = \ x . \ y  . ((x/y)*y = x);;
```

Exceptions are raised using the expression "`fail`" and handled (*caught*) using "?". The result of evaluating *exp1?exp2* is the result of evaluating *exp1* unless a failure is encountered, in which case it is the result of evaluating *exp2*. For example, the following function returns *false* if $y = 0$.

```
let divides x y = (if y = 0 then fail
                   else (y*(x/y)=x)
                   ) ? false;;
```

In fact, because dividing by 0 causes a failure, we could define the same function with

```
let divides x y = (y*(x/y)=x)?false;;
```

Using ML from Nuprl

From Nuprl the user can request that ML be run interactively by entering the command `ml` in the command window. Nuprl will respond with a `M>` prompt in the command window. Any input typed after this prompt is processed by ML. This facility allows for the examination of variables and constants in ML and for the tracing, definition and debugging of tactics. The user leaves the interactive ML mode by typing ↑D.

Although definitions can be entered interactively using this mode, the definitions entered are only temporary. Once the current Nuprl session has been concluded, any changes to the ML state disappear. If more permanent definitions are desired an ML object can be created and edited using the text–editing facilities of Nuprl, and the ML commands in this ML object can be evaluated by ML. To do this a library object of type "ML" is created and then edited in Nuprl. Once complete this ML object can be *checked*, during which process the commands of the ML object are evaluated sequentially by ML. If there are no errors in the object it is marked as good, and any changes in the ML state implicit in the commands will remain in effect throughout the remainder of the Nuprl session. Should an error be encountered while evaluating the ML object an error message will be displayed, and evaluation of the ML object will be discontinued. Any error–free commands that preceded the one with an error will have been evaluated with any changes in state remaining in effect. The user may then view the ML object, make the necessary corrections and recheck it. The commands in an ML object will also be evaluated when it is loaded or restored to the library.

Example Tactics

We now examine how some simple tactics are written. One should think of tactics as mappings from proofs to proofs which may take additional arguments. The simplest tactics are formed using the ML function `refine_using_prl_rule`. This function takes two arguments, a token (the ML equivalent of a string) and a proof. The token is parsed as a rule, and then the goal of the proof is refined by this rule. Conceptually the result is a proof with the top goal refined by the rule indicated by the token and with the appropriate subgoals for this refinement. Thus `refine_using_prl_rule` is a procedural embodiment of the primitive refinement rules of Nuprl. The function `refine_using_prl_rule` is an example of a tactic.

Atomic refinement steps and existing tactics may be combined using a combining form called a *tactical*. For example, the following tactic refines a goal by `intro`; then, each of the resultant subgoals is again refined by `intro`. To each of the subgoals of this the predefined tactic `immediate` is applied.

```
let two_intro proof =
  (refine_using_prl_rule `intro` THEN
```

```
refine_using_prl_rule `intro` THEN
  immediate) proof;;
```

The tactical `THEN` is an infix operator; it takes two tactics and produces a new tactic. When applied to a proof this new tactic applies the tactic on the left–hand side of the `THEN` and then, to each of the subgoals resulting from the left–hand tactic, the right–hand tactic. Note that `two_intro` will only work if it *makes sense* to perform two levels of introductions. For example, if the tactic is applied to a dependent product (where an intro rule requires an object of the left type) the refinement will fail, causing the tactic to fail.

In addition to `THEN`, two other tacticals are of general usefulness: `REPEAT` and `ORELSE`. The tactical `REPEAT` will repeatedly apply a tactic until the tactic fails (or fails to do anything). That is, the tactic is applied to the argument proof, and then to the children produced by tactics, and so on. The `REPEAT` tactical will catch all failures of the argument tactic and can not generate a failure. For example,

```
let intros = REPEAT (refine_using_prl_rule `intro`);;
```

will perform (simple) introduction until it no longer applies. The `ORELSE` tactical takes two tactics as arguments and produces a tactic that applies the –hand tactic to a proof and, if that tactic fails, the right–hand tactic. The following tactic performs all simple introductions and eliminations which are possible.

```
let goal_simplify =
  REPEAT ((refine_using_prl_rule `intro`)
          ORELSE (refine_using_prl_rule `elim`));;
```

Using `refine_using_prl_rule`, the four tacticals given here and the predefined tactics, many useful tactics can be written. We hope that tactics are easy enough to write and convenient enough to use that the user will be encouraged to experiment with them. In particular, we hope that the user will write tactics for any aspect of theorem proving that is repetitious, stylized or routine.

The power and flexibility of the tactic mechanism go well beyond these simple examples. In chapter 9 we continue the discussion begun here.

The Auto–tactic

Nuprl provides the facility for a distinguished transformation tactic known as the auto–tactic. This tactic is invoked automatically on the results of each primitive refinement step. After each primitive refinement performed in the refinement editor the auto–tactic is run on the subgoals as a transformation tactic. There is a default auto–tactic, but any refinement or transformation

tactic may be used. For example, to set the auto–tactic to a transformation tactic called `transform_tac`, the following ML code is used:

```
set_auto_tactic `transform_tac`;;
```

This code could be used either in an ML interactive window or placed in an ML object (and checked). To see what `auto_tactic` is currently set to, type

```
show_auto_tactic ();;
```

in the ML interactive window.

In general, an `auto_tactic` should either completely prove a subgoal or leave it unchanged. To ensure this the `COMPLETE` tactical can be applied to the intended tactic. This tactical changes the argument tactic so that partial results (i.e., those that still have unproved subgoals) are turned into failure. For example, to use `transform_tac` as the auto–tactic and to make it so that partial results are not used, use the command

```
set_auto_tactic `COMPLETE transform_tac`;;
```

To set `auto_tactic` to a refinement tactic `immediate` where partial results are not used, use the following:

```
set_auto_tactic `COMPLETE
                (refine (tactic \`immediate\`))`;;
```

This is the default setting of `auto_tactic`. When the auto–tactic fails the failure is not reported; rather, the subgoal is just unchanged. By watching the status characters on subgoals one can tell which have been proved by the auto–tactic and which still need to be proved.

The auto–tactic is applied to subgoals of primitive refinements only when a proof is being edited. When performing a library load or when using a tactic while editing a proof, the auto–tactic is inhibited.

Chapter 7

System Description

This chapter describes various parts of the Nuprl system: the command language, library, windows, editors and notation definitions. With the *command language* one can create, delete and manipulate library objects, evaluate terms, call up the editors, and "check" (i.e., parse and generate code for) objects. The *library* contains the various user–defined objects, the *editors* provide ways to modify them, *definitions* make them more readable, and the *window system* makes the editors more useful.

7.1 The Command Language

When a "`P>`" prompt appears in the command/status window, the command processor is waiting for a command.[1] Commands may be abbreviated to their shortest unique prefixes; for example, `check` can be abbreviated to `ch`, and `create` can be abbreviated to `cr`. Typing errors are corrected with the DELETE key (to delete a character) and the ERASE key (↑U) to delete the whole line. The commands are divided into four groups: a group consisting of commands that control the library window, a group consisting of commands that manipulate objects, a group consisting of commands that allow work to be saved between sections, and a miscellaneous group.

The following names are used in the syntactic descriptions below.

object is the name of an object in the library or one of the reserved names `first` or `last`

objects is *object* or *object-object*. This provides a convenient way to refer to a range of entries.

[1] If the cursor is within an edit window COMMAND must be pressed to get the "`P>`" prompt.

place is `top`, `bot`, `before` *object* or `after` *object*. (`Top` refers to the slot immediately before the first entry, and `bottom` refers to the slot immediately after the last entry.)

kind is `thm`, `def`, `eval` or `ml`.

filename is any legal Unix or Lisp Machine file name, as appropriate.

Optional arguments and keywords are indicated by curly braces.

Library Display

The library is a list of the objects defined by the user and may contain the theorems, definitions, evaluation bindings and ML code. Information about the library objects is displayed in the *library window*. The `jump` and `scroll` commands change what is being displayed in the library window, while the `move` command changes the order of library objects. The `move` command is often used to make sure that objects occur before their uses; this is important for correct library reloading.

`jump` *object*
: The library is redisplayed so that the specified entry is visible.

`move` *objects place*
: The specified library objects are moved to the position just after the specified place.

`scroll` {`up`} {*number*}
: The library window is moved the specified number of entries, in the specified direction. The default direction is down, and the default number of entries is one. For example, `scroll 5` shifts the window down five entries, while `scroll up` shifts the window up one entry.

Manipulating Objects

These are the commands that directly manipulate objects. The most used of these commands is `view`, since most interaction with the system takes place within the editors started up by the `view` command.

`archive` *objects*
: For each object name specified a copy of the object is made and saved as the *old* version of the object. The library window is redisplayed if any of the archived objects are visible in the window. Note that there is no way to delete, modify or unarchive an object.

`check` *objects*

Each of the indicated objects is checked. Checking an object makes the objects defined therein available for use in other objects. When an object is checked, its status is updated, as is the status of any object that references any of the specified objects.[2] If necessary the library window is redisplayed to show the new statuses.

Checking a large theorem can take awhile. If the body of the theorem was dumped to a file and has not been fully reloaded (see the `dump` and `load` commands below) then checking the theorem brings in and reconstructs the body of the proof.

`create` *name kind place*

A new library entry of the specified name and kind is created at the specified place. The library is redisplayed so that the newly created object is the first object in the library window. The object will be empty and its status will be RAW.

`delete` *objects*

The specified objects are removed from the library. The status of any entry depending on these objects is set to BAD. (This status change has been fully implemented only for definition objects.)

`view` {`old`} *object*

The object is displayed in a new window. If `old` is specified the archived version of the object will be viewed and the view will be read–only; otherwise, the unarchived version of the object will be viewed. If the object is not already being viewed the new view will be fully editable; otherwise, it and all other views of the object will be made read–only. The header line of the view will say `SHOW` for a read–only view and `EDIT` for an editable view.

The editor used depends on the kind of object. The refinement editor (*red*) is used on theorems, while the text editor (*ted*) is used for other objects. For more information on ted and red see sections 7.4 and 7.5, respectively. Typing ↑D in an edit window will make Nuprl delete the window and end its editing of the object. If there are other edit views on the screen the cursor will be placed in one of them; otherwise, the cursor will go back to the command window.

When `view` is done on a large theorem, it may take some time to activate red. If the body of the theorem has been dumped but not fully reloaded (see the `dump` and `load` commands below) then Nuprl reads the body in and reconstructs it before starting the editor.

[2] Note: This has not been fully implemented.

Storing Results

The `load` and `dump` commands let one save results between Nuprl sessions.

`dump` *objects* {`to` *filename*}
: Nuprl writes a representation of the selected objects from the current library to the file. If no filename is given the name of the last file to be loaded is used. The dumped objects are not removed from the library.

`load` *place* `from` *filename*
: This is the command used to read in a file produced by `dump`. The decoded library entries are added to the library starting at location *place*. If an object in the dumped file has the same name as an object already in the library, the object in the file is ignored. When the load is finished a message is displayed giving the number of objects loaded and the number of duplicates discarded. The name of the loaded file is displayed in the header of the library window.

  Theorem objects are not fully reconstructed. The top of the proof (its goal and status) is loaded, but not the actual body of the proof (the rule applications and subgoals). The body is loaded on demand when the theorem is checked or viewed or when the extractor needs the body of the proof. When the body is actually loaded, the theorem's status is recalculated. If the new status is different from the status the theorem had when it was dumped, a warning is given.

Be sure to keep library objects correctly sorted; if an object a is referenced by an object b then a must appear before b in the library, or reloading will fail.

General Commands

This section contains those commands which do not fit in any specific category.

`eval` Puts the command window into *eval* mode. The prompt changes from "`P>`" to "`EVAL>`" to indicate the change. Two kinds of inputs can be entered in eval mode: terms and bindings. Each input must be terminated by a double semicolon, "`;;`". If a term is entered Nuprl evaluates it and prints its value. Bindings have the form `let` *id* = *term*—if a binding is entered Nuprl evaluates the term, prints its value and binds the value to the identifier. The evaluator's binding environment persists from one eval session to the next. It also includes bindings made when EVAL objects are checked. Eval mode is terminated by ↑D.

`shell` Creates an interactive subshell. This command is available only on the Unix version of Nuprl. Upon return from the shell the screen is

redisplayed with its old contents, and processing is resumed where it left off. The stty suspend character, ↑Z, will suspend Nuprl without creating a subshell.

On Lisp Machines `select-L` brings up a Lisp Listener window, and `select-N` returns to Nuprl. A `select-N` will leave the screen blank; use ↑L to redraw it.

`exit` Terminates the current Nuprl session. Under Unix the user is returned to Lisp. To exit Lisp, use `(exit)`. On a Lisp Machine the user is returned to the Lisp Listener.

There is one final command, the PRINT command (↑P); it causes a snapshot of the current screen image to be appended to the user's snapshot file. On the Unix version of Nuprl the snapshot file is `snap-`*userid*. On Lisp Machines the snapshot file is `snapshot.lisp`.

7.2 The Library

The library is a list of all the objects defined by the user and includes theorems, definitions, evaluation bindings and ML code. The library window displays information about the various objects— their names, types, statuses and summary descriptions. In this section we give a short description of the different kinds of objects and their statuses.

Object Types

The library contains four kinds of objects: DEF, THM, EVAL and ML. DEF objects define new notations that can be used whenever text is being entered. THM objects contain proofs in the form of proof trees. Each node of the proof tree contains a number of assumptions, a conclusion, a refinement rule and a list of children produced by applying the refinement rule to the assumptions and conclusion. THM objects are checked only when a check command is issued or they are viewed or used as objects from which code is extracted.

EVAL objects contain lists of bindings, where a binding has the form `let` *id* = *term* and is terminated by a double semicolon, ";;". Checking an EVAL object adds its bindings to the evaluator environment so that they available to the `eval` command. All EVAL objects are checked when they are loaded into the library. ML objects contain ML programs, including tactics, which provide a general way of combining the primitive refinement rules to form more powerful refinement rules. Checking an ML object enriches the ML environment. All ML objects are checked when they are loaded into the library.

Library Entries

Every object has associated with it a status, either *raw*, *bad*, *incomplete* or *complete*, indicating the current state of the object. A *raw* status means an object has been changed but not yet checked. A *bad* status means an object has been checked and found to contain errors. An *incomplete* status is meaningful only for proofs and signifies that the proof contains no errors but has not been finished. A *complete* status indicates that the object is correct and complete.

An entry for an object in the library contains its status, type, name and a summary of the contents. The status is encoded as a single character: `?` for raw, `-` for bad, `#` for incomplete and `*` for complete. A typical entry might be the following.

```
,-----------------------------------------------------,
| Library                                             |
|-----------------------------------------------------|
| * simple                                            |
|   THM   >> int in U1                                |
|                                                     |
|                                                     |
'-----------------------------------------------------'
```

The library window provides the mechanism for viewing these entries. The entries are kept in a linear order, and at any one time a section of the entries is visible in the library window. The library placement commands described in section 7.1 can be used to change the order of the entries and to move the window around within the list of entries.

7.3 Window Management

All interaction with Nuprl happens inside a window of one kind or another. Except for the command/status and library windows, which are permanent, windows are dynamically created and deleted as necessary. There are no explicit Nuprl commands to manipulate windows; their sizes, locations and foreground/background depths can be changed only by using the mouse.

Mouse Mode

When the terminal does not have a mouse, Nuprl uses the keypad to simulate one. `MOUSE-MODE` appears in the command/status window during this simulation. To enter mouse mode, press the MOUSE key; to leave it, press DIAG. When the system is in mouse mode the screen cursor shows the position of the *mouse cursor*; the position of the *normal cursor* is remembered (though

not shown) and is restored when mouse mode is left. Only the keypad keys are meaningful in mouse mode, and they have meanings similar to (though different from) their usual ones; the exact meanings are given below.[3]

Positioning the Mouse Cursor

In mouse mode the arrow keys move the mouse cursor in the following ways:

↑, ↓
: Moves the mouse cursor up or down a line.

←, →
: Moves the mouse cursor left or right a character.

LONG ↑, LONG ↓
: Moves the mouse cursor up or down eight lines.

LONG ←, LONG →
: Moves the mouse cursor left or right eight spaces.

LONG LONG ↑, LONG LONG ↓
: Moves the mouse cursor to the top or bottom of the screen.

LONG LONG ←, LONG LONG →
: Moves the mouse cursor to the left or right edge of the screen.

All other keys and key combinations (except for DIAG) are ignored.

Adjusting and Moving Windows

To adjust or move a window in the Unix implementation of Nuprl, get into mouse mode, move the mouse cursor to the window to be affected, and press MOUSE. A *window command menu* will pop up. Move the mouse cursor so that it inside this new window and on the same line as the desired command. Press MOUSE again. Some commands require a third MOUSE press; see the list of commands below for details. If the second MOUSE press is made when the mouse cursor isn't inside the menu, Nuprl will move the menu to wherever the mouse cursor is and will wait for a new second MOUSE press, to specify a window command.

On a Lisp Machine one may use the actual mouse; follow the instructions above for Unix Nuprl, but use middle–click instead of the MOUSE key.

The commands and their meanings are as follows:

[3] Note that if the command window is hidden from view, one can enter mouse mode unwittingly. If this happens it will appear as though Nuprl is not responding to any input except for the arrow keys.

`PULL` Brings the window into the foreground. Any windows which are on top of the window are pushed behind it.

`PUSH` Moves the window into the background. This may cause it to be overlapped by or hidden by other windows. All obscured parts of the window are remembered, though invisible.

... *Elides* the window by shrinking it down so that only the header line is displayed. It also moves the window to the elided window area at the top right of the screen. If ... is invoked on an elided window it returns to the same size and position it had before it was elided. The contents of the window are not destroyed by this process; they simply are not shown in the interim.

`MOVE` Upon a third press of MOUSE the location of the mouse cursor will be the new location of the upper left corner of the current window.

`SIZE` Also waits for a third MOUSE press. The location of the mouse cursor will be the new location of the lower right corner of the current window. The location of the upper left corner will not change.

`KILL` Destroys the window. This command is obsolete and should not be used.

Other Keys in Mouse Mode

The following keys have special meanings in mouse mode:

SEL Is only meaningful if the mouse cursor is in a text editor window; a SEL is done on the location nearest the mouse cursor. (See section 7.4 for more information on text selections.)

JUMP This makes the view containing the mouse cursor the current view (the view in which the normal cursor will be placed when mouse mode is exited).

DIAG Mouse mode is exited. The cursor will be in the window in which it was when mouse mode was entered (unless JUMP was used). If that window has been destroyed another window is selected.

7.4 The Text Editor

Nuprl text consists of sequences of characters and definition references (also known as *def refs*). Since these sequences are stored internally as recursive trees, they are also known as *text trees*, or *T-trees*. The text editor, *ted*, is a structure editor for these trees and works in a way similar to that of the

Cornell Program Synthesizer; the cursor on the screen represents a cursor into the text tree being edited, and movements of the tree cursor are mapped into movements of the screen's cursor.

The text editor is used to create and modify DEF, ML and EVAL objects as well as rules and main goals of theorems. Changes to objects are reflected immediately; there is no separate "save" or "unsave" command.

The text editor is almost completely modeless. To insert a character, type it; to insert a def, *instantiate* it; to remove a character, delete it or KILL it; to move the cursor, use the arrow keys; to exit the editor, use ↑D. There is a *bracket* mode, explained in the next section, which changes the way def refs are displayed.

Tree Cursors, Tree Positions and Bracket Mode

A *tree cursor* is a path from the root of a text tree to a *tree position*, namely a character, a def ref or the end of a subtree. When text is being edited the tree cursor is mapped to a location on the screen, and this is where the editor places the screen cursor. Unfortunately, more than one tree position can map to a screen position. Take the following definition of the $\geq$ relation:[4]

```
<x>>=<y>==((<x><<y>) -> void).
```

In the tree represented by `3>=1` there are six tree positions: the entire def, the character `3`, the end of the subtree containing `3`, the character `1`, the end of the subtree containing `1` and the end of the entire tree. These six tree positions are mapped into four screen positions:

```
3>=1
^^ ^^
```

The first is for both the entire def and the character `3`, the second is for the end of the subtree containing `3`, the third is for the character `1`, and the fourth is for the end of the subtree containing `1` and also for the end of the entire tree. If the screen cursor were under the `3`, pressing the KILL key would either delete the entire def or just the `3`, depending on where the *tree* cursor actually was.

To eliminate this ambiguity the window may be put into *bracket mode*; in this mode all def refs are surrounded by square brackets, and the screen position that corresponds to the entire def ref is the left square bracket:

```
[3>=1]
^^^ ^^^
```

Notice that the right square bracket corresponds to the end of the subtree containing `1`; this gets rid of the other ambiguous screen position.

[4] A note for experienced Nuprl users: no backslashes are required in this def.

For another example of an ambiguity removed by bracket mode, consider the two trees

```
[23>=1]   and   2[3>=1].
```

The first is quite reasonable, but the second is probably a mistake, since it expands to `2((3<1) -> void)`. In unbracketed mode both trees would be displayed as `23>=1`.

The BRACKET toggle, ↑B, switches the current window into or out of bracket mode. It cannot be used in mouse mode.

Moving the Cursor

The cursor can be cycled through the edit windows with the JUMP key. This is most useful when copying text from one window to another. To move the cursor around within the current window, use the arrow and LONG keys, as follows.

↑, ↓
: Move the screen cursor up or down one line. The tree cursor is moved to the tree position that most closely corresponds to the screen position one line up or down. If the new tree position is not already being shown in the text window, scroll the window.

←, →
: Move the tree cursor left or right one position. That is, move one position backward or forward in a preorder walk of the tree. Keep in mind that this one–position change of the tree cursor doesn't always correspond to a one–column change of the screen cursor.

LONG ↑, LONG ↓
: Move up or down four lines in the screen. If necessary, scroll the window.

LONG ←, LONG →
: Move the tree cursor left or right four positions.

LONG LONG ↑, LONG LONG ↓
: Move up or down a screen.

LONG LONG ←, LONG LONG →
: Move the screen cursor to the left or right end of the current line. The tree cursor is moved left or right to match.

If LONG is used before any other key besides an arrow key it is ignored. (The LONG DIAG command works only in the proof editor.) Also, any extra LONGs are ignored.

Adding Text

To add a character, simply type it; it will be added to the left of the current position. This also includes the newline character; to start a new line, press carriage return.[5]

To *instantiate* a def, use ↑I. Nuprl will prompt for the name of the def in the command window with `I>`; type the name of the def and press RETURN. Nuprl will insert an instance of the def to the left of the current position and move the tree cursor to the first parameter.

Deleting and Killing Text

There are two ways to remove a character. The KILL key (↑K) removes the one at the current position, while the DELETE key removes the one to the left of the current position, assuming it is in the same subtree as the current position. (If it isn't DELETE does nothing.) To remove a def ref, use KILL. The cursor must point to the beginning of the reference. A *selection* of text may also be killed. This involves three steps:

1. With the mouse or the SEL key choose the left or right endpoint of the selection.
2. Select the other endpoint. It must be in the same window as the first. Also, if the first endpoint was within an argument to a def, the second must be within the same argument.
3. Press KILL; Nuprl will delete the two endpoints and everything between.

The killed text is moved to the *kill buffer* and can be used by the COPY command (see below). Only killed text goes into the kill buffer; text deleted by DELETE does not.

There can be at most two selected endpoints at any time. If another endpoint is selected the oldest one is forgotten. When a KILL or COPY is done *both* endpoints are forgotten. The two selected endpoints must be in the same window; if they are not KILL and COPY will give error messages.

Copying and Moving Text

To copy text from one location to another, select it, move the cursor to the target location, and press the COPY key, ↑C. If two endpoints were specified the selected text, including the endpoints, is copied to the left of the current position. If only one endpoint was specified the selected character or def ref is copied to the left of the current position. In either case all endpoints are forgotten.

[5] Carriage returns are significant only as delimiters.

To move text, use KILL and COPY together; if COPY is pressed when no selections have been made the kill buffer is copied to the left of the current position. Therefore, to move some text, select it, KILL it, move the cursor to the target position, and press COPY. The kill buffer is not affected by a COPY, so the easiest way to replicate a piece of text in many places is to type it, KILL it and then use COPY many times.

7.5 The Proof Editor

Nuprl proofs are sequent proofs organized in a refinement style. In other words they are trees where each node contains a *goal* and a *rule application*. The children of a node are the *subgoals* of their parent; in order to prove the parent it is sufficient to prove the children. (Nuprl automatically generates the children, given their parent and the rule to be applied; it is never necessary to type them in.) Not all goals have subgoals; for example, a goal proved by `hyp` does not. Also, an *incomplete* goal (one whose rule is bad or missing) has no subgoals.

In what follows the *main goal* (*the* goal for short) is the goal of the current node; *a* goal is either the main goal or one of its subgoals.

Basics of the Proof Editor

The proof editor, *red* (for *refinement editor*), is invoked when the `view` command is given on a theorem object. The proof editor window always displays the *current node* of the proof. When the editor is first entered the current node is the root of the tree; in general, the current node can be changed by going into one of its subgoals or by going to its parent. For convenience there are also ways to jump to the next unproved leaf and to the root of the theorem.

Figure 7.1 shows a sample proof editor window. The parenthesized numbers it contains are referenced below.

1. The header line names the theorem being edited, `t` in this case.
2. The *map*, `top 1 2`, gives the path from the root to the current node. (If the path is too long to fit on one line it is truncated on the left.) The displayed node is the second child of the first child of the root.
3. The goal's first (and only) hypothesis, `a:int`, appears under the map.
4. The conclusion of the goal is the formula to be proved, given the hypotheses. The conclusion is shown to the right of the the sequent arrow, `>>`. In the example it is

```
b:int # ((a=b in int)|((a=b in int) -> void)) in U1.
```

```
    ,-------------------------------------------------------,
(1) |EDIT THM t                                             |
    |-------------------------------------------------------|
(2) |* top 1 2                                              |
(3) |1. a:int                                               |
(4) |>> b:int # ((a=b in int) | ((a=b in int) -> void)) in U1|
    |                                                       |
(5) |BY intro                                               |
    |                                                       |
(6) |1* 1. a:int                                            |
    |   >> int  in int                                      |
    |                                                       |
(7) |2* 1. a:int                                            |
    |   2. b:int                                            |
    |   >> ((a=b in int) | ((a=b in int) -> void))  in U1    |
    '-------------------------------------------------------'
```

Figure 7.1: A Sample Proof Editor Window

5. The *rule* specifies how to refine the goal. In the example the `intro` rule is used to specify *product introduction*, since the main connective of the conclusion is the product operator, `#`.

6. The first subgoal has one hypothesis, `a:int`, and the conclusion `int in U1`.

7. The second subgoal has two hypotheses, `a:int` and `b:int`, and the conclusion

```
((a=b in int) | ((a=b in int) -> void)) in U1.
```

Notice that lines (2), (6) and (7) have asterisks, which serve as *status indicators*. The asterisk in line (2) means that this entire subgoal has the status *complete*; similarly, the asterisks in lines (6) and (7) mean that the two subgoals also are *complete*. Other status indicators are `#` (incomplete), `-` (bad), and `?` (raw).

Motion within a Proof Node

We first make the following definitions. The proof cursor is *in* a goal if it points to a hypothesis or the conclusion; it is *at* a goal if it points to the first part of the goal. If the cursor is not in a goal, it is *at* the rule of the current node or *in* or *at* a subgoal.

The cursor can be moved around within the current proof node in the following ways:

↑, ↓
: Moves the cursor up or down to the next or previous minor stopping point. A minor stopping point is an assumption, a conclusion or the rule.

LONG ↑, LONG ↓
: Moves the cursor up or down to the next or previous major stopping point. A major stopping point is a subgoal, a rule or the goal.

LONG LONG ↑
: Moves the cursor to the goal.

LONG LONG ↓
: Moves the cursor to the conclusion of the last subgoal.

Motion within the Proof Tree

The following commands allow one to move around within the proof. They change the current node by moving into or out of subproofs.

→
: Only has an effect when the cursor is in a subgoal; the node for that subgoal becomes the current node.

←
: Sets the current node to be the parent of the current node. The cursor will still point to the current goal, but the current goal will be displayed as one of the subgoals of the parent.

DIAG
: If the cursor isn't in the goal DIAG moves it to point to the goal. If it *is* in the goal, the parent of the current node will become the new current node, (as in ←) and the cursor will point to its goal (unlike ←).

LONG DIAG
: Has the same effect as four DIAGs.

LONG LONG DIAG
: Sets the current node to be the root of the proof. The cursor will point to the environment of its goal.

LONG →
: Sets the current node to be the next unproved node of the proof tree.

The search is done in preorder and wraps around the tree if necessary. If the proof has no incomplete nodes the current node is set to be the root.

Any other combination of LONG and the arrow keys is ignored. Extra LONGs are ignored.

Initiating a Refinement

To refine a goal, a refinement rule must be supplied. To do this, first move the cursor to point to the rule of the current node and press SEL; this will bring up an instance of the text editor in a window labeled `EDIT rule of` *thmname*. Type the rule into this window and press ↑D. Nuprl will parse the rule and try to apply it to the goal of the current proof tree node. If it succeeds the proof window will be redrawn with new statuses and subgoals as necessary. If it fails then one of two things may happen. If the error is severe, the status of the node (and the proof) will be set to *bad*, an error message will appear in the command/status window, and the rule will be set to `??bad refinement rule??`. If the error is mild and due to a missing input, Nuprl will display a `HELP` message in the command/status window and leave the rule window on the screen so that it can be fixed.

If an existing rule is selected for editing Nuprl will copy it to the edit window. If the text of the rule is changed, then when ↑D is pressed Nuprl will again parse and refine the current goal. When it does it will throw away any proofs for the children of the current node, even if they could have been applied to the new children of the current node. On the other hand, if the text is not changed, then when ↑D is pressed Nuprl will not reparse the rule.

Showing Goals, Subgoals and Rules

In some circumstances it is convenient to be able to copy goals or subgoals into other proofs. This can't be done directly, but there is a roundabout way. If SEL is pressed while the cursor is in a goal Nuprl creates a read–only text window containing the text of the goal. The window will be labeled `SHOW goal of` *thmname*. The copy mechanism of the text editor can then be used to copy portions of the goal to other windows.

There is a second way to `SHOW` a goal or subgoal: move the cursor to it and press ↑W. Nuprl will display the goal or subgoal with all definitions expanded and all terms fully parenthesized. This also works for rules.

Invoking Transformation Tactics

Transformation tactics are tactics which act as proof transformers rather than as generalized refinement rules. To invoke a transformation tactic,

press ↑T (↑A in some older versions). This will bring up a text edit window labelled `Transformation Tactic`. Type the name of the tactic and any arguments into the window and press ↑D; Nuprl will call the tactic and redisplay the proof window to show any effects. If the tactic returns an error message it will be displayed in the command/status window.

7.6 Definitions and Definition Objects

In Nuprl text is used to represent proof rules, theorem goals and and ML tactic bodies. This text is made up of characters and calls to text macros. Since internally pieces of text are stored as trees of characters and macro references, they are also called *text trees* or *T-trees*.

Text macros are used mainly to improve the readability of terms and formulas. For example, since there is no $\geq$ relation built into Nuprl, a formula like $x \geq y$ might be represented by `((x<y) -> void)`. Instead of always doing this translation one may define

```
<x>>=<y>==((<x><<y>) -> void)
```

and then use `x>=y` instead of `((x<y) -> void)`, thereby effectively adding $\geq$ as a new relation. (Note that for the definition to be used it must be *instantiated* (see section 4 above) — one cannot just type "`x>=y`".) Since most macros are used to extend the notation available for terms and formulas, they are also known as *definitions*, or *defs*, and the library objects that contain them are called *definition objects*.

The left–hand side of a definition specifies its formal parameters and also shows how to display a call to it. The right–hand side shows what the definition stands for. In the example above the two formal parameters are `<x>` and `<y>`. (Formal parameters are always specified between angle brackets.)

The left–hand side of definitions should contain zero or more formal parameters plus whatever other characters one wants displayed. For instance,

```
<x> greater than or equal to <y>
(ge <x> <y>)
<x>>=<y>
```

are all perfectly valid left–hand sides. A formal parameter is an *identifier* enclosed in angle brackets or an *identifier*:*description* enclosed in angle brackets, where a *description* is a piece of text that does not include an unescaped `>`. (The escape mechanism is explained below.) After the def object has been checked the description is used to help display the object in the library window, as <*description*> is inserted in place of the formal parameter. For example, given the following def of `GE`,

```
<x:int1>>=<y:int2>==((<x><<y>) -> void)
```

the library window would show the following:

```
,-------------------------------------------------------,
| Library                                               |
|-------------------------------------------------------|
| * GE                                                  |
|   DEF: <int1>>=<int2>                                 |
|                                                       |
'-------------------------------------------------------'
```

If the description is omitted or void, the empty string is used, so that formal parameter is displayed as "<>".

Sometimes one wishes to use > and > as something other than angle brackets on the left–hand side of definitions. Backslash is used to suppress the normal meanings of < and >. To keep a < from being part of a formal parameter or to make a > part of a description, precede it with a backslash, as in

```
<x>\<=<y>==((<y><<x>) -> void)
```

and

```
square root(<x: any int\>0 >)== ... .
```

To get a backslash on the left–hand side, escape it with another backslash:

```
<x>\\<y>== ... .
```

The right–hand side of a definition shows what the definition stands for. Besides arbitrary characters it can contain formal references to the parameters (identifiers enclosed in angle brackets) and uses of checked defs. (Note: defs may not refer to each other recursively.) Every formal reference must be to a parameter defined on the left–hand side; no nonlocal references are allowed.

Here is an example of a def that uses another def:

```
<x>>=<y>>=<z>==<x>>=<y> & <y>>=<z>.
```

An example of a use of this def is `3>=2>=1`, which expands ultimately into `((3<2) -> void) & ((2<1) -> void)`. Blanks are *very* significant on the right–hand sides of defs; the three characters `<x>` are considered to be a formal reference, but the four characters `<x >` are not.

Extra backslashes on the right–hand side do not affect parsing, although they do affect the readability of the generated text.[6] For example, the (unnecessary) backslash in

[6] Unlike, for example, the Unix program nroff Nuprl does not strip off backslashes when evaluating macros. They are stripped off only when text is actually parsed.

```
<x>>=<y>==((<x>\<<y>) -> void)
```

is displayed whenever the generated text is displayed. For better readability it is best to avoid backslashes on the right–hand side whenever possible.

The routine that parses the right–hand side of defs does backtracking to reduce the number of backslashes needed on right–hand sides. The only time a backslash is absolutely required is when there is a < followed by what looks like an identifier followed by a >, all with no intervening spaces. For example, the right–hand side below requires its backslash, since <x> looks like a parameter reference:

```
... == <f1>, 0\<x>> <f2>.
```

If the def were rewritten as

```
... == <f1>, 0<x >> <f2>
```

then it would no longer need a backslash.

Notice that the right–hand side does not have to refer to all of the formal parameters and can even be empty, as in the "comment" def.

```
(* <string:comment> *)== .
```

A use of such a def might be the following.

```
,-------------------------------------------------------,
| EDIT main goal of thm                                 |
|-------------------------------------------------------|
| >> x:int # ( ... )      (* This text is ignored *)     |
|                                                       |
|                                                       |
'-------------------------------------------------------'
```

Chapter 8

The Rules

8.1 Semantics

The Nuprl semantics is a variation on that given by Martin-Löf for his type theory [Martin-Löf 82]. There are three stages in the semantic specification: the *computation system*, the *type system* and the so–called *judgement forms*. Here is how we shall proceed. We shall specify a computation system consisting of *terms*, divided into *canonical* and *noncanonical*, and a procedure for *evaluating* terms which for a given term t returns at most one canonical term, called the *value* of t. In Nuprl whether a term is canonical depends only on the outermost form of the term, and there are terms which have no value. We shall write $t \leftarrow s$ to mean that s has value t.

Next we shall specify a system of types. A *type* is a term T (of the computation system) with which is associated a transitive, symmetric relation, $t = s \in T$, which respects evaluation in t and s; that is, if T is a type and $t' \leftarrow t$ and $s' \leftarrow s$, then $t' = s' \in T$ if and only if $t = s \in T$. We shall sometimes say "T type" to mean that T is a type. We say t is a *member* of T, or $t \in T$, if $t = t \in T$. Note that $t = s \in T$ is an equivalence relation (in t and s) when restricted to members of T.[1] Actually, $t = s \in T$ is a three–place relation on terms which respects evaluation in all three places. We also use a transitive, symmetric relation on terms, $T = S$, called *type equality*, which $t = s \in T$ respects in T; that is, if $T = S$ then $t = s \in T$ if and only if $t = s \in S$. The relation $T = S$ respects evaluation in T and S, and T is a type if and only if $T = T$. The restriction of $T = S$ to types is an equivalence relation.

For our purposes, then, a type system for a given computation system consists of a two–place relation $T = S$ and a three–place relation $t = s \in T$

[1] There is a similarity between a type and Bishop's notion of *set*. Bishop [Bishop 67] says that to give a set, one gives a way to construct its members and gives an equivalence relation, called the *equality* on that set, on the members.

on terms such that

$T = S$ is transitive and symmetric;
$T = S$ if and only if $\exists T'.\ T' \leftarrow T\ \&\ T' = S$;
$t = s \in T$ is transitive and symmetric in t and s;
$t = s \in T$ if and only if $\exists t'.\ t' \leftarrow t\ \&\ t' = s \in T$;
$T = T$ if $t = s \in T$;
$t = s \in T$ if $T = S\ \&\ t = s \in S$.

We define "T type" and "$t \in T$" by

T type if and only if $T = T$;
$t \in T$ if and only if $t = t \in T$.

Finally, so–called judgements will be explained. This requires consideration of terms with free variables because substitution of closed terms for free variables is central to judgements as presented here. In the description of semantics given so far "term" has meant a closed term, i.e., a term with no free variables. There is only one form of judgement in Nuprl, $x_1 : T_1, \ldots, x_n : T_n$ `>>` S `[ext` s`]`, which in the case that n is 0 means $s \in S$. The explanation of the cases in which n is not 0 must wait.

Substitution

For the purposes of giving the procedure for evaluation and explaining the semantics of judgements, we would only need to consider substitution of closed terms for free variables and hence would not need to consider simultaneous substitution or capture. However, for the purpose of specifying inference rules later we shall want to have simultaneous substitution of terms with free variables for free variables. The result of such a substitution is indicated thus:

$$t[t_1, \ldots, t_n / x_1, \ldots, x_n]$$

where $0 \leq n$, $x_1, \ldots, x_n$ are variables (not necessarily distinct) and $t_1, \ldots, t_n$ are terms. It is handy to permit multiple occurrences of the same variable among the targets for replacements, all but the last of which are ignored. $t[t_1, \ldots, t_n / x_1, \ldots, x_n]$ is the result of replacing each free occurrence of x_i in t by s_i for $1 \leq i \leq n$, where s_i is t_j with j the greatest k such that x_i is x_k.

The Computation System

Figure 8.1 shows the terms of Nuprl. Variables are terms, although since they are not closed they are not executable. Variables are written as identifiers, with distinct identifiers indicating distinct variables.[2] Nonnegative integers

[2] An identifier is any string of letters, digits, underscores or at–signs that starts with a letter. The only identifiers which cannot be used for variables are `term_of` and those which serve as operator names, such as `int` or `spread`.

are written in standard decimal notation. There is no way to write a negative integer in Nuprl; the best one can do is to write a noncanonical term, such as `-5`, which evaluates to a negative integer. Atom constants are written as character strings enclosed in double quotes, with distinct strings indicating distinct atom constants.

The free occurrences of a variable x in a term t are the occurrences of x which either are t or are free in the immediate subterms of t, excepting those occurrences of x which become *bound* in t. In figure 8.1 the variables written below the terms indicate which variable occurrences become bound; some examples are explained below.

- In x:A`#`B the x in front of the colon becomes bound and any free occurrences of x in B become bound. The free occurrences of variables in x:A`#`B are all the free occurrences of variables in A and all the free occurrences of variables in B except for x.

- In <a`,`b> no variable occurrences become bound; hence, the free occurrences of variables in <a`,`b> are those of a and those of b.

- In `spread(`s`;`x`,`y`.`t`)` the x and y in front of the dot and any free occurrences of x or y in t become bound.

Parentheses may be used freely around terms and often must be used to resolve ambiguous notations correctly. Figure 8.2 gives the relative precedences and associativities of Nuprl operators.

The closed terms above the dotted line in figure 8.1 are the canonical terms, while the closed terms below it are the noncanonical terms. Note that carets appear below most of the noncanonical forms; these indicate the *principal argument places* of those terms. This notion is used in the evaluation procedure below. Certain terms are designated as *redices*, and each redex has a unique *contractum*. Figure 8.3 shows all redices and their contracta.

The evaluation procedure is as follows. Given a (closed) term t,

If t is canonical then the procedure terminates with result t.

Otherwise, execute the evaluation procedure on each principal argument of t, and if each has a value, replace the principal arguments of t by their respective values; call this term s.

If s is a redex then the procedure for evaluating t is continued by evaluating the contractum of s.

If s is not a redex then the procedure is terminated without result; t has no value.

x	n	i	`void`
`int`	`atom`	`axiom`	`nil`
`U`k	`inl(`a`)`	`inr(`a`)`	A `list`
`\`x`.`b	a`<`b	`<`a`,`b`>`	a`.`b
A`#`B	x`:`A`#`B	A`->`B	x`:`A`->`B
A`|`B	A`//`B	x`,`y`:`A`//`B	{A`|`B}
{x`:`A`|`B}	a `=` b `in` A		

canonical if closed

noncanonical if closed

`-`a	`any(`a`)`	t`(`a`)`	a`+`b
a`-`b	a`*`b	a`/`b	a `mod` b

`spread(`a`;`x`,`y`.`t`)`

`decide(`a`;`x`.`s`;`y`.`t`)`

`list_ind(`a`;`s`;`x`,`y`,`u`.`t`)`

`ind(`a`;`x`,`y`.`s`;`b`;`u`,`v`.`t`)`

`atom_eq(`a`;`b`;`s`;`t`)`

`int_eq(`a`;`b`;`s`;`t`)`

`less(`a`;`b`;`s`;`t`)`

x, y, u, v	range over variables.
a, b, s, t, A, B	range over terms.
n	ranges over integers.
k	ranges over positive integers.
i	ranges over atom constants.

Variables written below a term indicate where the variables become bound.
"^" indicates principal arguments.

Figure 8.1: Terms

Lower Precedence	
`=,in`	left associative
`#,->,|,//`	right associative
`<` (as in a`<`b)	left associative
`+,-` (infix)	left associative
`*,/,mod`	left associative
`inl,inr,-` (prefix)	—
`.` (as in a`.`b)	right associative
`\x.`	—
`list`	—
`(`a`)` (as in t`(`a`)`)	—
Higher Precedence	

Figure 8.2: Operator Precedence

The Type System

For convenience we shall extend the relation $t \leftarrow s$ to possibly open terms. If s or t contain free variables then $t \leftarrow s$ does not hold; otherwise, it is true if and only if s has value t. Also, define $t \leftarrow s, r$ to mean that $t \leftarrow s$ and $t \leftarrow r$.

Recall that the members of a type are its canonical members and the terms which have those members as values. The integers are the canonical members of the type `int`. The denumerably many atom constants are the canonical members of the type `atom`. The type `void` is empty. The type A`|`B is like a disjoint union of types A and B. The terms `inl(`a`)` and `inr(`b`)` are canonical members so long as $a \in A$ and $b \in B$; a and b need not be canonical. (The operator names `inl` and `inr` are mnemonic for "inject left" and "inject right".) The canonical members of x`:`A`#`B are the terms `<`a`,`b`>` with $a \in A$ and $b \in B[a/x]$, a and b not necessarily canonical. Note that the type from which the second component is selected may depend on the value of the first component. The canonical members of the type A `list` represent lists of members of A. The empty list is represented by `nil`. A nonempty list with head a is represented by a`.`b, where b evaluates to a member of the type A `list` and represents the tail.

A term of the form t`(`a`)` is called an *application* of t to a, and a is called its *argument*. The members of x`:`A`->`B are called *functions*, and each canonical member is a *lambda term*, `\`x`.`b, whose application to any $a \in A$

Redex	**Contractum**
`(\`$x.b$`)(`a`)`	$b[a/x]$
`spread(<`a`,`b`>;`x`,`$y.t$`)`	$t[a,b/x,y]$
`decide(inl (`a`) ;`$x.s$`;`$y.t$`)`	$s[a/x]$
`decide(inr (`b`) ;`$x.s$`;`$y.t$`)`	$t[b/y]$
`list_ind(nil;`s`;`x`,`y`,`$u.t$`)`	s
`list_ind(`$a.b$`;`s`;`x`,`y`,`$u.t$`)`	$t[a,b,$`list_ind(`b`;`s`;`x`,`y`,`$u.t$`)`$/x,y,u]$
`atom_eq(`i`;`j`;`s`;`t`)`	s if i is j; t otherwise
`int_eq(`m`;`n`;`s`;`t`)`	s if m is n; t otherwise
`less(`m`;`n`;`s`;`t`)`	s if m is less than n; t otherwise
`-`n	the negation of n
m`+`n	the sum of m and n
m`-`n	the difference
m`*`n	the product
m/n	`0` if n is `0`; otherwise, the floor of the obvious rational.
m `mod` n	`0` if n is `0`; otherwise, the positive integer nearest `0` that differs from m by a multiple of n.
`ind(`m`;`x`,`$y.s$`;`b`;`u`,`$v.t$`)`	b if m is `0`; $t[m,$`ind(`$m-1$`;`x`,`$y.s$`;`b`;`u`,`$v.t$`)`$/u,v]$ if m is positive; $s[m,$`ind(`$m+1$`;`x`,`$y.s$`;`b`;`u`,`$v.t$`)`$/x,y]$ if m is negative.

a, b, s, t	range over terms.
x, y, u, v	range over variables.
m, n	range over integers.
i, j	range over atom constants.

Figure 8.3: Redices and Contracta

is a member of $B[a/x]$. It is required that applications to equal members of type A be equal. Clearly, $t(a) \in B[a/x]$ if $t \in x{:}A\texttt{->}B$ and $a \in A$.

The significance of some constructors derives from the representation of propositions as types. A proposition represented by a type is true if and only if the type is inhabited. The type $a\texttt{<}b$ is inhabited if and only if the value of a is less than the value of b. The type $(a\texttt{=}b\ \texttt{in}\ A)$ is inhabited if and only if $a = b \in A$. Obviously, the type $(a\texttt{=}a\ \texttt{in}\ A)$ is inhabited if and only if $a \in A$, so "a `in` A" has been adopted as a notation for this type. The members of $\{x{:}A \mid B\}$ are the members a of A such that $B[a/x]$ is inhabited. Types of the form $\{x{:}A \mid B\}$ are called *set types*. The set constructor provides a device for specifying subtypes; for example, {`x:int|0<x`} has just the positive integers as canonical members.

The members of $x\texttt{,}y{:}A\texttt{//}B$ are the members of A. The difference between this type and A is equality. $a = a' \in x\texttt{,}y{:}A\texttt{//}B$ if and only if a and a' are members of A and $B[a, a'/x, y]$ is inhabited. Types of this form are called *quotient types*. The relation $\exists b.\, b \in B[a, a'/x, y]$ is an equivalence relation over A in a and a'; this is built into the criteria for $x\texttt{,}y{:}A\texttt{//}B$ being a type.

Now consider equality on the other types already discussed. (Recall that terms are equal in a given type if and only if they evaluate to canonical terms equal in that type. Recall also that $a = a' \in A$ is an equivalence relation in a and a' when restricted to members of A.) Members of `int` are equal (in `int`) if and only if they have the same value. The same goes for type `atom`. Canonical members of $A \mid B$ ($x{:}A\texttt{\#}B$, A `list`) are equal if and only if they have the same outermost operator and their corresponding immediate subterms are equal (in the corresponding types). Members of $x{:}A\texttt{->}B$ are equal if and only if their applications to any member a of A are equal in $B[a/x]$. We say equality on $x{:}A\texttt{->}B$ is *extensional*. The types $a\texttt{<}b$ and $(a\texttt{=}b\ \texttt{in}\ A)$ have at most one canonical member, `axiom`. Equality in $\{x{:}A \mid B\}$ is just the restriction of equality in A to $\{x{:}A \mid B\}$.

We must now consider the notion of *functionality*. A term B is *type–functional in* $x{:}A$ if and only if A is a type and $B[a/x] = B[a'/x]$ for any a and a' such that $a = a' \in A$. A term b is *B–functional in* $x{:}A$ if and only if B is type–functional in $x{:}A$ and $b[a/x] = b[a'/x] \in B[a/x]$ for any a and a' such that $a = a' \in A$. There are restrictions on type formation involving type–functionality. These can be seen in the type formation clauses of section 8.2 for $x{:}A\texttt{\#}B$, $x{:}A\texttt{->}B$, and $\{x{:}A \mid B\}$. In each of these B must be type–functional in $x{:}A$.[3] We may now say that the members of $x{:}A\texttt{->}B$ are the lambda terms $\backslash x.\, b$ such that b is B–functional in $x{:}A$. In the type $x\texttt{,}y{:}A\texttt{//}B$, that B must be type–functional in both x,$y{:}A$ follows from the fact that $x{:}A\texttt{->}y{:}A\texttt{->}B\texttt{->}B$ must be a type. There are also constraints on the typehood of $x\texttt{,}y{:}A\texttt{//}B$ which guarantee that the relation $\exists b.\, b \in$

[3] In the formation of these as members of U_k, B must be U_k–functional in $x{:}A$.

$B[a, a'/x, y]$ is an equivalence relation on members of A and respects equality in A. It should be noted that if A is empty then every term is type–functional in its free variables over A. Hence, x:`void#3` is a type (with no members) even though `3` is not a type.

Equal types have the same membership and equality, but not conversely. Type equality in Nuprl is not extensional; that is, it is not enough for type equality that two types should have the same membership and equality. In Nuprl equal canonical types always have the same outermost type constructor.[4] The relations that must hold between the respective immediate subterms are seen easily enough in the definition of type equality given in section 8.2 on page 143. It should be noted that in contrast to equality between types of the form x:`A#B` or x:`A->B`, much less is required for $\{x{:}A \mid B\}$=$\{x{:}A \mid B'\}$ than type–functional equality of B and B' in x:A. All that is required is the existence of functions which for each $a \in A$ evaluate to functions mapping back and forth between $B[a/x]$ and $B'[a/x]$. Equality between quotient types is defined similarly. If x does not occur free in B then `A#B`=x:`A#B`, `A->B`=x:`A->B`, and $\{A \mid B\}$=$\{x{:}A \mid B\}$; if x and y do not occur free in B then `A//B`=x`,`y:`A//B`. As a result there is no need for clauses in the type system description giving the criteria for $t = t' \in$ `A#B` and the others explicitly.

Now consider the so–called *universes*, U_k (k positive). The members of U_k are types. The universes are cumulative; that is, if j is less than k then membership and equality in U_j are just restrictions of membership and equality in U_k. Universe U_k is closed under all the type–forming operations except formation of U_i for i greater than or equal to k. Equality (hence membership) in U_k is similar to type equality as defined previously except that equality (membership) *in* U_k is required wherever type equality (typehood) was formerly required, and although all universes are types, only those `U`i such that i is less than k are in U_k. Equality in U_k is the restriction of type equality to members of U_k.

So far the only noncanonical form explicitly mentioned in connection with the type system is application. We shall elaborate here on a couple of forms, and it should then be easy to see how to treat the others. The `spread` form is used for computational analysis of pairs. The pair of components is *spread* apart so that the components can be used separately.

`spread(`e`;`x`,`y`.`t`)` $\in T[e/z]$ if
$e \in x$:`A#B`
& T is type functional in z:(x:`A#B`)
& $\forall a, b.\ t[a, b/x, y] \in T$[`<`$a$`,`$b$`>`$/z$]
if $a \in A$ and $b \in B[a/x]$

[4] modulo the difference between `A#B` and x:`A#B`, etc.

Since $e \in x{:}A\#B$, then for some a and b $\langle a,b\rangle \leftarrow e$ where $a \in A$, and $b \in B[a/x]$. Hence $\mathtt{spread}(e;x,y.t)$ and $t[a,b/x,y]$ have the same value, so it is enough that $t[a,b/x,y] \in T[e/z]$. But from our hypotheses it follows that $t[a,b/x,y] \in T[\langle a,b\rangle/z]$, so it is enough that $T[e/z] = T[\langle a,b\rangle/z]$. Now $e = \langle a,b\rangle \in x{:}A\#B$ since $e \in x{:}A\#B$ and equality respects evaluation; therefore $T[e/z] = T[\langle a,b\rangle/z]$ in light of T's functionality in $z{:}(x{:}A\#B)$.[5]

The list induction form allows one to perform inductive analysis of lists. $\mathtt{list\_ind}(e;s;x,y.t)$ is a function (in e) which halts on all members of A `list`. It is the function (in e) defined by primitive recursion, where s is the result for the base case of $e =$`nil` (in A `list`) and t shows how to build the value for $e = x.y$ (in A `list`) from x, y and the value of the function being defined on y, this value being passed through u during evaluation.

$\mathtt{list\_ind}(e;s;x,y,u.t) \in T[e/z]$ if
$e \in A$ `list`
& T is type functional in $z{:}(A$ `list`$)$
& $s \in T[\mathtt{nil}/z]$
& $\forall a,b,c.\ t[a,b,c/x,y,u] \in T[a.b/z]$
if $a \in A$ & $b \in A$ `list` & $c \in T[b/z]$.

Let us prove this by induction on the length of the list represented by e, all other variables universally quantified. Suppose $\mathtt{nil} \leftarrow e$. By definition we know that $\mathtt{list\_ind}(e;s;x,y,u.t)$ and s have the same value, so it is enough for the base case that $s \in T[e/z]$; this is true since T is functional in $z{:}A$ `list`, $\mathtt{nil} = e \in A$ `list`, and $s \in T[\mathtt{nil}/z]$. Now suppose that for some a and b, $a.b \leftarrow e$, $a \in A$, and $b \in A$ `list`. Now

$$\mathtt{list\_ind}(e;s;x,y,u.t)$$

and

$$t[a,b,\mathtt{list\_ind}(\text{b;s;x,y,u.t})/\text{x,y,u}$$

have the same value, so it is enough that the substitution into t is in $T[e/z]$. $b \in A$ `list` and b represents a shorter list than e; therefore, by inductive hypothesis

$$\mathtt{list\_ind}(b;s;x,y,u.t) \in T[b/z].$$

It follows that the substitution into t is in $T[a.b/z]$, so it is enough that $T[a.b/z] = T[e/z]$; this holds because of T's functionality and the equality of $a.b$ and e (in A `list`).

The `decide` form is used to discern a left from a right injection, and to permit computation on the injected term. The `ind` form is used to define

[5] If the reader finds this a bit difficult to follow, perhaps it would help to consider first the case in which x is not free in B and z is not free in T and then the more general cases afterward.

functions recursively on integers.[6] The reader is referred to chapter 2 or to the exposition of the rules for further elaboration of the use of noncanonical forms.

Judgements

The significance of the so–called judgements lies in the fact that they constitute the claims of a Nuprl proof. They are the units of assertion; they are the objects of inference. The judgements of Nuprl have the form

$$x_1{:}T_1\,,\ldots,x_n{:}T_n \texttt{ >> } S \texttt{ [ext } s\texttt{]}$$

where $x_1,\ldots,x_n$ are distinct variables and $T_1,\ldots,T_n,S,s$ are terms (n may be 0), every free variable of T_i is one of $x_1,\ldots,x_{i-1}$, and every free variable of S or of s is one of $x_1,\ldots,x_n$. The list $x_1{:}T_1\,,\ldots,x_n{:}T_n$ is called the *hypothesis list* or *assumption list*, each $x_i{:}T_i$ is called a *declaration* (of x_i), each T_i is called a *hypothesis* or *assumption*, S is called the *consequent* or *conclusion*, s the *extract term* (the reason will be seen later), and the whole thing is called a *sequent*.

Before explaining the conditions which make a Nuprl sequent true we shall define a relation $H@l$, where H is a hypothesis list and l is a list of terms, and we shall define what it is for a sequent to be *true at* a list of terms.

$x_1{:}T_1\,,\ldots,x_n{:}T_n$ @ $t_1,\ldots,t_n$ if and only if
$\forall j < n.\ t_{j+1} \in T_{j+1}[t_1,\ldots,t_j/x_1,\ldots,x_j]$
& $\forall t'_1,\ldots,t'_j.\ T_{j+1}[t_1,\ldots,t_j/x_1,\ldots,x_j] =$
$T_{j+1}[t'_1,\ldots,t'_j/x_1,\ldots,x_j]$
if $\forall i < j.\ t_{i+1} = t'_{i+1} \in T_{i+1}[t_1,\ldots,t_i/x_1,\ldots,x_i]$

We can also express this relation by saying that every t_i is a member of

$$T_i[t_1,\ldots,t_{i-1}/x_1,\ldots,x_{i-1}]$$

and every T_i is type–functional in $x_1{:}\hat{T}_1,\ldots,x_{i-1}{:}\hat{T}_{i-1}$, where $\hat{T}_j$ means the set type

$$\{x_j{:}T_j[t_1,\ldots,t_{j-1}/x_1,\ldots,x_{j-1}] | x_j = t_j \texttt{ in } T_j[t_1,\ldots,t_{j-1}/x_1,\ldots,x_{j-1}]\}.$$

The sequent

$x_1{:}T_1\,,\ldots,x_n{:}T_n$ >> S [ext s]

[6] This is defined by primitive recursion with `0` as the base case and an induction step for each direction out (positive and negative). Of course, in a higher–order system such as Nuprl, where functions can take functions as arguments and have functions as values, one can define by primitive recursion functions in `int->int` which are not primitive recursive.

is true at $t_1, \ldots, t_n$ if and only if

$$\begin{array}{l} \forall t'_1, \ldots, t'_n.\ (\ S[t_1, \ldots, t_n/x_1, \ldots, x_n] = S[t'_1, \ldots, t'_n/x_1, \ldots, x_n] \\ \quad \& \ s[t_1, \ldots, t_n/x_1, \ldots, x_n] = \\ \quad\quad s[t'_1, \ldots, t'_n/x_1, \ldots, x_n] \in S[t_1, \ldots, t_n/x_1, \ldots, x_n]\ ) \\ \quad \text{if } x_1{:}T_1, \ldots, x_n{:}T_n \ @\ t_1, \ldots, t_n \\ \quad\quad \& \ \forall i < n.\ t_{i+1} = t'_{i+1} \in T_{i+1}[t_1, \ldots, t_i/x_1, \ldots, x_i] \end{array}$$

Equivalently, we can say that s is S–functional in $x_1{:}\hat{T}_1, \ldots, x_n{:}\hat{T}_n$ if $x_1{:}T_1, \ldots, x_n{:}T_n \ @\ t_1, \ldots, t_n$. The sequent

$x_1{:}T_1, \ldots, x_n{:}T_n$ `>>` S `[ext` s`]` is true if and only if
$\forall t_1, \ldots, t_n.\ x_1{:}T_1, \ldots, x_n{:}T_n$ `>>` S `[ext` s`]` is true at $t_1, \ldots, t_n$

The connection between functionality and the truth of sequents lies in the fact that S is type–functional (or s is S–functional) in $x{:}T$ if and only if T is a type and for each member t of T, S is type–functional (s is S–functional) in $x{:}\{x{:}T \mid\ x{=}t$ `in` $T\}$. Therefore, s is S–functional in $x{:}T$ if and only if T is a type and the sequent $x{:}T$ `>>` S `[ext` s`]` is true.[7]

It is not always necessary to declare a variable with every hypothesis in a hypothesis list. If a declared variable does not occur free in the conclusion, extract term or any hypothesis, then the variable (and the colon following it) may be omitted.

In Nuprl it is not possible for the user to enter a complete sequent directly; the extract term must be omitted. In fact, a sequent is never displayed with its extract term. The system has been designed so that upon completion of a proof, the system automatically provides, or *extracts*, the extract term. This is because in what is anticipated to be the standard mode of use, the user tries to prove that a certain type is inhabited without regard to the identity of any member. One expects that in this mode the user thinks of the type (that is to be shown inhabited) as a proposition, and that it is merely the truth of this proposition that the user wants to show. When one does wish to show explicitly that $a = b \in A$ or that $a \in A$, one instead shows the type `(`a `=` b `in` A`)` or the type `(`a `in` A`)` to be inhabited.[8]

Also, the system can often extract a term from an incomplete proof when the extraction is independent of the extract terms of any unproven claims within the proof body. Of course, such unproven claims may still contribute to the truth of the proof's main claim. For example, it is possible to provide

[7] If we were to give a similar semantic account of the judgements of [Martin-Löf 82], the plainest difference between the truth conditions for those judgements and Nuprl judgements would be that the truth of the former requires type–functionality of each assumption in all its variables.

[8] Recall that the term `(`a `=` b `in` A`)` is a type whenever $a \in A$ and $b \in A$ and is inhabited just when $a = b \in A$. As a special case, the term `(`a `in` A`)` is a type and is inhabited just when $a \in A$.

an incomplete proof of the untrue sequent `>> 1<1 [ext axiom]`, the extract term `axiom` being provided automatically.

Although the term extracted from a proof of a sequent is not displayed in the sequent, the term is accessible by other means through the name assigned to the proof in the user's library. It should be noted that in the current system proofs named in the user's library cannot be proofs of sequents with hypotheses.

8.2 The Type System in Detail

This section may be skipped on the first reading, as the informal description of the type system given in section 8.1 should meet the needs of most readers.

This section consists of two parts. First, we give the formal definition of Nuprl's type system by means of a recursive definition of the necessary predicates. Then, for ease of understanding and reference, from the definition we extract criteria for typehood and membership. Those who do read this section may benefit by reading the latter part first.

Formal Definition of Equality

What follows is a recursive definition of four predicates— T type, $t \in T$, $T = S$ and $t = s \in T$—on possibly open terms. When restricted to closed terms these are just the predicates which constitute the Nuprl type system.[9]
In the clauses below,
$T, T', A, A', B, B', a, a', b, b', e, e', t, t', f, g, h$ range over possibly open terms,
$x, x', y, y', u, u', v, w$ range over variables;
k, j range over positive integers;
m, n range over integers;
i ranges over atom constants.

T type iff $T = T$

$t \in T$ iff $t = t \in T$

$T = T'$ iff $\texttt{void} \leftarrow T, T'$ or $\texttt{atom} \leftarrow T, T'$ or $\texttt{int} \leftarrow T, T'$

or $\exists A, A', a, a', b, b'.\ (a\texttt{=}b\ \texttt{in}\ A) \leftarrow T\ \&\ (a'\texttt{=}b'\ \texttt{in}\ A') \leftarrow T'$
$\&\ A = A'\ \&\ a = a' \in A\ \&\ b = b' \in A$

or $\exists a, a', b, b'.\ (a\texttt{<}b) \leftarrow T\ \&\ (a'\texttt{<}b') \leftarrow T'\ \&\ a = a' \in \texttt{int}\ \&\ b = b' \in \texttt{int}$

or $\exists A, A'.\ A\ \texttt{list} \leftarrow T\ \&\ A'\ \texttt{list} \leftarrow T'\ \&\ A = A'$

[9] It turns out that these predicates will hold only for closed terms.

or $\exists A, A', B, B'.\ A \mid B \leftarrow T\ \&\ A' \mid B' \leftarrow T'\ \&\ A = A'\ \&\ B = B'$

or $\exists A, A', x, x', B, B'.$
$(\ x : A\texttt{\#}B \leftarrow T$ or $A\texttt{\#}B \leftarrow T\ )$
$\&\ (\ x' : A'\texttt{\#}B' \leftarrow T'$ or $A'\texttt{\#}B' \leftarrow T'\ )$
$\&\ A = A'\ \&\ \forall a, a'.\ B[a/x] = B'[a'/x']$ if $a = a' \in A$

or $\exists A, A', x, x', B, B'.$
$(\ x : A\texttt{->}B \leftarrow T$ or $A\texttt{->}B \leftarrow T\ )$
$\&\ (\ x' : A'\texttt{->}B' \leftarrow T'$ or $A'\texttt{->}B' \leftarrow T'\ )$
$\&\ A = A'\ \&\ \forall a, a'.\ B[a/x] = B'[a'/x']$ if $a = a' \in A$

or $\exists A, A', x, x', B, B', e, e', u.$
$(\ \{x : A \mid B\} \leftarrow T$ or $\{A \mid B\} \leftarrow T\ )$
$\&\ (\ \{x' : A' \mid B'\} \leftarrow T'$ or $\{A' \mid B'\} \leftarrow T'\ )$
$\&\ A = A'\ \&\ u$ occurs in neither B nor B'
$\&\ e \in u : A\texttt{->}B[u/x]\texttt{->}B'[u/x']$
$\&\ e' \in u : A\texttt{->}B'[u/x']\texttt{->}B[u/x]$

or $\exists A, A', x, x', y, y', B, B', e, e', f, g, h, u, v, w.$
$(\ x\texttt{,}y : A\texttt{//}B \leftarrow T$ or $A\texttt{//}B \leftarrow T\ )$
$\&\ (\ x'\texttt{,}y' : A'\texttt{//}B' \leftarrow T'$ or $A'\texttt{//}B' \leftarrow T'\ )$
$\&\ A = A'$
$\&\ u, v, w$ are distinct and occur in neither B nor B'
$\&\ e \in u : A\texttt{->}v : A\texttt{->}B[u, v/x, y]\texttt{->}B'[u, v/x', y']$
$\&\ e' \in u : A\texttt{->}v : A\texttt{->}B'[u, v/x', y']\texttt{->}B[u, v/x, y]$
$\&\ f \in u : A\texttt{->}B[u, u/x, y]$
$\&\ g \in u : A\texttt{->}v : A\texttt{->}B[u, v/x, y]\texttt{->}B[v, u/x, y]$
$\&\ h \in u : A\texttt{->}v : A\texttt{->}w : A\texttt{->}B[u, v/x, y]\texttt{->}B[v, w/x, y]\texttt{->}$
$B[u, w/x, y]$

or $\exists k.\ \texttt{U}k \leftarrow T, T'$

$t = t' \in T$ if $T = T'\ \&\ t = t' \in T'$

not $t \in$ `void`

$t = t' \in$ `atom` iff $\exists i.\ i \leftarrow t, t'$

$t = t' \in$ `int` iff $\exists n.\ n \leftarrow t, t'$

$t \in$ `(`a`=`b `in` A`)` iff `axiom`$\leftarrow t\ \&\ a = b \in A$

$t \in a\texttt{<}b$ iff `axiom`$\leftarrow t\ \&\ \exists m, n.\ m \leftarrow a\ \&\ n \leftarrow b\ \&\ m$ is less than n

$t = t' \in A$ `list` iff A `list` type
& `nil`$\leftarrow t, t'$ or $\exists a, a', b, b'.$ `(`a`.`b`)`$\leftarrow t$ & `(`a'`.`b'`)`$\leftarrow t'$
& $a = a' \in A$ & $b = b' \in A$ `list`

$t = t' \in A \mid B$ iff $A \mid B$ type
& ($\exists a, a'.$ `inl(`a`)`$\leftarrow t$ & `inl(`a'`)`$\leftarrow t'$ & $a = a' \in A$)
or $\exists b, b'.$ `inr(`b`)`$\leftarrow t$ & `inr(`b'`)`$\leftarrow t'$ & $b = b' \in B$

$t = t' \in x : A\texttt{\#}B$ iff $x : A\texttt{\#}B$ type
& $\exists a, a', b, b'.$ `<`a`,`b`>`$\leftarrow t$ & `<`a'`,`b'`>`$\leftarrow t'$
& $a = a' \in A$ & $b = b' \in B[a/x]$

$t = t' \in x : A\texttt{->}B$ iff $x : A\texttt{->}B$ type
& $\exists u, u', b, b'.$ `\`u`.`$b\leftarrow t$ & `\`u'`.`$b'\leftarrow t'$
& $\forall a, a'.$ $b[a/u] = b'[a'/u'] \in B[a/x]$
if $a = a' \in A$

$t = t' \in \{x : A \mid B\}$ iff $\{x : A \mid B\}$ type & $t = t' \in A$ & $\exists b.$ $b \in B[t/x]$

$t = t' \in x, y : A\texttt{//}B$ iff $x, y : A\texttt{//}B$ type & $t \in A$ & $t' \in A$
& $\exists b.$ $b \in B[t, t'/x, y]$

$T = T' \in$ `U`k iff `void`$\leftarrow T, T'$ or `atom`$\leftarrow T, T'$ or `int`$\leftarrow T, T'$

or $\exists A, A', a, a', b, b'.$ `(`a`=`b `in` A`)`$\leftarrow T$ & `(`a'`=`b' `in` A'`)`$\leftarrow T'$
& $A = A' \in$ `U`k & $a = a' \in A$ & $b = b' \in A$

or $\exists a, a', b, b'.$ `(`a`<`b`)`$\leftarrow T$ & `(`a'`<`b'`)`$\leftarrow T'$
& $a = a' \in$ `int` & $b = b' \in$ `int`

or $\exists A, A'.$ A `list`$\leftarrow T$ & A' `list`$\leftarrow T'$ & $A = A' \in$ `U`k

or $\exists A, A', B, B'.$ $A \mid B \leftarrow T$ & $A' \mid B' \leftarrow T'$
& $A = A' \in$ `U`k & $B = B' \in$ `U`k

or $\exists A, A', x, x', B, B'.$
($x : A\texttt{\#}B \leftarrow T$ or $A\texttt{\#}B \leftarrow T$)
& ($x' : A'\texttt{\#}B' \leftarrow T'$ or $A'\texttt{\#}B' \leftarrow T'$)
& $A = A' \in$ `U`k
& $\forall a, a'.$ $B[a/x] = B'[a'/x'] \in$ `U`k if $a = a' \in A$

or $\exists A, A', x, x', B, B'.$
$\quad$ (x:A->$B$$\leftarrow T$ or A->$B$$\leftarrow T$)
$\quad$ & (x':A'->$B'$$\leftarrow T'$ or A'->$B'$$\leftarrow T'$)
$\quad$ & $A = A' \in \mathtt{U}k$
$\quad$ & $\forall a, a'.\ B[a/x] = B'[a'/x'] \in \mathtt{U}k$ if $a = a' \in A$

or $\exists A, A', x, x', B, B', e, e', u.$
$\quad$ ($\{x$:$A \mid B\}\leftarrow T$ or $\{A \mid B\}\leftarrow T$)
$\quad$ & ($\{x'$:$A' \mid B'\}\leftarrow T'$ or $\{A' \mid B'\}\leftarrow T'$)
$\quad$ & $A = A' \in \mathtt{U}k$
$\quad$ & $\forall a.\ B[a/x] \in \mathtt{U}k$ if $a \in A$
$\quad$ & $\forall a.\ B'[a/x'] \in \mathtt{U}k$ if $a \in A$
$\quad$ & u occurs in neither B nor B'
$\quad$ & $e \in u$:A->$B[u/x]$->$B'[u/x']$
$\quad$ & $e' \in u$:A->$B'[u/x']$->$B[u/x]$

or $\exists A, A', x, x', y, y', B, B', e, e', f, g, h, u, v, w.$
$\quad$ (x,y:A//$B$$\leftarrow T$ or A//$B$$\leftarrow T$)
$\quad$ & (x',y':A'//$B'$$\leftarrow T'$ or A'//$B'$$\leftarrow T'$)
$\quad$ & $A = A' \in \mathtt{U}k$
$\quad$ & $\forall a, b.\ B[a, b/x, y] \in \mathtt{U}k$ if $a \in A$ & $b \in A$
$\quad$ & $\forall a, b.\ B'[a, b/x', y'] \in \mathtt{U}k$ if $a \in A$ & $b \in A$
$\quad$ & u, v, w are distinct and occur in neither B nor B'
$\quad$ & $e \in u$:A->v:A->$B[u, v/x, y]$->$B'[u, v/x', y']$
$\quad$ & $e' \in u$:A->v:A->$B'[u, v/x', y']$->$B[u, v/x, y]$
$\quad$ & $f \in u$:A->$B[u, u/x, y]$
$\quad$ & $g \in u$:A->v:A->$B[u, v/x, y]$->$B[v, u/x, y]$
$\quad$ & $h \in u$:A->v:A->w:A->$B[u, v/x, y]$->$B[v, w/x, y]$->$B[u, w/x, y]$

or $\exists j.\ \mathtt{U}j\leftarrow T, T'$ & j is less than k

Typehood and Membership

What follows is *not* a definition but a body of facts easily derived from the definition above.

T type iff `void`$\leftarrow T$ or `atom`$\leftarrow T$ or `int`$\leftarrow T$

or $\exists A, a, b.$ `(`a`=`b `in` A`)`$\leftarrow T$ & $a \in A$ & $b \in A$

or $\exists a, b.$ `(`a`<`b`)`$\leftarrow T$ & $a \in$ `int` & $b \in$ `int`

or $\exists A.$ A `list`$\leftarrow T$ & A type

or $\exists A, B.\ A \mid B \leftarrow T$ & A type & B type

or $\exists A, x, B.$ ($x : A\texttt{\#}B \leftarrow T$ or $A\texttt{\#}B \leftarrow T$)
& A type & $\forall a, a'.\ B[a/x] = B[a'/x]$ if $a = a' \in A$

or $\exists A, x, B.$ ($x : A\texttt{->}B \leftarrow T$ or $A\texttt{->}B \leftarrow T$)
& A type & $\forall a, a'.\ B[a/x] = B[a'/x]$ if $a = a' \in A$

or $\exists A, x, B.$ ($\{x : A \mid B\} \leftarrow T$ or $\{A \mid B\} \leftarrow T$)
& A type & $\forall a, a'.\ B[a/x] = B[a'/x]$ if $a = a' \in A$

or $\exists A, x, y, B, f, g, h, u, v, w.$
($x\texttt{,}y : A\texttt{//}B \leftarrow T$ or $A\texttt{//}B \leftarrow T$)
& A type
& u, v, w are distinct and don't occur in B
& $f \in u : A\texttt{->}B[u, u/x, y]$
& $g \in u : A\texttt{->}v : A\texttt{->}B[u, v/x, y]\texttt{->}B[v, u/x, y]$
& $h \in u : A\texttt{->}v : A\texttt{->}w : A\texttt{->}B[u, v/x, y]\texttt{->}B[v, w/x, y]\texttt{->}$
$B[u, w/x, y]$

or $\exists k.\ \texttt{U}k \leftarrow T$

$t \in T$ if $T = T'$ & $t \in T'$

not $t \in$ `void`

$t \in$ `atom` iff $\exists i.\ i \leftarrow t$

$t \in$ `int` iff $\exists n.\ n \leftarrow t$

$t \in$ `(`a`=`b `in` A`)` iff `axiom`$\leftarrow t$ & $a = b \in A$

$t \in a\texttt{<}b$ iff `axiom`$\leftarrow t$ & $\exists m, n.\ m \leftarrow a$ & $n \leftarrow b$ & m is less than n

$t \in A$ `list` iff A `list` type
& `nil`$\leftarrow t$ or $\exists a, b.$ `(`a`.`b`)`$\leftarrow t$ & $a \in A$ & $b \in A$ `list`

$t \in A \mid B$ iff $A \mid B$ type
& ($\exists a.$ `inl(`a`)`$\leftarrow t$ & $a \in A$) or $\exists b.$ `inr(`b`)`$\leftarrow t$ & $b \in B$

$t \in x : A\texttt{\#}B$ iff $x : A\texttt{\#}B$ type & $\exists a, b.$ `<`a`,`b`>`$\leftarrow t$ & $a \in A$ & $b \in B[a/x]$

$t \in x{:}A\texttt{->}B$ iff $x{:}A\texttt{->}B$ type
& $\exists u, b.\ \backslash u\,.\,b \leftarrow t$
& $\forall a, a'.\ b[a/u] = b[a'/u] \in B[a/x]$
if $a = a' \in A$

$t \in \{x{:}A \mid B\}$ iff $\{x{:}A \mid B\}$ type & $t \in A$ & $\exists b.\ b \in B[t/x]$

$t \in x\texttt{,}y{:}A\texttt{//}B$ iff $x\texttt{,}y{:}A\texttt{//}B$ type & $t \in A$

$T \in \texttt{U}k$ iff $\texttt{void} \leftarrow T$ or $\texttt{atom} \leftarrow T$ or $\texttt{int} \leftarrow T$

or $\exists A, a, b.\ \texttt{(}a\texttt{=}b\ \texttt{in}\ A\texttt{)} \leftarrow T$ & $A \in \texttt{U}k$ & $a \in A$ & $b \in A$

or $\exists a, b.\ \texttt{(}a\texttt{<}b\texttt{)} \leftarrow T$ & $a \in \texttt{int}$ & $b \in \texttt{int}$

or $\exists A.\ A\ \texttt{list} \leftarrow T$ & $A \in \texttt{U}k$

or $\exists A, B.\ A \mid B \leftarrow T$ & $A \in \texttt{U}k$ & $B \in \texttt{U}k$

or $\exists A, x, B.$ ($x{:}A\texttt{\#}B \leftarrow T$ or $A\texttt{\#}B \leftarrow T$)
& $A \in \texttt{U}k$ & $\forall a, a'.\ B[a/x] = B[a'/x] \in \texttt{U}k$ if $a = a' \in A$

or $\exists A, x, B.$ ($x{:}A\texttt{->}B \leftarrow T$ or $A\texttt{->}B \leftarrow T$)
& $A \in \texttt{U}k$ & $\forall a, a'.\ B[a/x] = B[a'/x] \in \texttt{U}k$ if $a = a' \in A$

or $\exists A, x, B.$ ($\{x{:}A \mid B\} \leftarrow T$ or $\{A \mid B\} \leftarrow T$)
& $A \in \texttt{U}k$ & $\forall a, a'.\ B[a/x] = B[a'/x] \in \texttt{U}k$ if $a = a' \in A$

or $\exists A, x, y, B, f, g, h, u, v, w.$
($x\texttt{,}y{:}A\texttt{//}B \leftarrow T$ or $A\texttt{//}B \leftarrow T$)
& $A \in \texttt{U}k$
& $\forall a, a', b, b'.\ B[a, b/x, y] = B[a', b'/x, y] \in \texttt{U}k$
if $a = a' \in A$ & $b = b' \in A$
& u, v, w are distinct and don't occur in B
& $f \in u{:}A\texttt{->}B[u, u/x, y]$
& $g \in u{:}A\texttt{->}v{:}A\texttt{->}B[u, v/x, y]\texttt{->}B[v, u/x, y]$
& $h \in u{:}A\texttt{->}v{:}A\texttt{->}w{:}A\texttt{->}B[u, v/x, y]\texttt{->}B[v, w/x, y]\texttt{->}$
$B[u, w/x, y]$

or $\exists j.\ \texttt{U}j \leftarrow T$ & j is less than k

8.3 The Rules

The Nuprl system has been designed to accommodate the top–down construction of proofs by refinement. In this style one proves a judgement (i.e., a *goal*) by applying a *refinement rule*, thereby obtaining a set of judgements called *subgoals*, and then proving each of the subgoals. The mechanics of using the proof–editing system were discussed in chapter 7. In this section we will describe the refinement rules themselves. First we give some general comments regarding the rules and then proceed to give a description of each rule.

The Form of a Rule

To accommodate the top–down style of the Nuprl system the rules of the logic are presented in the following *refinement* style.

H `>>` T `ext` t `by` *rule*
  H_1 `>>` T_1 `ext` t_1
  $\vdots$
  H_k `>>` T_k `ext` t_k

The goal is shown at the top, and each subgoal is shown indented underneath. The rules are defined so that if every subgoal is true then one can show the truth of the goal, where the truth of a judgement is to be understood as defined in section 8.1. If there are no subgoals ($k = 0$) then the truth of the goal is axiomatic.

One of the features of the proof editor is that the extraction terms are not displayed and indeed are not immediately available. The idea is that one can judge a term T to be a type and T to be inhabited without explicitly presenting the inhabiting object. When one is viewing T as a proposition this is convenient, as a proposition is true if it is inhabited. If T is being viewed as a specification this allows one to implicitly build a program which is guaranteed to be correct for the specification. The extraction term for a goal is built as a function of the extraction terms of the subgoals and thus in general cannot be built until each of the subgoals have been proved. If one has a specific term, t, in mind as the inhabiting object and wants it displayed, one can use the explicit intro rule and then show that the type t `in` T is inhabited. The rules have the property that each subgoal can be constructed from the information in the rule and from the goal, exclusive of the extraction term. As a result some of the more complicated rules require certain terms as parameters.

Implicit in showing a judgement to be true is showing that the conclusion of the judgement is in fact a type. We cannot directly judge a term to be a type; rather, we show that it inhabits a universe. An examination of the semantic definition will reveal that this is sufficient for our purposes. Due to

the rich type structure of the system it is not possible in general to decide algorithmically if a given term denotes an element of a universe, so this is something which will require proof. The logic has been arranged so the proof that the conclusion of a goal is a type can be conducted simultaneously with the proof that the type is inhabited. In many cases this causes no great overhead, but some rules have subgoals whose only purpose is to establish that the goal is a type, that is, that it is *well–formed*. These subgoals all have the form H `>>` T `in U`i and are referred to as *well–formedness* subgoals.

Organization of the Rules

The rules for reasoning about each type and objects of the type will be presented in separate sections. Recall from above that for each judgement of the form H `>>` T `ext` t where the inhabiting object t is left implicit, there is a corresponding explicit judgement H `>>` t `in` T `ext axiom`. As the content of these judgements is essentially the same, the rules for reasoning about them will be presented together. In the preceding chapters, the rules have been classified mainly as introduction or elimination rules, where the introduction rules break down the form of the conclusion of a goal, and the elimination rules use a hypothesis. This is too coarse a classification for the purposes of this section, as the large majority of the rules are introduction rules. Here we will use different criteria for classifying the rules which will hopefully be more illuminating. For each type we will have the following categories of rules:

- *Formation*
  These rules give the conditions under which a canonical type may be judged to inhabit a universe, thus verifying that it is indeed a type.

- *Canonical*
  These rules give the conditions under which a canonical object (implicitly or explicitly presented) may be judged to inhabit a canonical type. Note that the formation rules are all actually canonical rules, but it is convenient to separate them.

- *Noncanonical*
  These rules give the conditions under which a noncanonical object may be judged to inhabit a type. The elimination rules all fall in this category, as the extract term for an elimination rule is a noncanonical term.

- *Equality*
  These rules give the conditions under which objects having the same outer form may be judged to be equal. Recall that the rules are being presented in implicit/explicit pairs, H `>>` T `ext` t and

H >> t in T. The explicit judgement H >> t in T is simply the reflexive instance of the general equality judgement H >> t = t' in T, and in most cases the rule for the general form is an obvious generalization of the rule for the reflexive form, and thus will be omitted. As the rules for the reflexive judgement are given in one of the other categories, there will be no equality rules presented for some types.

- *Computation*
  These rules allow one to make judgements of equalities resulting from computation.

Rules such as the *sequence*, *hypothesis* and *lemma* rules which are not associated with one particular type are grouped together under the heading "miscellaneous".

Specifying a Rule

In the context of a particular goal a rule is specified by giving a name and, possibly, certain parameters. As there are a large number of rules it would be unfortunate to have to remember a unique name for each one. Instead, there are small number of generic names, and the proof editor infers the specific rule desired from the form of the goal. In fact, for the rules dealing with specific types or objects of specific types, there are only the names *intro*, *elim* and *reduce*. The *intro* rules are those which break down the conclusion of the goal, and the *elim* rules are those which use a hypothesis. Accordingly, the first parameter of any elim rule is the declared variable or number of the hypothesis to be used. The *reduce* rules are the computation rules. The first parameter of a reduce rule is a number that specifies which term of the equality is to be reduced. Among the parameters of some rules are keyword parameters which have the following form:

- `new` $x_1, \cdots, x_n$
  This parameter is used to give new names for hypotheses in the subgoals. In most cases the defaults, which are derived from subterms of the conclusion of the goal, suffice. For technical reasons the same variable can be declared at most once in a hypothesis list, so if a default name is already declared a new name will have to be given. Whenever this parameter is used it must be the case that the names given are all distinct and do not occur in the hypothesis list of the goal.

- `using` T, `over` $z.T$
  These parameters are used when judging the equality of noncanonical forms in types dependent on the principal argument of the noncanonical form. The `using` parameter specifies the type of the principal argument of the noncanonical form. The value should be a canonical type which is appropriate for the particular noncanonical form.

The `over` parameter specifies the dependence of the type over which the equality is being judged on the principal argument of the form. Each occurrence of z in T indicates such a dependency. The proof editor always checks that the term obtained by substituting the principal argument for z in T is α–convertible to the type of the equality judgement.

- `at` Ui
  The value of this parameter is the universe level at which any type judgements in the subgoals are to be made. The default is `U1`.

Optional Parameters and Defaults

Each rule will be presented in its most general form. However, some of the parameters of a rule may be optional, in which case they will be enclosed by square brackets (`[]`). If a new hypothesis in a subgoal depends on an optional parameter, and in a particular instance of the rule the optional parameter is not given, that new hypothesis will not be added. Such a dependence is usually in the form of a hypothesis specifically referring to an optional `new` name. The `over` parameter discussed above is almost always optional. If it is not given, it is assumed that the type of the equality has no dependence on the principal argument of the noncanonical form.

The issue of default values for variable names arises when the main term of a goal's conclusion contains binding variables. In general, the default values are taken to be those binding variables. For example, the rule for explicitly showing a product to be in a universe is

H `>>` x:A`#`B `in` Ui `by intro [new y]`
`>>` A `in` Ui
y:A `>>` B `in` Ui

The rule is presented as if a new name is given, but the default is to use x. All the dependent types follow this general pattern.

For judging the equality of terms containing binding variables the binding variables of the first term are in general the default values for the "appropriate" new hypotheses. Consider the rule for showing that a spread term is in a type:

H `>>` `spread(`e`;`x`,`y`.`t`)` `in` T`[`e`/`z`]`
`by intro [over` z.T`]` `using` w:A`#`B `[new` u`,`v`]`
H `>>` e `in` w:A`#`B
H`,`u:A`,`v:B`[`u`/`w`]` `>>` t`[`u`,`v`/`x`,`y`]` `in` T`[<`u`,`v`>/`z`]`

Here the new variables default to x, y. If no new names are given and x and y don't appear in H, then the second subgoal will be

$$H, x{:}A, y{:}B \texttt{ >> } t \texttt{ in } T\texttt{[<}x\texttt{,}y\texttt{>/}z\texttt{]}$$

Again this is the general pattern for rules of this type.

Hidden Assumptions

For certain rules, we need to be able to control the free variables occuring in the extract term. The mechanism used to achieve this is that of *hidden* hypotheses. A hypothesis is hidden when it is displayed enclosed in square brackets. At the moment the only place where such hypotheses are added is in a subgoal of the set elim rule. The intended meaning of a hypothesis being hidden is that the name of the hypothesis cannot appear free in the extracted term; that is, that it cannot be used computationally. Accordingly, a hidden hypothesis cannot be the object of an `elim` or `hyp` rule. For the rules for which the extract term is the trivial term `axiom`, the extract term contains no free variable references and so all restrictions on the use of hidden hypotheses can be removed. The editor will remove the brackets from any hidden hypotheses in displaying a goal of this form.

Shortcuts in the Presentation

With the exception of one of the direct computation rules, each of the rules has the property that the list of hypotheses in a subgoal is an extension of the hypothesis list of the goal. To highlight the new hypotheses and to save space, we will show only the new hypotheses in the subgoals. Also, we will not explicitly display trivial extraction terms, that is, extraction terms which are just `axiom`.

ATOM

formation

1. H >> Ui ext atom by intro atom
2. H >> atom in Ui by intro

canonical

3. H >> atom ext "$\cdots$" by intro "$\cdots$"
4. H >> "$\cdots$" in atom by intro

where '$\cdots$' is any sequence of prl characters.

5. H >> atom_eq(a;b;t;t') in T by intro
   >> a in atom
   >> b in atom
   a=b in atom >> t in T
   (a=b in atom)->void >> t' in T

computation

6a. H >> atom_eq(a;a;t;t')=t'' in T by reduce 1
   >> t=t'' in T

where a is a canonical token term.

6b. H >> atom_eq(a;b;t;t')=t'' in T by reduce 1
   >> t'=t'' in T

where a and b are different canonical token terms.

VOID

formation

1. H >> Ui ext void by intro void
2. H >> void in Ui by intro

noncanonical

3. H,z:void >> T ext any(z) by elim z
4. H >> any(e) in T by intro
   >> e in void

INT

formation

1. H >> Ui ext int by intro int
2. H >> int in Ui by intro

canonical

3. H >> int ext c by intro c
4. H >> c in int by intro

where c must be an integer constant.

noncanonical

5. H >> -t in int
   >> t in int

6. H >> int ext m *op* n by intro *op*
   >> int ext m
   >> int ext n
7. H >> m *op* n in int by intro
   >> m in int
   >> n in int

where *op* must be one of +,-,*,/, or mod.

8. H,x:int,H' >> T ext ind(x;y,z.t_d;t_b;y,z.t_u) by elim x new z[,y]
   y:int,y<0,z:T[y+1/x] >> T[y/x] ext t_d
   >> T[0/x] ext t_b
   y:int,0<y,z:T[y-1/x] >> T[y/x] ext t_u

The optional new name must be given if x occurs free in H'.

9. H >> ind(e;x,y.t_d;t_b;x,y.t_u) in T[e/z]
   by intro [over z.T] [new u,v]
   >> e in int
   u:int,u<0,v:T[u+1/z] >> t_d[u,v/x,y] in T[u/z]
   >> t_b in T[0/z]
   u:int,0<u,v:T[u−1/z] >> t_u[u,v/x,y] in T[u/z]

10. H >> int_eq(a;b;t;t') in T by intro
    >> a in int
    >> b in int
    a=b in int >> t in T
    (a=b in int)->void >> t' in T

11. H >> less(a;b;t;t') in T by intro
    >> a in int
    >> b in int
    a<b >> t in T
    (a<b)->void >> t' in T

computation

12a. H >> ind(nt;x,y.t_d;t_b;x,y.t_u) = t in T by reduce 1 down
    >> t_d[nt,(ind(nt+1;x,y.t_d;t_b;x,y.t_u))/x,y] = t in T
    >> nt<0

12b. H >> ind(zt;x,y./t_d;t_b;x,y.t_u) = t in T by reduce 1 base
    >> t_b = t in T
    >> zt = 0 in int

12c. H >> ind(nt;x,y.t_d;t_b;x,y.t_u) = t in T by reduce 1 up
    >> t_u[nt,(ind(nt-1;x,y.t_d;t_b;x,y.t_u))/x,y] = t in T
    >> 0<nt

13a. H >> int_eq(a;a;t;t') = t'' in T by reduce 1
    >> t=t'' in T

13b. H >> int_eq(a;b;t;t') = t'' in T by reduce 1
    >> t' = t'' in T

where a and b are canonical int terms, and $a \neq b$.

14a. H >> less(a;b;t;t') = t'' in T by reduce 1
    >> t=t'' in T

where a and b are canonical int terms such that $a < b$.

14b. H >> less(a;b;t;t') = t'' in T by reduce 1
    >> t'=t'' in T

where a and b are canonical int terms such that $a \geq b$.

LESS

formation

1. H >> Ui ext a<b by intro less
   H >> int ext a
   H >> int ext b
2. H >> a<b in Ui by intro
   H >> a in int
   H >> b in int

equality

3. H >> axiom in a<b
   H >> a<b

LIST

formation

1. H >> Ui ext A list by intro list
   >> Ui ext A
2. H >> A list in Ui by intro
   >> A in Ui

canonical

3. H >> A list ext nil by intro nil at Ui
   >> A in Ui
4. H >> nil in A list by intro at Ui
   >> A in Ui
5. H >> A list ext $h.t$ by intro .
   >> A ext h
   >> A list ext t
6. H >> $a.b$ in A list by intro
   >> a in A
   >> b in A list

noncanonical

7. $H, x{:}A$ list, H' >> T ext list_ind($x; t_b; u, v, w.t_u$)
   by elim x new w, u[, v]
   >> T[nil/x] ext t_b
   $u{:}A, v{:}A$ list, $w{:}T[v/x]$ >> $T[u.v/x]$ ext t_u
8. H >> 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]
   >> e in A list
   >> t_b in T[nil/z]
   $u{:}A, v{:}A$ list, $w{:}T[v/z]$
   >> $t_u[u, v, w/x, y, z]$ in $T[u.v/z]$

computation

9a. H >> list_ind(nil; t_b; $u, v, w.t_u$) = t in T by reduce 1
   >> t_b = t in T
9b. H >> list_ind($a.b; t_b; u, v, w.t_u$) = t in T by reduce 1
   >> $t_u[a, b$,list_ind($b; t_b; u, v, w.t_u$)$/u, v, w]$ = t in T

UNION

formation

1. H >> Ui ext $A|B$ by intro union
   >> Ui ext A
   >> Ui ext B
2. H >> $A|B$ in Ui by intro
   >> A in Ui
   >> B in Ui

canonical

3. H >> $A|B$ ext inl(a) by intro at Ui left
   >> A ext a
   >> B in Ui
4. H >> inl(a) in $A|B$ by intro at Ui
   >> a in A
   >> B in Ui
5. H >> $A|B$ ext inr(b) by intro at Ui right
   >> B ext b
   >> A in Ui
6. H >> inr(b) in $A|B$ by intro at Ui
   >> b in B
   >> A in Ui

noncanonical

7. H,z:$A|B$,H' >> T ext decide(z;$x.t_l$;$y.t_r$) by elim z [new x,y]
   x:A,z=inl(x) in $A|B$ >> T[inl(x)/z] ext t_l
   y:B,z=inr(y) in $A|B$ >> T[inr(y)/z] ext t_r
8. H >> decide(e;$x.t_l$;$y.t_r$) in T[e/z]
   by intro [over $z.T$] using $A|B$ [new u,v]
   >> e in $A|B$
   u:A, e=inl(u) in $A|B$ >> t_l[u/x] in T[inl(u)/z]
   v:B, e=inr(v) in $A|B$ >> t_r[v/y] in T[inr(v)/z]

computation

9a. H >> decide(inl(a);$x.t_l$;$y.t_r$) = t in T by reduce 1
   >> t_l[a/x] = t in T
9b. H >> decide(inr(b);$x.t_l$;$y.t_r$) = t in T by reduce 1
   >> t_r[b/y] = t in T

FUNCTION

formation

1. H >> Ui ext x:A->B by intro function A new x
   >> A in Ui
   x:A >> Ui ext B
2. H >> x:A->B in Ui by intro [new y]
   >> A in Ui
   y:A >> B[y/x] in Ui
3. H >> Ui ext A->B by intro function
   >> Ui ext A
   >> Ui ext B
4. H >> A->B in Ui by intro
   >> A in Ui
   >> B in Ui

canonical

5. H >> x:A->B ext \y.b by intro at Ui [new y]
   y:A >> B[y/x] ext b
   >> A in Ui
6. H >> \x.b in y:A->B by intro at Ui [new z]
   z:A >> b[z/x] in B[z/y]
   >> A in Ui

noncanonical

7. H,f:(x:A->B),H' >> T ext t[f(a)/y] by elim f on a [new y]
   >> a in A
   y:B[a/x], y=f(a) in B[a/x] >> T ext t
8. H,f:(x:A->B),H' >> T ext t[f(a)/y] by elim f [new y]
   >> A ext a
   y:B >> T ext t

   The first form is used when x occurs free in B, the second when it doesn't.

9. H >> f(a) in B[a/x] by intro using x:A->B
   >> f in x:A->B
   >> a in A

equality

10. H `>>` f `=` g `in` x`:`A`->`B `by extensionality [new` y`]`
    y`:`A `>>` f`(`y`)` `=` g`(`y`)` `in` B`[`y`/`x`]`
    `>>` f `in` x`:`A`->`B
    `>>` g `in` x`:`A`->`B

computation

11. H `>>` `(\`x`.`b`)(`a`)` `=` t `in` B `by reduce 1`
    `>>` b`[`a`/`x`]` `=` t `in` B

PRODUCT

formation

1. H >> Ui ext x:A#B by intro product A new x
   >> A in Ui
   x:A >> Ui ext B
2. H >> x:A#B in Ui by intro [new y]
   >> A in Ui
   y:A >> B[y/x] in Ui
3. H >> Ui ext A#B by intro product
   >> Ui ext A
   >> Ui ext B
4. H >> A#B in Ui by intro
   >> A in Ui
   >> B in Ui

canonical

5. H >> x:A#B ext <a,b> by intro at Ui a [new y]
   >> a in A
   >> B[a/x] ext b
   y:A >> B[y/x] in Ui
6. H >> <a,b> in x:A#B by intro at Ui [new y]
   >> a in A
   >> b in B[a/x]
   y:A >> B[y/x] in Ui
7. H >> A#B ext <a,b> by intro
   >> A ext a
   >> B ext b
8. H >> <a,b> in A#B by intro
   >> a in A
   >> b in B

noncanonical

9. H,z:(x:A#B),H' >> T ext spread(z;u,v.t) by elim z new u,v
   u:A,v:B[u/x],z=<u,v> in x:A#B >> T[<u,v>/z] ext t
10. H >> spread(e;x,y.t) in T[e/z]
    by intro [over z.T] using w:A#B [new u,v]
    >> e in w:A#B
    u:A,v:B[u/w],e=<u,v> in w:A#B >> t[u,v/x,y] in T[<u,v>/z]

computation

11. H `>> spread(<`a`,`b`>;`$x, y.t$`) =` s `in` T `by reduce 1`
    `>>` t`[`$a, b/x, y$`]` `=` s `in` T

QUOTIENT

formation

1. H >> Ui ext $(x,y):A//E$ by intro quotient A,E new x,y,z
   >> A in Ui
   $x:A,y:A$ >> E in Ui
   $x:A$ >> $E[x,x/x,y]$
   $x:A,y:A,E[x,y/x,y]$ >> $E[y,x/x,y]$
   $x:A,y:A,z:A,E[x,y/x,y]$, $E[y,z/x,y]$ >> $E[x,z/x,y]$
2. H >> $(u,v):A//E$ in Ui by intro new x,y,z
   >> A in Ui
   $x:A,y:A$ >> $E[x,y/u,v]$ in Ui
   $x:A$ >> $E[x,x/u,v]$
   $x:A,y:A,E[x,y/u,v]$ >> $E[y,x/u,v]$
   $x:A,y:A,z:A,E[x,y/u,v]$, $E[y,z/u,v]$ >> $E[x,z/u,v]$

canonical

3. H >> $(x,y):A//E$ ext a by intro at Ui
   >> $(x,y):A//E$ in Ui
   >> A ext a
4. H >> a in $(x,y):A//E$ by intro at Ui
   >> $(x,y):A//E$ in Ui
   >> a in A

noncanonical

5. $H,u:(x,y):A//E,H'$ >> t=t' in T by elim u at Ui [new v,w]
   $v:A,w:A$ >> $E[v,w/x,y]$ in Ui
   >> T in Ui
   $v:A,w:A,E[v,w/x,y]$ >>
   $t[v/u]$ = $t'[w/u]$ in $T[v/u]$

equality

6. H >> $(x,y):A//E$ = $(u,v):B//F$ in Ui by intro [new r,s]
   >> $(x,y):A//E$ in Ui
   >> $(u,v):B//F$ in Ui
   >> A = B in Ui
   A=B in Ui,$r:A,s:A$ >> $E[r,s/x,y]$ -> $F[r,s/u,v]$
   A=B in Ui,$r:A,s:A$ >> $F[r,s/u,v]$ -> $E[r,s/x,y]$

7. H `>>` t `=` t' `in` (x, y)`:`A`//`E `by intro at U`i
   `>>` (x, y)`:`A`//`E `in U`i
   `>>` t `in` A
   `>>` t' `in` A
   `>>` E`[`$t, t'/x, y$`]`

SET

formation

1. H >> Ui ext $\{x{:}A \mid B\}$ by intro set A new x
   >> A in Ui
   $x{:}A$ >> Ui ext B
2. H >> $\{x{:}A \mid B\}$ in Ui by intro [new y]
   >> A in Ui
   $y{:}A$ >> $B[y/x]$ in Ui
3. H >> Ui ext $\{A \mid B\}$ by intro set
   >> Ui ext A
   >> Ui ext B
4. H >> $\{A \mid B\}$ in Ui by intro
   >> A in Ui
   >> B in Ui

canonical

5. H >> $\{x{:}A \mid B\}$ ext a by intro at Ui a [new y]
   >> a in A
   >> $B[a/x]$ ext b
   $y{:}A$ >> $B[y/x]$ in Ui

   All hidden hypothesis in H become unhidden in the second subgoal.

6. H >> a in $\{x{:}A \mid B\}$ by intro at Ui [new y]
   >> a in A
   >> $B[a/x]$
   $y{:}A$ >> $B[y/x]$ in Ui
7. H >> $\{A \mid B\}$ ext a by intro
   >> A ext a
   >> B ext b

   All hidden hypotheses in H become unhidden in the second subgoal.

8. H >> a in $\{A \mid B\}$ by intro
   >> a in A
   >> B ext b

noncanonical

9. H,u:$\{x{:}A \mid B\}$,H' >> T ext $(\backslash y.t)(u)$ by elim u at Ui [new y]
   y:A >> B[y/x] in Ui
   y:A,[B[y/x]],u=y in A >> T[y/u] ext t

   Note that the second new hypotheses of the second subgoal is hidden.

equality

10. H >> $\{x{:}A \mid B\}$ = $\{y{:}A' \mid B'\}$ in Ui by intro [new z]
    >> A = A' in Ui
    z:A >> B[z/x] -> B'[z/y]
    z:A >> B'[z/y] -> B[z/x]

EQUALITY

formation

1. H `>>` `U`i `ext` a_1=$\cdots$=a_n `in` A `by intro equality` A n
   `>>` A `in` `U`i
   `>>` A `ext` a_1
   $\vdots$
   `>>` A `ext` a_n

   The default for n is 1.

2. H `>>` (a_1=$\cdots$=a_n `in` A) `in` `U`i `by intro`
   `>>` A `in` `U`i
   `>>` a_1 `in` A
   $\vdots$
   `>>` a_n `in` A

canonical

3. H `>>` `axiom in` (a `in` A) `by intro`
   `>>` a `in` A

4. H,x:T,H' `>>` x `in` T `by intro`

   This rule doesn't work when T is a set or quotient term, since intro will invoke the equality rule for the set or quotient type, respectively. In any case, the `equality` rule can be used.

UNIVERSE

canonical

1. H `>> U`i `ext U`j `by intro universe U`j
2. H `>> U`j `in U`i `by intro`

where $j < i$. Note that all the formation rules are intro rules for a universe type.

noncanonical

Currently there are no rules in the system for analyzing universes. At some later date such rules may be added.

MISCELLANEOUS

hypothesis

1. $H, x{:}A, H'$ `>>` A' `ext` x `by hyp` x

   where A' is α-convertible to A

sequence

2. H `>>` T `ext` $(\backslash x_1.\cdots(\backslash x_n.t)(t_n))\cdots)(t_1)$
   `by seq` $T_1,\ldots,T_n$ `[new` $x_1,\ldots,x_n$`]`
   `>>` T_1 `ext` t_1
   $x_1{:}T_1$ `>>` T_2 `ext` t_2
   $\vdots$
   $x_1{:}T_1,\ldots,x_n{:}T_n$ `>>` T `ext` t

lemma

3. H `>>` T `ext` t`[term_of(`*theorem*`)/`x`]` `by lemma` *theorem* `[new` x`]`
   $x{:}C$ `>>` T `ext` t

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

def

4. H `>>` T `ext` t `by def` *theorem* `[new` x`]`
   x`:term_of(`*theorem*`) =` *ext-term* `in` C `>>` T `ext` t

   where C is the conclusion of the complete theorem, *theorem*, and *ext-term* is the term extracted from that theorem. [10]

explicit intro

5. H `>>` T `ext` t `by explicit intro` t
   `>>` t `in` T

cumulativity

6. H `>>` T `in U`i `by cumulativity at U`j
   `>>` T `in U`j

   where $j < i$

[10] This rule introduces very strong interproof dependencies. A proof using this rule depends not only on C but also on the way C is proved.

substitution

7. H `>>` T`[`t`/`x`]` `ext` s `by subst at U`i t=t' `in` T' `over` x.T
   `>>` t `=` t' `in` T'
   `>>` T`[`t'`/`x`]` `ext` s
   x:T' `>>` T `in U`i

direct computation

8. H `>>` T `ext` t `by compute using` S
   `>>` T' `ext` t
9. H, x:T, H' `>>` T'' `ext` t `by compute hyp i using` S
   H, x:T', H'`>>` T'' `ext` t

   where x:T is the i^{th} hypothesis, where S is obtained from T by "tagging" some of its subterms, and where 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 `[[*;`t`]]` or `[[`n`;`t`]]`, where n is a positive integer constant. The latter tag causes t to be computed for n steps, and the former causes computation to proceed as far as possible. There are some 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 variable 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. Note that many of the computation rules, such as the one for product, are instances of direct computation.

equality

10. H `>>` 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 `>>` 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. Roughly

speaking, `arith` knows about the ring axioms for integer multiplication and addition, the total order axioms of $<$, the reflexivity, symmetry and transitivity of equality, and a limited form of substitutivity of equality. We will describe the class of problems decidable by `arith` by giving an informal account of the procedure which `arith` uses to decide whether or not C follows from H.

The terms that `arith` understands are those denoting arithmetic relations, namely terms of the form s`<`t, s`=`t `in int` or the negation of a term of this form. As the only equalities `arith` concerns itself with are those of the form s`=`t `in int`, we will drop the `in int` and write only s`=`t in the rest of this description. For `arith` the negation of an `arith`metic relation $s\theta t$ where θ is one of $<$ or $=$ is of the form $(s\theta t)$`->void`, which we will write as $\neg s\theta t$. As integer equality and less–than are decidable relations, $s\theta t$ and $\neg\neg s\theta t$ denote the same relation and will be treated identically by `arith`.

The `arith` rule may be used to justify goals of the form

$$H_1\ ,\ \ldots\ ,\ H_n\ \texttt{>>}\ C_1\ \texttt{|}\ \ldots\ \texttt{|}\ C_m\ ,$$

where each H_i and C_i is a term denoting an arithmetic relation. If `arith` can justify the goal it will produce subgoals requiring the user to show that the left– and right–hand sides of each C_i denote integer terms. As a convenience `arith` will attempt to prove goals in which not all of the C_i are arithmetic relations; it simply ignores such disjuncts. If it is successful on such a goal, it will produce subgoals requiring the user to prove that each such disjunct is a type at the level given in the invocation of the rule.

`Arith` begins by normalizing the relevant formulas of the goal according to the following conventions:

1. Rewrite each relation of the form $\neg s = t$ as the equivalent $s < t \,|\, t < s$. A conclusion C follows from such an assumption if it follows from either $s < t$ or $t < s$; hence `arith` tries both cases. Henceforth, we assume that all negations of equalities have been eliminated from consideration.
2. Replace all occurrences of terms which are not addition, subtraction or multiplication by a new variable. Multiple occurrences of the same term are replaced by the same variable. `Arith` uses only facts about addition, subtraction and multiplication, so it treats all other terms as atomic. At this point all terms are built from integer constants and integer variables using $+$, $-$ and $*$.

3. Rewrite all terms as polynomials in canonical form. The exact nature of the canonical form is irrelevant (the reader may think of it as the form used in high school algebra texts) but has the important property that any two equal terms are identical. Each term now has the form $p + c\theta p' + c'$, where p and p' are nonconstant polynomials in canonical form, c and c' are constants, and θ is one of $<$, $=$ or $\geq$ ($s \geq t$ is equivalent to $\neg t < s$).

4. Replace each nonconstant polynomial p by a new variable, with each occurrence of p being replaced by the same variable. This amounts to treating each nonconstant polynomial as an atom. Now each formula is of the form $z + c\theta z' + c'$. `Arith` now decides whether or not the conclusion follows from the hypotheses by using the total order axioms of $<$, the reflexivity, symmetry, transitivity and substitutivity of $=$, and the following so–called *trivial monotonicity* axioms (c and d are constants).

   - $x \geq y, c \geq d \Rightarrow x + c \geq y + d$
   - $x \geq y, c \leq d \Rightarrow x - c \geq y - d$

   These rules capture all of the acceptable forms of reasoning which may be applied to formulas in canonical form.

For a detailed account of the `arith` rule and a proof of its correctness, see the article by Tat-hung Chan in [Constable, Johnson, & Eichenlaub 82].

ML Constructors

One of the features of the Nuprl system is that it comes with its own formal metalanguage, implemented in ML. Accordingly, ML needs access to the rules of the system. Following is a list of the ML rule constructors. The constructors are categorized and numbered in the same manner as the rules. For example, the constructor numbered 4 in the `atom` section corresponds to the rule numbered 4 in the `atom` section. Each constructor is presented in the form `constructor_name(parameter names): type`. The parameter names are never actually used but appear here solely for documentation purposes. Following are some conventions regarding the parameter names which we give with their types:

- `hyp:int` is used to indicate a hypothesis.
- `where:int` is used to indicate which equand of an equality should be reduced in a computation rule.
- `level:int` is used to give a universe level.
- `using:term` is used to indicate a using term.
- `over_id:tok, over:term` are used to indicate an over term. If the `over_id` is the token '`nil`', the over term is ignored.

All parameters corresponding to new identifiers have the same names as the new names given in the corresponding rule, and should be ML tokens. To specify the default identifier for an optional new identifier, give the token '`nil`'.

Unlike the rules, each constructor needs a unique name. The general form of the names is *base–type*`_`*kind*`_`*qualifier*, where *base–type* is a prl type and *kind* is one of `intro`, `elim`, `equality`, `formation` or `computation`. So, for example, `list_equality_nil` is the name of the constructor for the rule which specifies how to show two `nil` terms are the same in a `list` type, and `int_elim` is the name of the constructor for the elim rule for integers. There are some exceptions to this general pattern, the main one being that the name of the constructor for the rule for showing two canonical type terms to be equal in some universe is *type*`_equality` rather than `universe_equality_`*type*.

ATOM

formation

1. `universe_intro_atom: rule`.
2. `atom_equality: rule`.

canonical

3. `atom_intro(token): term → rule.`
4. `atom_equality_token: rule.`

equality

5. `atom_eq_equality: rule.`

computation

6. `atom_eq_computation(where): int → rule.`

VOID

formation

1. `universe_intro_void: rule.`
2. `void_equality: rule.`

noncanonical

3. `void_elim(hyp): int → rule.`
4. `void_equality_any: rule.`

INT

formation

1. `universe_intro_integer: rule.`
2. `int_equality: rule.`

canonical

3. `integer_intro_integer(c): int → rule.`
4. `integer_equality_natural_number: rule.`

noncanonical

5. `integer_equality_minus: rule.`

6a. `integer_intro_addition: rule.`
6b. `integer_intro_subtraction: rule.`
6c. `integer_intro_multiplication: rule.`
6d. `integer_intro_division: rule.`
6e. `integer_intro_modulo: rule.`

7a. `integer_equality_addition: rule.`
7b. `integer_equality_subtraction: rule.`
7c. `integer_equality_multiplication: rule.`
7d. `integer_equality_division: rule.`
7e. `integer_equality_modulo: rule.`

8. `integer_elim(hyp y z): int → tok → tok → rule.`
9. `integer_equality_induction(over_id over u v):`
   `tok → term → tok → tok → rule.`

equality

10. `int_eq_equality: rule.`
11. `int_less_equality: rule.`

computation

12. `integer_computation(kind where): tok → int → rule.`

    where kind should be one of ‘UP‘, ‘BASE‘ or ‘DOWN‘.

13. `int_eq_computation(where): int → rule.`
14. `int_less_computation(where): int → rule.`

LESS

formation

1. `universe_intro_less: rule.`
2. `less_equality: rule.`

equality

3. `axiom_equality_less: rule.`

LIST

formation

1. `universe_intro_list: rule.`
2. `list_equality(level): int → rule.`

canonical

3. `list_intro_nil(level): int → rule.`
4. `list_equality_nil(level): int → rule.`
5. `list_intro_cons: rule.`
6. `list_equality_cons: rule.`

noncanonical

7. `list_elim (hyp u v w): int → tok → tok → tok → rule.`
8. `list_equality_induction (over_id over using u v w):`
   `tok → term → term → tok → tok → tok → rule.`

computation

9. `list_computation(where): int → rule.`

UNION

formation

1. `universe_intro_union: rule.`
2. `union_equality: rule.`

canonical

3. `union_intro_left(level): int → rule.`
4. `union_equality_inl(level): int → rule.`
5. `union_intro_right(level): int → rule.`
6. `union_equality_inr(level): int → rule.`

noncanonical

7. `union_elim(hyp x y): int → tok → tok → rule.`
8. `union_equality_decide(over_id over using u v):`
   `tok → term → term → tok → tok → rule.`

computation

9. union_computation(where): int → rule.

FUNCTION

formation

1. universe_intro_function(x term): tok → term → rule.
2. function_equality(y): tok → rule.
3. universe_intro_function_independent: rule.
4. function_equality_independent: rule.

canonical

5. function_intro(level y): int → tok → rule.
6. function_equality_lambda(level z): int → tok → rule.

noncanonical

7. function_elim(hyp on y): int → term → tok → rule.
8. function_elim_independent(hyp y): int → tok → rule.
9. function_equality_apply(using): term → rule.

equality

10. extensionality(y): tok → rule.

computation

11. function_computation(where): int → rule.

PRODUCT

formation

1. universe_intro_product(x left_term): tok → term → rule.
2. product_equality(y): tok → rule.
3. universe_intro_product_independent: rule.
4. product_equality_independent: rule.

canonical

5. `product_intro_dependent(left_term level y):`
   `term → int → tok → rule.`
6. `product_equality_pair(level y): int → tok → rule.`
7. `product_intro: rule.`
8. `product_equality_pair_independent: rule.`

noncanonical

9. `product_elim(hyp u v): int → tok → tok → rule.`
10. `product_equality_spread(over_id over using u v):`
    `tok → term → term → tok → tok → rule.`

computation

11. `product_computation(where): int → rule.`

QUOTIENT

formation

1. `universe_intro_quotient(A E x y z):`
   `term → term → tok → tok → tok → rule.`
2. `quotient_formation(x y z): tok → tok → tok → rule.`

canonical

3. `quotient_intro(level): int → rule.`
4. `quotient_equality_element(level): int → rule.`

noncanonical

5. `quotient_elim (hyp level v w):`
   `int → int → tok → tok → rule.`

equality

6. `quotient_equality(r s): tok → tok → rule.`
7. `quotient_equality_element(level): int → rule.`

SET

formation

1. `universe_intro_set(x type): tok → term → rule.`
2. `set_formation(y): tok → rule.`
3. `universe_intro_set_independent: rule.`
4. `set_formation_independent: rule.`

canonical

5. `set_intro(level left_term y): int → term → tok → rule.`
6. `set_equality_element(level y): int → tok → rule.`
7. `set_intro_independent: rule.`
8. `set_equality_element: rule.`

noncanonical

9. `set_elim(hyp level y): int → int → tok → rule.`

equality

10. `set_equality(z): tok → rule.`

EQUALITY

formation

1. `universe_intro_equality(type number_terms):`
   `term → int → rule.`
2. `equal_equality: rule.`

canonical

3. `axiom_equality_equal: rule.`
4. `equal_equality_variable: rule.`

UNIVERSE

canonical

1. `universe_intro_universe (level): int → rule.`
2. `universe_equality: rule.`

MISCELLANEOUS

hypothesis

1. `hyp(hyp): int → rule.`

sequence

2. `seq(term_list id_list): (term list) → (tok list) → rule.`

   where the length of the token list should match the length of the term list.

lemma

3. `lemma(lemma_name x): tok → tok → rule.`

def

4. `def(theorem x): tok → tok → rule.`

explicit intro

5. `explicit_intro(term): term → rule.`

cumulativity

6. `cumulativity(level): int → rule.`

substitution

7. `substitution(level equality_term over_id over):`
   `int → term → tok → term → rule.`

direct computation

8. `direct_computation(tagged_term): term → rule.`
9. `direct_computation_hyp(hyp tagged_term):`
   `int → term → rule.`

equality

10. `equality: rule.`

arith

11. `arith(level): int → rule`.

Chapter 9

The Metalanguage

This chapter explores the metalanguage, ML, and its relationship to the object language, Nuprl. It examines details of the interface and explains how ML allows extensions to the deductive apparatus. Beginning with a brief summary of ML as a general purpose language, the chapter presents a detailed look at *tactics*, which are functions for developing proofs, and describes tools for writing tactics and ways to combine tactics. The chapter concludes with some example tactics.

9.1 An Overview of ML

This section gives a brief summary of the features of ML. For a complete account the reader is referred to the book *Edinburgh LCF* [Gordon, Milner, & Wadsworth 79]. Originally, ML was developed as the metalanguage for the LCF proof system; several dialects of the language have since evolved, including Cambridge ML, the implementation of ML used in Nuprl. The ML used in Nuprl differs in several minor ways from the original ML, for we have specialized the language to the Nuprl logic.

The principal features of ML are:

- a functional style including the ability to use functions as values;
- a mechanism allowing the definition of new (abstract) data types;
- a polymorphic type discipline; and
- an exception–trap mechanism.

Functional Style

Functions are defined using either an abstraction construct or an equational definition. The latter form can also be used to define functions by recursion.

Strictly speaking, ML does allow imperative features such as assignment, but these are almost never used in applications of ML to Nuprl.

Type Definitions

The basic ML data types are *tokens*, *integers*, *booleans* and a special type written "." whose only member is "()". The other basic data types include character strings (surrounded by back quotes; for example, `` `PRODUCT` `` is a token in ML), integers and boolean values, respectively.

ML provides the user with a facility for building more complex types from simpler ones by means of *type constructors* that build the types inhabited by complex values. For example, the unary constructor `list` builds the type of lists of its argument type. The type constructors available are:

#	the product type constructor;
+	the sum type constructor;
list	the list type constructor; and
→	the function type constructor.

The user may assign names for types built with the type constructors using the `lettype` construct.

The types built using the type constructors described above are inhabited by data built using *constructors*. The most commonly used ones are:

,	the tuple constructor;
`inl, inr`	the constructors for the (disjoint) sum type; and
.	the list constructor analogous to (infix) "cons".

The corresponding *destructor* functions decompose structured values. It is also possible to build structures, called *varstructs*, containing variables; these structures may then be matched with a similarly structured expression to obtain bindings for the embedded variables. All the identifiers used in a varstruct must be distinct. For example, the varstruct `[u;v]` can be matched to the expression `[1;2]`, resulting in `u` being bound to `1` and `v` being to `2`. If the match is not possible, an error is returned.

The user can also build new types using the `abstype` construct. As the name suggests, the form of the definition is in the style of an "abstract" data type; there are functions that translate between the new data type and its representation (which are visible only in the definition) and functions on the new type which are available throughout the scope of the definition. The defined type can be recursive if we use `absrectype` instead of `abstype`.

The type system of ML is partially described in figure 9.1. In addition to the types in figure 9.1 the metalanguage of Nuprl contains special basic types which are used for writing tactics. These include the types `proof, tactic` and `goal`. A complete discussion of these types is contained in section 9.3.

ty ::= sty	standard type
| ty `#` ty	cartesian product
| ty `+` ty	disjoint sum
| ty `->` ty	function type
sty ::= .|`int`|`bool`|`token`	basic types
| vty	type variable
| id	defined type
| tyarg id	abstract type
tyarg ::= sty	single type argument
| (ty,. . .,ty)	multiple type arguments

Figure 9.1: A Partial Syntax of ML Types

Polymorphic Typing

The ML system associates a *type* with every value, including functions, using Milner's type–checking algorithm; thus the programmer is usually not required to declare types explicitly. Furthermore, the type assignment is *polymorphic*; a given value may have a number of different types associated with it. Essentially, the "degree of freedom" of type assignments corresponds to the presence of *type variables* in the type expressions. For example, the identity function $\lambda x.x$ can be used on any type; we do not need to define different identity functions for each type. Thus, the type associated with the identity function is $\alpha \rightarrow \alpha$, where the α is a type variable.[1] A more interesting example is the list constructor ".". This has type $\alpha \times \alpha\ list \rightarrow \alpha\ list$. The user may also define an *append* function that "glues" two lists together; the system would assign to it the type $\alpha\ list \times \alpha\ list \rightarrow \alpha\ list$. Note that if an attempt is made to use the append function to concatenate a list of integers with a list of tokens, a type error will result.

Failures

ML has a sophisticated exception–handling facility. Certain functions yield runtime errors on certain arguments and are therefore said to *fail* on these arguments. The result of a failure is a token, usually the function name. The token returned by a failure can be specified by using the construct *failwith* in a fashion to be described later.

A special operator ? allows one to "catch" failures. The combination $e_1?e_2$ has the value e_1 unless e_1 results in a failure, in which case it has

[1] The system actually uses sequences of asterisks rather than Greek letters.

the value of e_2. As we shall see, the exception–trap mechanism is useful in writing tactics.

Syntax and Semantics

Figure 9.2 gives a summary of the basic constructs needed for programming in ML. An abstract syntax of ML expressions is summarized in figure 9.3, with expressions shown in order of decreasing precedence. These are essentially taken from [Gordon, Milner, & Wadsworth 79]. In addition to the expression forms shown in figure 9.3, ML contains the standard arithmetic and logical operators[2] and the equality predicate written in the traditional infix form. The various `let` forms have corresponding `where` forms in which the expression and the binding surround the `where`. For example, instead of `let x=3 in e` one could write `e where x=3`. The `where` constructs bind more tightly than the `let` constructs.

The semantics of ML can be given in terms of the *store–environment model*. Evaluation of expressions takes place in an *environment*, which is a mapping between identifiers and either values or locations; the *store* is a mapping between locations and values and is only needed to understand imperative constructs. Evaluating an expression in an environment produces a value or a failure. In the latter case a failure token, which may be used as a value in the rest of the program, is produced.

A declaration changes the environment. The general form of a declaration is `let b`, `letref b` or `letrec b`, where `b` is a binding. (Bindings are described later.) If a `let`, is used the identifiers in `b` cannot be updated; if assignable variables are required then `letref` must be used. The `letrec` form must be used with recursively defined functions. Bindings, like varstructs, have three general forms. There are *simple* bindings, *function definitions* and *multiple* bindings. The multiple binding is just an abbreviation for a sequence of simple bindings. In a simple binding a varstruct is equated to an expression. The form of the varstruct and of the expression are compared, and if they match structurally then the variables in the varstruct are bound to the corresponding substructures of the expression. The scope of the binding produced by an expression of the form `let x=`e_1 `in` $e2$ is the expression e_2. A declaration of the form `let x=`e_1 has scope throughout the rest of the session until another declaration causes global rebinding of the variables in `x`. Type declarations are effected using the `lettype`, `abstype` and `absrectype` forms. The scope of the bindings associated with these terms mirrors that of data bindings described above; type bindings are described in more detail in [Gordon, Milner, & Wadsworth 79].

The evaluation of expressions is more or less standard and will only be discussed briefly. The evaluation of a constant expression returns the value

[2] The notation for the logical "or" is `or` and for "and" is `&`.

Declarations

d ::= `let` b	basic form
| `letref` b	assignable variable
| `letrec` b	recursive function form
| `lettype` db	type definition
| `abstype` ab	abstract types
| `absrectype` ab	recursive abstract types

Bindings

b ::= v=e	
| id v1 v2 ... vn :ty = e	function definition
| b1 `and` b2 ... `and` bn	multiple binding

Varstructs

v ::= ()	empty varstruct
| id	identifier
| v:ty	type constraint
| v1.v2	list construction with *cons*
| v1,v2	tuple forming
| []	empty list
| [v1;v2;...;vn]	list of n elements

Figure 9.2: Syntax of Basic ML Constructs

Expressions

e	::=	ce	constant expression
	|	id	variable
	|	e1 e2	function application
	|	e:ty	type constraint
	|	e1.e2	list cons
	|	e1,e2	pairing
	|	v:=e	assignment
	|	`failwith` e	failure with explicit token
	|	`if` e1 `then` e1'	conditional
		{`if` e2 `then` e2'	
		.	
		.	
		.	
		`if` en `then` en'}	
		{`else` en"}	
	|	e1 ? e2	failure trap
	|	e1;e2;...;en	sequencing
	|	[]	empty list
	|	[e1;e2;...;en]	list of n elements
	|	d `in` e	local declaration
	|	\v1 v2 ... vn. e	function abstraction

Figure 9.3: Syntax of ML Expressions

denoted by the constant. The evaluation of a simple variable returns the value to which the variable is bound. The evaluation of an application, `f e`, results in the evaluation of `f`, which must be a function, the subsequent evaluation of `e` and finally the evaluation of the resulting application. The evaluation of the logical expressions `e1 or e2` and `e1 & e2` forces evaluation of `e1` first and only causes the evaluation of `e2` if the value of the logical expression is not determined by the value of `e1`. In a sequencing expression of the form `e1;e2;e3...;en` the expressions are evaluated in the given order and the value resulting from the evaluation of `en` is returned. The list construct is evaluated similarly; the evaluation of an expression of the form `[e1;e2;...;en]` results in the evaluation of the expressions `e1` through `en` in that order, with the value returned being the list formed from the values of the expressions. The order of evaluation is not important unless assignable variables are being updated in the subexpressions. The evaluation of `failwith e` causes the evaluation of `e` first, after which a failure (rather than an ordinary value) with the value of `e` is returned. For operational descriptions of other ML constructs, see [Gordon, Milner, & Wadsworth 79].

Using ML in Nuprl

ML can be used in conjunction with the Nuprl system in a variety of ways. The easiest way to invoke ML is to give the command `ml` in the command window; this invokes an interactive ML interpreter. Expressions for evaluation are typed at the M> prompt and ended with ;; followed by a carriage return. Figure 9.4 illustrates some examples of using ML from the Nuprl command window. The user's input is on lines indicated with `M>`, while the system responds on the line below with the value of the expression (except when the expression is a function) and the type of the expression. Note how a polymorphic expression is typed in the third example.

Another way to invoke ML is to use the `create` command to create an object of ML type as in the following example.

```
create m1 ml top
```

One can then invoke the text editor on the object using the `view` command. The body of an ML program can be written, and the result can be checked using the `check` command in the Nuprl command window. This process is illustrated in figure 9.5.

9.2 Tactics in ML

Conceptually it is convenient to think of tactics as mappings from proofs to proofs. The actual implementation of tactics, however, is not quite that

```
M>let x =3 in x+x;;
6 : int

M>let f x = x+3;;
f = - : (int -> int)

M>let g x = x @ x;;
g = - : (* list -> * list)
```

Figure 9.4: Examples of ML Expressions with Their Inferred Types

```
,-------------------------------------------------------------.
| NuPRL Command/Status                  | Library             |
|---------------------------------------|---------------------'
|P>cr m1 ml top                         |                     |
|P>v m1                                 |* m1                 |
|P>check m1                             |  ML                 |
|1 library object(s) checked.           |                     |
|P>                                     |                     |
|Returning to PRL.                      |                     |
'---------------------------------------'---------------------'
```

Figure 9.5: Using the Text Editor to Write ML Programs

simple. The actual type of tactics is[3]

```
tactic = proof -> proof list # validation,
```

where `proof` is the type of (possibly incomplete) proofs and

```
validation = proof list -> proof.
```

The idea is that a tactic decomposes a goal proof G into a finite list of subgoals, $G_1, \ldots, G_n$, and a validation, v. The subgoals are degenerate proofs, i.e., proofs which have only the top–level sequent filled in and have no refinement rule or subgoals. We say that a proof P_i *achieves* the subgoal G_i if the top–level sequent of P_i is the same as the top–level sequent of G_i. The validation, v, takes a list $P_1, \ldots, P_n$, where each P_i achieves G_i, and produces a proof P that achieves G. (Note that wherever we speak of proof in the context of ML, we mean possibly incomplete proofs.)

Thus a tactic completes some portion of a proof and produces subgoals that it was unable to prove completely and a validation that will take any achievement of those subgoals and produce an achievement of the original goal. If the validation is applied to achievements $P_1, \ldots, P_n$ that are complete proofs, then the result of the validation is a complete proof of the goal G. In particular, if the tactic produces an empty list of subgoals then the validation applied to the empty list produces a complete proof of the goal.

The advantage of this system becomes apparent when one composes tactics. Since a tactic produces an explicit list of subgoals a different (or the same) tactic can easily be applied to each of the subgoals; the tactics thus decompose goals in a top–down manner. This makes it easy to write specialized and modular tactics and to write special tacticals for composing tactics. This system has the disadvantage, however, that it does not easily accommodate a bottom–up style of tactic.

The Nuprl system recognizes two kinds of tactics: *refinement* tactics and *transformation* tactics. A refinement tactic operates like a derived rule of inference. Applying a refinement tactic results in the name of the tactic appearing as the refinement rule and any unproved subgoals produced by the tactic appearing as subgoals to the refinement. The following are the actual steps that occur when a refinement tactic is invoked.

1. The ML variable `prlgoal` is associated with the current sequent, which is viewed as a degenerate proof. Note that there may be a refinement rule and subgoals below the sequent but that these are ignored as far as refinement tactics are concerned.

2. The given tactic is applied to `prlgoal`, producing a (possibly empty) list of unproved subgoals and a validation.

[3] This is a specialization of the LCF [Gordon, Milner, & Wadsworth 79] concept of tactic in that both *goal* and *event* are taken as *partial proofs*.

3. The validation is applied to the subgoals.

4. The tactic name is installed as the name of the refinement rule in the proof. The refinement tree that was produced by the validation in the previous step is stored in the proof, and any remaining unproved subgoals become subgoals of the refinement step.

The four steps above assume that the tactic terminates without producing an error or throwing a failure that propagates to the top level. If such an event does occur then the error or failure message is reported to the user and the refinement is marked as *bad*; this behavior exactly mimics the failure of a primitive refinement rule.

Transformation tactics have the effect of mapping proofs to proofs. The result of a transformation tactic is spliced into the proof where the tactic was invoked and takes the place of any previous proof. The following are the steps that occur when a transformation tactic is invoked.

1. The ML variable `prlgoal` is associated with the proof below, and including, the current sequent.

2. The specified transformation tactic is applied to `prlgoal`, producing a (possibly empty) list of subgoals and a validation.

3. The validation is applied to the list of subgoals.

4. The proof that is the result of the previous step is grafted into the original proof below the sequent.

The key difference between refinement and transformation tactics is that transformation tactics are allowed to examine the subproof that is below the current node whereas refinement tactics are not. The result of a transformation tactic will, in general, depend upon the result of the examination. Since most tactics do not depend on the subproof below the designated node, they may be used as either transformation tactics or refinement tactics; in fact, since a refinement tactic cannot examine the subproof, any refinement tactic may be used as a transformation tactic. The main implementation difference between refinement tactics and transformation tactics is how the result of the tactic is used. In the former the actual proof constructed by the tactic is hidden from the user, and only the remaining unproved subgoals are displayed. In the latter the result is explicitly placed in the proof.

9.3 Basic Types in ML for Nuprl

In specializing the ML language to Nuprl we have made each of the *kinds* of objects that occur in the logic into ML base types. The ML type `proof` has

`proof`: Type of partial Nuprl proofs. Proofs consist of proof nodes. Each node represents one refinement step. A node consists of a sequent, a refinement rule and proofs of the children of the refinement, where the latter two will be missing in some leaf nodes of an incomplete proof.

`rule`: Type of Nuprl refinement rules.

`declaration`: Type of Nuprl bindings. A declaration associates a variable (in the environment of a sequent) with a type (term). The declarations in a Nuprl sequent occur on the left of the >>.

`term`: Terms of the logic.

Figure 9.6: Summary of the Primitive Object Language Types for ML

already been encountered. There are also types for terms, rules and declarations (see figure 9.6). These types should not be confused with *types* of the Nuprl logic; the types described here are data types in the programming language ML that represent objects of the logic.

For each of the types there is an associated collection of predicates, constructors and destructors. The predicates on the type `term`, for example, allow the kind of a term to be determined. An example of a predicate on terms is `is_universe_term`, which returns *true* if and only if the term it is applied to is a universe term. The constructors and destructors for each of the types allow new objects of the types to be synthesized and existing objects of the type to be divided into component parts. Appendix A contains a list of primitive extensions to ML that have been made in implementing Nuprl tactics.

Proofs

We think of proofs as a recursive data type whose components are as follows.

A list of declarations. These represent the hypotheses of the top–level sequent of the proof. They are accessed with the function `hypotheses`, which maps proofs to a list of terms.

A term. The goal of the top–level sequent. It is accessed with the function `conclusion`.

A rule. The top–level refinement rule. This will be missing when the goal has not been refined. It is accessed with the function `refinement`. The function fails if there is no refinement.

A list of proofs. The subgoals to the refinement, if any. They are accessed with the function `children`. The function fails if there is no refinement.

For the ML type `proof` a complement of destructors are available that allow the conclusion, hypotheses (declarations), rule and children to be extracted from a proof. There is only one primitive function in ML that constructs new proof objects; that is `refine`. The function `refine` maps rules into tactics and forms the basis of all tactics. When supplied with an argument rule and proof, `refine` performs, in effect, one refinement step upon the sequent of the proof using the given rule. The result of this is the typical tactic result structure of a list of subgoals paired with a validation. The list of subgoals is the list of children (logically sequents, but represented as degenerate proofs) resulting from the refinement of the sequent with the rule.

The function `refine` is the representation of the actual Nuprl logic in ML, for every primitive refinement step accomplished by a tactic will be performed by applying `refine`. The subgoals of the refinement are calculated by the Nuprl refinement routine, `deduce_children`, from the proof and the rule. Constructing the validation, an ML function, is more complicated. Given achievements of the subgoals, the purpose of the validation is to produce an achievement of the goal. The validation therefore constructs a new proof node whose sequent is the sequent of the original goal, whose refinement rule is the rule supplied as an argument to refine, and whose children are the achievements (partial proofs) of the subgoals.

The user will probably never construct or use a validation except to apply one to a list of achievements; all the validations one will probably use will be constructed by `refine` and by standard tacticals. The important thing to know about validations is that they cannot construct incorrect proofs. This is enforced by the strong type structure of ML and by the interface between Nuprl and ML. See [Constable, Knoblock, & Bates 84] for an account of how the correctness of proofs is ensured.

The function `refine_using_prl_rule` is a composition of `refine` and another ML function, `parse_rule_in_context`. This function takes a token (the ML equivalent of a string) and a proof and produces a rule. It uses the same parser that is used when a rule is typed into the proof editor. This allows one to avoid using the explicit rule constructors, although efficiency is lost. Note that the proof argument is used to disambiguate the rules.

In addition to `refine` there is one other basis tactic, `IDTAC`, which is the identity tactic. The result of applying `IDTAC` to a proof is one subgoal, the original proof, and a validation that, when applied to an achievement of the proof, returns the achievement. The function `IDTAC` is useful since it may serve as a *no–op* tactic.

Rules

The rule constructors in ML do not correspond precisely to the rules in Nuprl. Refinement rules in Nuprl are usually entered as "`intro`", "`elim`", "`hyp`", etc. Strictly speaking, the notation "`intro`" refers not to a single refinement rule but to a collection of introduction refinement rules, and the context of the proof is used to disambiguate the intended introduction rule at the time the rule is applied to a sequent. There is a similar ambiguity with the other *names* of the refinement rules. In addition to this ambiguity, the various sorts of the rules require different additional arguments. For example, applying an `intro` rule to "`>>1 in int`" requires no further information, but applying `intro` to "`>>int`" requires that an integer be given. Because rules in ML may exist independently of the proof context that allows the particular kind of rule to be determined, and because functions in ML are required to have a fixed number of arguments, the rule constructors have been subdivided beyond the ambiguous classes that are normally visible to the Nuprl user. For example, there are constructors like `universe_intro_integer`, `product_intro` and `function_intro`, all of which are normally designated in proof editing with `intro`. The ML rule constructors and destructors are listed along with the inference rules in chapter 8.

Declarations

The constructor for declarations, `construct_declaration`, takes a pair of type `tok#term` and produces a declaration. The token represents the variable name in the declaration. The general destructor, `destruct_declaration`, maps a declaration into pair of type `tok#term`. The ML function `id_of_declaration` maps a declaration into a token representing the variable of the declaration, and the function `type_of_declaration` maps a declaration to the term of the declaration.

Terms

The type of terms is a recursive data type; it is a variant union based upon the type of term: universe, function, product, etc. The function `term_kind` returns the kind of the term as a token: `` `UNIVERSE` ``,`` `FUNCTION` `` or `` `PRODUCT` ``, for example. Corresponding to each kind of terms is a boolean predicate: `is_universe_term`, `is_function_term` and `is_product_term`, for example.

For each kind of term there is a constructor and a destructor. For example, for universe terms the constructor, `make_universe_term`, maps integers (representing universe levels) to terms. The corresponding destructor is `destruct_universe` and maps terms to integers. The destructor fails if applied to a term that is not a universe term, and the constructors for terms

fail if the semantic restrictions on the arguments are not met. For example, `destruct_universe` fails if applied to a nonpositive integer since it is not a legal universe level.

Nuprl defs used to construct terms are invisible to ML. Terms are, in fact, composed of two parts: an underlying term structure and a print representation. The constructors and destructors described above operate on the term structure; in the case of the constructors a reasonable print representation is calculated automatically. See below for how to examine the print representation of a term.

In addition to the individual term constructors, constant terms may be used by entering any legal Nuprl term enclosed in single quotes, as in `'int in U7'`. Note that these quotation marks are different from those used for tokens in ML.

Print Representation

For the types of `term`, `rule` and `declaration`, the print representation of the object can be accessed using the functions `term_to_tok`, `rule_to_tok` and `declaration_to_tok`, respectively. These are the representations that would be used if the object were displayed by the Nuprl proof editor. This gives one a rudimentary way to check for defs.

A Note about Equality

In ML two objects of type `term` are equal if and only if they are α–convertible variants of each other. That is, they are equal if and only if the bound variables can be renamed so that the terms are identical. For objects of type `proof`, `rule` and `declaration`, two objects are equal if and only if they are the *same* object.

9.4 Tools for Tactic Writers

In this section we summarize some of the more important tools that have been included in the system to aid in writing tactics.

Tacticals

Tacticals allow existing tactics to be combined to form new tactics. In chapter 6 we saw several example tactics formed by `refine_using_prl_rule` and the tacticals `THEN`, `ORELSE` and `REPEAT`. In this section we summarize all of the currently existing tacticals.

tac_1 `THEN` tac_2: The `THEN` tactical is the composition functional for tactics. tac_1 `THEN` tac_2 defines a tactic which when invoked on a proof first applies tac_1 and then, to each subgoal produced by tac_1, applies tac_2.

tac `THENL` *tac_list*: The `THENL` tactical is similar to the `THEN` tactical except that the second argument is a list of tactics rather than just one tactic. The resulting tactic applies *tac*, and then to each of the subgoals (a list of proofs) it applies the corresponding tactic from *tac_list*. If the number of subgoals is different from the number of tactics in *tac_list*, the tactic fails.

tac_1 `ORELSE` tac_2: The `ORELSE` tactical creates a tactic which when applied to a proof first applies tac_1. If this application fails, or fails to make progress, then the result of the tactic is the application of tac_2 to the proof. Otherwise it is the result that was returned by tac_1. This tactical gives a simple mechanism for handling failure of tactics.

`REPEAT` *tac*: The `REPEAT` tactical will repeatedly apply a tactic until the tactic fails. That is, the tactic is applied to the goal of the argument proof, and then to the subgoals produced by the tactic, and so on. The `REPEAT` tactical will catch all failures of the argument tactic and can not generate a failure.

`COMPLETE` *tac*: The `COMPLETE` tactical constructs a tactic which operates exactly like the argument tactic except that the new tactic will fail unless a *complete* proof was constructed by *tac*, i.e., the subgoal list is null.

`PROGRESS` *tac*: The `PROGRESS` tactical constructs a tactic which operates exactly like the argument tactic except that it fails unless the tactic when applied to a proof makes some progress toward a proof. In particular, it will fail if the subgoal list is the singleton list containing exactly the original proof.

`TRY` *tac*: The `TRY` tactical constructs a tactic which tries to apply *tac* to a proof; if this fails it returns the result of applying `IDTAC` to the proof. This insulates the context in which *tac* is being used from failures. This is often useful for the second argument of an application of the `THEN` tactical.

Substitution

Two functions currently perform substitutions in terms: `substitute` and `replace`. Both take a term and a substitution as arguments and produce a term, where a *substitution* is a list of pairs, the first element of which is a term representing the variable to be substituted for and the second element

of which is the term to substitute for that variable. The function `substitute` will automatically rename bound variables in order to avoid capture in the term during the substitution,[4] while the function `replace` will perform a substitution without regard to capture.

Unification

The function `unify` is a unification function for terms. This function takes two terms and produces the most general unifier (a substitution) such that if a `replace` is performed on each of the terms using the unifier, the result is the same term. The unification function fails if there is no unifier. This function provides a simple way to decompose terms without using the term destructors. The only variables that will be substituted for are those whose names are preceded by an underscore, e.g., "`_meta`". For example,

```
unify 'x:int#(x=x in int)' '_var:_left_type#_right_type'
```

results in the unifier

```
[('_var', 'x');
 ('_left_type', 'int');
 ('_right_type', '(x=x in int)')].
```

9.5 Example Tactics

A few examples should give enough of the flavor of tactic writing in the Nuprl context to permit the reader to build substantial tactics of his own. Our first example is a very simple one which encodes a frequently used pattern of reasoning, namely, case analysis. When called as a refinement tactic with an argument which is a union type (i.e., of the form `A|B`) `cases` will do a `seq` with its argument and then do an `elim` on the copy of the term which appears in the hypothesis list of the second subgoal. The effect is essentially to implement a derived rule of the form

```
>> T   by cases 'A|B'
    >> A | B
  A >> T
  B >> T
```

except that any of the subgoals which the auto–tactic manages to prove will not appear. (Note the use of the quotes to make text into an ML term.) The following is the implementation of the tactic.

[4] This is usually how substitution takes place throughout the Nuprl system.

```
let cases union_term =
  refine (seq [union_term] [‘nil‘])
  THENL
    [IDTAC
    ;\pf.
     refine (union_elim (length (hypotheses pf)) ‘nil‘ ‘nil‘)
     pf
    ]
    THEN
      TRY (COMPLETE immediate)
 ;;
```

The first `refine` above will do the `seq` rule; the `‘nil‘` argument means that the new hypothesis in the second subgoal generated will not have a tag. The two members of the list following the `THENL` are what will be applied to the two subgoals resulting from the `seq`. The first one simply leaves alone the introduced union term, and the second does the elimination. The only complication is that the rule constructor `union_elim` requires the number of the hypothesis to be eliminated as an argument; in this case it is the last one, so the length of the current hypothesis list is the required number. Since the length of the actual proof to which `refine` is applied is necessary, the second tactic in the list of tactics is actually a lambda term. Recall that the type of tactics is `proof->proof list # validation`. Therefore, if X is a piece of ML code which uses an identifier `pf` and which is of type `tactic` under the assumption that `pf` is of type `proof`, then `\pf.(`X `pf)` is also a tactic. When it is applied to a proof `pf` will be bound to the proof, and then the tactic defined by X will be applied. In the last line an attempt is made to prove completely all of the unproved subgoals generated so far.

The next example illustrates some of the other functions available in the ML environment of Nuprl. Given an integer, `bring_hyp` returns a refinement tactic implementing the rule

```
>> T'  by bring_hyp i
  >> T->T'
```

where `T` is the type in hypothesis `i`. This is often useful when it is desired that the substitutions done by an elimination be done in a hypothesis also; using this tactic, doing the elimination and then doing an introduction accomplishes this.

Figure 9.7 shows the implementation of this tactic. The function `select` takes an integer n and a list and returns the n^{th} element of the list, and `type_of_declaration` breaks down a declaration into its two components and returns the second part. `make_function_term` takes an identifier x and terms A and B to a term x:A->B. Note that because of the intended purpose of the tactic no attempt is made to prove one of the generated subgoals.

```
let bring_hyp hyp_no proof =
  let hyp_term
   = type_of_declaration (select hyp_no (hypotheses proof)) in
  let new_conclusion
   = make_function_term `nil` hyp_term (conclusion proof) in
( refine (seq [new_conclusion] [`nil`])
 THENL
    [IDTAC
    ;(\proof.
      refine
        (function_elim_independent (length (hypotheses proof)))
         proof)
     THEN immediate
    ]
) proof
;;
```

Figure 9.7: The `bring_hyp` Tactic

Example Transformation Tactics

To illustrate the use of transformation tactics we examine a pair of tactics, `mark` and `copy`, that can be used to copy proofs. To use these tactics the user locates the proof he wants to copy and invokes the `mark` tactic as a transformation tactic. He then locates the goal where he wants the proof inserted and invokes `copy` as a transformation tactic. The application of `mark` does not change the proof and records the proof in the ML state so that it is available when the `copy` tactic is used. Note that the goal of the proof where the copied proof is inserted cannot be changed by the `copy` tactic.

The `mark` tactic is defined as follows. The variable `saved_proof` is a reference variable of type proof.

```
let mark goal_proof =
    (saved_proof := goal_proof;
     IDTAC goal_proof
    );;
```

The basis of the copy tactic is a verbatim copy of the saved proof. This is accomplished by recursively traversing the saved proof and refining using the refinement rule of the saved proof. The following is a first approximation of the copy tactic.

```
letrec copy_pattern pattern goal =
    (refine (refinement pattern)
     THENL (map copy_pattern (children pattern))
    ) goal;;


let copy goal =  copy_pattern saved_proof goal;;
```

This version of `copy` will fail if the saved proof is incomplete since the selector `refinement` fails if applied to a proof without a refinement rule. To correct this deficiency we define a predicate `is_refined` and change `copy_pattern` to apply `immediate` where there is no refinement in the saved proof.

```
letrec copy_pattern pattern goal =
    if is_refined pattern then
        (refine (refinement pattern)
         THENL (map copy_pattern (children pattern))
        ) goal
    else
        immediate goal;;
```

With this as a basis any number of more general proof–copying tactics can be defined. For example, the following version of `copy` looks up the actual formula being referenced in elimination rules, rather than just the index in the hypothesis list, and locates the corresponding hypothesis in the context where the proof is being inserted. Furthermore, if one of the refinements from the pattern fails in the new context then `immediate` is tried.

```
letrec copy_pattern pattern goal =
    if is_refined pattern then
        (refine (adjust_elim_rules pattern goal)
            THENL (map copy_pattern (children pattern))
         ORELSE immediate
        ) goal
    else
        immediate goal;;
```

The function `adjust_elim_rules` checks if the refinement is an elimination rule. If the refinement is not an elimination, the function returns the value of the rule unchanged. If it is an elimination rule, the tactic looks up the hypothesis in the pattern indexed by the rule and finds the index of an equal hypothesis in the hypothesis set of the goal. In this case the value returned is the old elimination rule with the index changed to be the index in the hypothesis set of the goal rather than the pattern.

A Larger Example

We end this section with a somewhat larger example; `backchain_with` is a tactic that implements *backchaining*, a restricted form of resolution. It tries to prove the goal by "backing through" hypotheses, i.e., by finding a hypothesis that is an implication whose consequent matches the conclusion of the goal, and then recursively calling itself to prove the antecedents of the implication.

The main component is `back_thru_hyp`, which, given i, proceeds as follows. First, it uses `match_part` to match the conclusion against part of hypothesis `i`; if the hypothesis has the form $x_1 : A_1$ `->` $\cdots$ `->` $x_m : A_m$ `->` `(` B_1 `&` $\cdots$ `&` B_n `)` (where some of the variables may not be present), and if terms can be obtained that, when substituted for the x_i in one of the B_j, yield the conclusion, then `match_part` succeeds and returns those terms. The terms are used to obtain instances of the A_i (`antecedents`). If $m = 0$ then the goal can be proved immediately; otherwise, a tactic based on the `seq` rule is used to obtain the subgoals that the `seq` rule would produce, except that all but the last of the subgoals have no extra hypotheses. The last of these subgoals is completely proved using repeated eliminations; of the rest, the ones which have conclusions of the form a `in` A are left for the membership tactic, and `backchain_with` is recursively applied to the remainder.

When invoked, `backchain_with` first breaks down the conclusion and then tries to back through each hypothesis in turn until it succeeds; if it does not, the tactic `tactic` is applied. If `tactic` is `\p.fail` then `backchain_with` behaves like a primitive Prolog interpreter, where the hypotheses are the clauses and the conclusion is the query. The fact that `backchain_with` takes a tactic as an argument allows one to make this basic search method more powerful, since it means that one can specify what actions to take at the "leaves" of the search.

```
letrec backchain_with tactic =

   let atomize_concl =
      REPEAT
      (\ p . let c = concl p in
             if is_function_term c or is_and_term c
             then intro p else fail
      )                  in

   let back_thru_hyp i p =
      let t = type_of_hyp i p and c = conclusion p  in
      let inst_list = match_part c t  in
      let antecedents = antecedents t inst_list  in
      ( if null antecedents then
```

```
            repeat_and_elim i
            THEN
            hypothesis
         else
            parallel_seq antecedents
            THEN
            (\ p . (if conclusion p = c then
                       repeat_function_elim i inst_list
                       THEN
                       ( hypothesis
                         ORELSE
                         ( apply_to_last_hyp repeat_and_elim
                           THEN
                           hypothesis
                         )
                       )
                     else
                       if_not_membership_goal
                         (backchain_with tactic)
                    )  p
            )
      ) p                          in

   atomize_concl
   THEN
   if_not_membership_goal
      (apply_to_a_hyp back_thru_hyp ORELSE tactic)

;;
```

9.6 Existing Tactics

This section describes some of the tactics which have been written so far. If you have a copy of the system you should find most, if not all, of the described tactics in the ML files that come with it. Since these files for the most part are fairly well documented, and since the tactic collection is not stable, we limit ourselves here to describing briefly some of the tactics which are particularly useful and which provide an example of what can be done. See the files for the exact names and types of the described tactics.

One of the most tedious parts of reasoning in Nuprl involves dealing with the numerous trivial membership subgoals which arise. Looking at the rules, one can see that many of them have subgoals of the form A `in U`i. Also, applying a lemma involves using the `lemma` rule and then applying the

`elim` rule to instantiate the universally quantified variables of the lemma; each such `elim` entails proving that the term introduced to instantiate the variable is of the appropriate type. Both these types of subgoals have the same form, t `in` T, and the vast majority are fairly trivial. Fortunately, the tactic written to deal with these (currently called `member`) has so far been able to prove almost all of them. This tactic repeatedly applies `intro` to the membership subgoal until the subgoal has been broken down into components with trivial enough conclusions (like x `in` A where $x:A$ is a hypothesis) to be immediately justifiable from the hypotheses. The hard part is guessing the `using` types required by many of the intro rules; for example, the `intro` rule for a goal of the form `>>` $f(a)$ `in` B takes as a parameter the type of f. It is the type–guessing component which contains the bulk of the heuristics of the tactic. A good part of the success of this tactic owes to the extensive use of extractions, which provide an important source of type information; this is discussed further in the next chapter.

There are several tactics whose function, in part, is to make the rules easier to use. Most of the primitive rules in Nuprl require arguments. This makes it difficult to chain them together using the tacticals, so several tactics have been written which apply an appropriate member of a class of rules and guess the arguments required. An example is the *tactic* `intro`, which handles all the introduction steps except product and set intro, where a "witness" term is required, and union intro, where one must decide which half to prove. These tactics also deal with noncanonical forms. That is, if a canonical form is expected, as in the case of an elimination where the hypothesis must be a canonical type, but a noncanonical form is encountered, as will often happen if one defines the basic objects of his theory to be extractions, then the tactic will first use the direct computation rules to do any necessary reduction steps.

Since the lemma rule causes the statement of the lemma to be put in the hypothesis list, proving goals whose conclusions are instances of previously proven results would be tedious without tactic support. A particularly useful tactic for this takes a theorem name and tries to match the goal's conclusion with part of the theorem in order to determine the terms with which to instantiate the theorem. If successful, the tactic will produce subgoals corresponding to the antecedents of the implication forming the theorem, the consequent of which was matched against the goal's conclusion.

Finally, we briefly discuss a set of functions which provide support for implementing term–rewriting tactics. They are slight modifications of what Paulson has written for use in LCF [Paulson 83b] and which he used to write the main tactics in his proof of the correctness of the unification algorithm [Paulson 84]. The basic idea is to use analogues of the tacticals `THEN`, `ORELSE`, etc., as the basis for constructing larger rewriting tactics from smaller ones. To this end we define a type of *conversions*:

```
lettype conv = term -> (term # tactic).
```

Given a term t, a conversion will produce a term u and a tactic which should prove a goal of the form H `>>` t=u `in` T. There is a function which takes a theorem name and produces from the theorem (assuming it has the appropriate form, which is roughly that of a universally quantified chain of implications which ends in an equality) a conversion which, if it succeeds, rewrites an instance of the left–hand side of the equality of the theorem into the corresponding instance of the right–hand side. The basic combining forms are `THENC`, which composes two conversions, `ORELSEC` and `REPEATC`, which also work analogously to the corresponding tacticals, and `SUBCONV` c, which for c a conversion takes a term t and returns the term which results from applying c to the immediate subterms of t. Using these connectives we can easily define functions which take a collection of conversions and employ various strategies for applying them to the subterms of a given term until some kind of normal form is reached. They have been used to write a collection of tactics which do various kinds of algebraic simplification. In the current system, because of certain inefficiencies which will not be present in the next version of the system, it is not feasible to do any really large rewriting tasks, but it does seem that tactics based on rewriting can handle a good part of the more tedious reasoning.

Chapter 10

Building Theories

This chapter is addressed primarily to those who plan to use Nuprl to build a nontrivial collection of theorems. In it we discuss working with proofs, structuring knowledge in a theory, representing common concepts in the Nuprl logic, and dealing with certain recurring problems.

10.1 Proofs

Proofs in informal mathematics generally have a linear form; one proceeds by deducing new facts from established ones until the desired result has been reached. Proofs in Nuprl, however, have a structure geared toward top–down development. This means, for example, that facts established earlier in the course of a proof are not immediately available. In this section we discuss this problem and others which arise from the structure of Nuprl proofs.

If in the middle of a proof one discovers that the current subgoal is similar to one proved previously, the pair of tactics `mark` and `copy` can be used. As mentioned in the preceding chapter, they are used by running `mark` as a transformation tactic on the subgoal which is thought to be similar and then running `copy` as a transformation tactic on the subgoal to be proved. The tactic `copy` will attempt to reproduce the proof of the `mark`ed subgoal, making simple modifications if the subgoals are not identical. Using `mark` and `copy` is not, however, a complete solution to the problem; it is often difficult to locate the required subgoal, and it can be time consuming to use the proof editor to get to it even when one has a very good idea of where it is.

Fortunately, this problem can to a large extent be avoided with a bit of planning. First, the linear style of informal mathematics can be approximated using the library by proving separately, as lemmas, those things of which it is anticipated that several uses will be made. Well–formedness

goals, i.e., those of the form >> T `in U`i, are by far the most common type of duplicated subgoal.[1] The tactic `immediate` can handle a good number of these, but many are quite nontrivial.[2] Therefore, it is often wise at the outset to prove the appropriate well–formedness theorems for the basic objects of a theory. For example, if the constructive reals, `R`, and addition of real numbers have been defined then the theorems

```
>> R in U1
>> All x,y:R.(x+y in R)
```

should be proved. If one is building a reasonably large theory then a much better solution is to define the basic objects of the theory to be extractions from theorems, for then such well–formedness subgoals can be handled automatically. This is discussed in more detail in the next section.

There is no "thinning" rule in Nuprl, i.e., a rule which would allow one to remove hypotheses from a goal. This means that it can be difficult for a tactic to assemble a proof in a bottom–up fashion; an already constructed proof cannot be used to prove a goal with the same conclusion unless the hypothesis lists are identical. A solution to this is to arrange things so that what are constructed bottom–up are tactics, not proofs. This was the approach taken in the writing of the term–rewriting package described in chapter 9. A function which rewrites a term t produces not a proof of t=u `in` T, for some u, T, but u and a *tactic* which can be applied to goals of the form >> t=u `in` T. This approach also provides some flexibility with respect to the containing type T.

We conclude this section by making a few points of a more minor nature. First, if in the middle of a proof one realizes that a particular unproven fact will be required several times in the subproof, the sequence rule, `seq`, can be used to get it into the hypothesis list. If one is reasonably careful in choosing what lemmas to prove, however, the need for this use of the sequence rule does not appear to arise too often. Second, it is a good idea to be attentive to the timing of `elim`'s. Often a considerable amount of work can be saved by doing them before other branching of the proof occurs. Also, an `elim` involves a substitution in the conclusion; since this substitution is not done in the hypotheses following the eliminated hypothesis, it may be desirable to do eliminations before the left part of a function type is moved into the hypothesis list via an introduction. Third, one should try to arrange things so that the hypothesis list does not become unnecessarily large, especially when writing tactics. For example, if one wants to use an instantiation of a universally quantified theorem, using the lemma rule and then doing the

[1] For example, doing `intro`'s on a universally quantified formula generates as many subgoals of this form as there are quantified variables.

[2] Proving that a type of the form t `in` T is well-formed, for instance, requires proving that t is a member of T, and this can be a Nuprl statement that a program meets its specifications.

appropriate number of eliminations will give as many extra hypotheses as there are universal quantifiers to remove; this can badly clutter up a proof. This can be avoided by first sequencing in the instantiated form of the lemma. The first goal will be to prove the instantiated version, which is done with the lemma rule, and the second subgoal will have just the desired version of the lemma added to the hypothesis list. Finally, when proving a conjunct A `&` B both A and B need to be proven, but it might be easier, or necessary, to have A as a hypothesis in the proof of B. If the first step in the proof of A `&` B is `seq` A B, then the nontrivial subgoals will be `>>` A and A `>>` B.

10.2 Definitions

The advantages of the definition mechanism in Nuprl are obvious, and it should be utilized as frequently as possible. By layering definitions finely, i.e., by having the right–hand sides of the definitions of theory objects mention primarily other definitions and not Nuprl code, the meaning will be clear at all stages of a proof. However, unless some care is taken it is easy to build definitions whose denoted text is very large, and this can affect system performance. In practice it has happened on occasion that objects whose display form looked quite innocent caused the system performance to be rather seriously degraded. Fortunately this problem can be avoided.

First, avoid unnecessary formal parameter duplication. For example, the obvious way to define negation of a rational number (which we think of as a pair of integers) is

```
-<x:Q> == < -fst(<x>), snd(<x>) >
```

where `fst(<x>)` and `snd(<x>)` are the appropriate selector functions defined in terms of `spread`. If we made all the definitions involved in rational arithmetic in this manner then the size of the expanded form of an arithmetic expression could be exponentially larger than the displayed form. To avoid this we make better use of `spread`:

```
-<x:Q> == spread( <x>; u,v. <-u,v>).
```

Of course, tricks such as the above cannot always be done, and even without formal parameter duplication defined objects will eventually be very large. A general way to handle the size buildup involves defining an object to be an extraction `term_of(`t`)`, where t is a theorem name, instead of a combination of other definitions and Nuprl code. Here are a few examples.

```
THM Q_
>> U1

DEF Q
Q == term_of(Q_)

THM mult_
>> Q -> Q -> Q

DEF mult
<x:Q>*<y:Q> == term_of(mult_)(<x>)(<y>)
```

Each of the two theorems would have an extraction which was the appropriate code. For example, the first step of the proof of `mult_` might be

```
explicit intro
\x. \y. <fst(x)*fst(y), snd(x)*snd(y)>
```

where `fst` and `snd` are projections from pairs defined using `spread`.

This method of using extractions is workable because of the direct computation rules. When a variable is declared in a hypothesis list to be of a type which is really an extraction (such as the type `Q` above), then the actual (canonical) type must be obtained before an elimination of the variable can be done. An application of the direct computation rule will do this, and in fact it is easy to incorporate such computations into tactics so that the "indirectness" of the definitions can be made virtually invisible.

It might be useful to have an idea of when display forms can be maintained. It is rather dismaying to be proceeding smoothly through a proof and suddenly come upon an unreadable chunk of raw Nuprl code resulting from definitions being eliminated. This doesn't happen too often, but when it does it can make a proof very difficult. Because definitions are text macros (i.e., they are not tied to term structure), substitution can cause definitions to be lost, leaving only the actual text the definition references denote. When a term u is substituted for a variable x in a term t, two different kinds of replacement are done. First, the usual substitution is done on the term t. Then a textual replacement is done, with a result the same as would be obtained by a user manually replacing in a ted window *all* visible occurrences of x, whether free or bound. If the two resulting terms are not equal (or if the textual replacement did not even result in a term), then the definition is removed, leaving just the term which resulted from the usual substitution. For example, consider substituting `x` for `y` in `all x:T. x<y` (where `all` has the usual definition). The substitution requires α–conversion; all `x`'s in the term are renamed `x@`i, and the substitution results in `x@`i`:T -> x@`i`<x`. The replacement, on the other hand, results in `all x:T. x<x`; these two terms

are not equal, so the definition `all` is not retained and the display form of the result is the former term.

We end this section with a word about debugging definitions. Many of the basic objects of a theory have a computational meaning and so can be executed. Before one starts proving properties of these objects it is a good idea to "debug" them, i.e., to use the Nuprl evaluator to test the objects in order to remove trivial errors. In this way one can avoid the frustrating experience of reaching the bottom of a proof and discovering that it cannot be completed because of a mistake in one of the definitions used.

10.3 Sets and Quotients

The quotient type provides a powerful abstraction mechanism and puts a user–defined equality into the theory so that the rules for reasoning about equality (the `equality` and `substitution` rules and all the `intro` rules for equality of noncanonical members) can be applied to it. The obverse of this is that if x is declared in a hypothesis list to be in a quotient type then any type T which appears as a later hypothesis or as the conclusion must be independent of the representation of x; that is, it must yield equal types when different but equal elements of the quotient type are substituted for x. For example, for t `in` T to be true under the assumption that x is in some quotient type, it is required that both t *and* T be independent of the representative for x (see the quotient–elim rule). Therefore, if we define the rational numbers to be the obvious quotient on pairs of integers whose second component is nonzero, then we will not be able to show that the following definition of less–than is a type:

```
<x:Q> < <y:Q> in Q ==
   fst(<x>)*snd(<y>) < fst(<y>)*snd(<x>).
```

The difficulty here is that type equality in Nuprl is strong; two types are equal if they are *structurally* the same, and so there are types which have the same members but are not equal. In particular, two less–than types can be shown to be equal only if it can be shown that their corresponding components are equal as integers. Therefore, if we could show that the above were a type then for any pairs of integers `<i,j>`, `<i',j'>`, and `<m,n>` such that `<i,j>` and `<i',j'>` are equal as rational numbers, we would have

```
( i*n < m*j ) = ( i'*n < m*j' )  in U1
```

. In particular, we would have

```
i*n = i'*n  in int,
```

which is not always true. Thus we cannot use the above definition of less–than over the quotient–type `Q`. However, we are only interested in the truth

value of less–than, i.e., whether or not it is inhabited, and this property *is* independent of representatives. In this situation, we *squash* the type. Define

```
||<T:type>|| == {0 in int|<T>}.
```

This squashes a type in the sense that given a type T, it produces a type which has one element if T is inhabited and is empty otherwise. For the less–than example, we define less–than to be the squash of what it was before, and the result is a type, since to show that two set types are equal involves showing the proposition parts are equivalent instead of equal. (See the rule for equality of set types.) More specifically, to prove $||T||$ `=` $||T'||$ `in` `U`i , one only has to be able to prove T `=>` T' and T' `=>` T. In general, this use of the squash operator will often be required when one wishes to create a type which deals with the representations of objects from a quotient type.

Note that squashing does away with any computational content that the type might have had (this will be discussed further later in this section). If you are interested in that content and you cannot reformulate so as to bring out from under the squash operator the part you are interested in, you cannot use the quotient type and must treat the equality you want as you would any other defined relation. More precisely, suppose that Q is some quotient type and $P(x)$ is some property of members of Q that one wishes to express in Nuprl. If, as above, one is interested only in the *truth value* of $P(x)$, i.e., whether or not its type representation is inhabited, and if the truth value respects the equality relation of Q, then the squash operator can be applied. Usually in this case the type resulting from a direct translation of $P(x)$ according to the propositions–as–types principle will have at most one member, and that member will be known in advance (e.g., `axiom`), so one would lose nothing by squashing. Often, however, P will have some content of interest; for example, it might involve an assertion of the existence of some object, and one might wish to construct a function which takes a member x of Q satisfying $P(x)$ to the object whose existence is asserted, but this may not be possible if the property has been squashed. What one has to do in this case is to separate $P(x)$ into two pieces so that it can be expressed by a type z:T `#` $||T'||$, or equivalently, $\{z{:}T\,|\,T'\}$, where T represents the computationally interesting part and T' represents the part of which only the truth value is interesting. This requires that the type T respect equality in x and that for any two representatives chosen for x the collection of computationally interesting objects be the same.

We illustrate the points raised above with an example. The usual formulation of the constructive real numbers is as a certain set of sequences of rational numbers with known convergence rates, where two such sequences are considered equal if they are the same in the limit. One crucial property of real numbers is that each one has a rational approximation to within any desired accuracy. Given a particular real number, obtaining such an approximation simply involves picking out an element suitably far along in

the sequence, since the convergence rate is known. However, clearly one can have two different *but equal* real numbers for which this procedure gives two different answers, and in fact it is impossible constructively to come up with approximations in a way which respects equality. Since many computations over the real numbers involve approximations, one cannot bury the property of having a rational approximation under the squash operator. The upshot is that the constructive reals cannot be "encapsulated" as a quotient type; one must take the reals to be just the type of rational sequences and treat the equality relation separately. However, one can still use the quotient type to some extent in a situation such as this. For example, in the real number case one can still form the quotient type and use it to express the equality relation. Thus, although hypothetical real numbers will be members of the unquotiented version, one can express equality between reals as equality in the quotient type and so can use the equality rule, etc., to reason about it.

It should not be concluded from the preceding discussion that one should never represent anything with the quotient type. There is a wide class of domains for which the quotient as a device for complete abstraction with respect to equality is a safe choice. This class consists of those domains where the desired equality is such that canonical representatives exist, i.e., where one can prove the existence of a *function* from the quotient type to its base type which takes equal members of the quotient type to the same member of the equivalence class in the base type. This will allow one to express properties which do not respect equality by having them refer to the canonical representative. An example of a common domain with this property is the rational numbers, where the canonical representative for a fraction is the fraction reduced to lowest terms.

The squash operation defined earlier is just one particular use of the information–hiding capability of the set type. It can also be used to remove from the "computational part" of the theorem information whose computational content is uninteresting and which would otherwise reduce the efficiency of the extracted code or information which would make the theorem weaker than one desired. As an example of the first kind of information, we could define the type of integers which are perfect squares two different ways:

```
n:int # Some m:int. (n=m*m in int)
```

or

```
{ n:int | Some m:int. (n=m*m in int) }.
```

Suppose we had some variable declared in a hypothesis to be in one of the perfect square types. In each case we could eliminate it to get an integer which we know to be a perfect square. In the first case we could do another elimination to get our hands on the integer whose square is the first integer,

and we could use this square root freely in the proof of the conclusion. However, this will not be allowed in the second case. The system will ensure that the proof that the number is a square is not used in the term which is built as the proof of the conclusion. On the other hand, if one is interested only in the integer which is a perfect square, and only needs the perfect square property to prove something about a program which does not need the square root, then the second version can be used to get a more efficient extracted object.

As an example of the second kind of information hiding referred to above, consider the specification of a function which is to search an array of integers for a given integer and return an index, where it is known in advance that the integer occurs in the array. If integer arrays of length n are represented as functions from $\{1..n\}$ to the integers, where $\{1..n\}$ is defined to be the appropriate subset of the integers, a naive translation of the requirements for the search program into the Nuprl logic might be the following:

```
All n:N. All a:{1..n}->int. All k:int.
  Some i:{1..n}. a(i)=k in int
  => Some i:{1..n}. a(i)=k in int
```

This specifies a trivial function which requires as an argument the answer which it is to produce, and so it does not capture the meaning of the informal requirements. This problem is solved by use of the set type to "hide" the proof of the fact that `k` occurs in the array. If we make the Nuprl definition

```
All <x>:<T> where <P>. <P'> ==
  <x>: {<x>:<T>|<P>} -> <P'>
```

then we can give the correct specification as

```
All n:N. All a:{1..n}->int.
  All k:int where ( Some i:{1..n}. a(i)=k in int ) .
    Some i:{1..n}. a(i)=k in int
```

In a proof of this assertion, if one has done `intro` steps to remove the universal quantifiers and thus has `n`, `a` and `k` declared in the hypothesis list, then the type that `k` is in is a set type. The only way to get at the information about `k` (i.e., that it occurs in the array) is to use the `elim` rule on it. However, doing so will cause the desired information to be *hidden* (see the set elim rule), and it will not become unhidden until the proof has proceeded far enough so that the required function has been completely determined. Thus, the information can be used only to prove that the constructed function meets the specification, not to accomplish the construction itself.

The set type should be regarded primarily as an information–hiding construct. It cannot be used as a general device for collecting together all objects (from some type) with a certain property, as it would in conventional mathematics. One of the reasons has just been discussed: if the set type is used

to represent a subtype of a given type then being given a member of the subtype does not involve being given a proof of its membership; that is, the proof that the member *is* a member will be hidden, and in the Nuprl theory this means that information is irretrievably lost. Another reason is that for x a member of a type A the property of being a member of $\{x:A\,|\,B\}$ cannot be expressed unless it is always true. This is because the type t `in` T is really a short form for t=t `in` T; it expresses an equality judgement and is well–formed only if t is a member of T. The problem remains of how to form the type which represents the power set of a given type. Bishop [Bishop 67] defines a subset of a given set S to be a set A together with an injection i from A into S. If one takes this definition and does the obvious translation into Nuprl then it turns out that the collection of such subsets (within some fixed universe `U`i) of a given type T is equivalent to the function type T`->U`i. This latter formulation, where we identify a subset with the predicate which "identifies" the members of the subset, is much simpler to deal with than Bishop's version. For example, the proposition that x (for x in T) is in a subset P is expressed just by P`(`x`)`; the union of two subsets P and P' is just `\x.`P`(x)|`P'`(x)`. Notice that being given the fact that x is in a subset includes being given the *proof* that it is. Of course, one can still use the set type in this context to hide information when desired.

10.4 Theories

In section 10.2 the use of extractions to keep the size of objects down was described; we now describe another important advantage of this scheme. The conclusion of the theorem from which an object is extracted gives a type for the object, and this type is used by a special tactic which proves the numerous "membership" subgoals (i.e., those of the form H`>>`t `in` T) that arise in virtually every proof. As briefly mentioned in chapter 9, this tactic works by repeatedly breaking down the conclusion using intro rules until the goals are trivial. Some of the intro rules require `using` terms as parameters, and these require types to be assigned to components of the conclusion of the goal; this task can be much easier when the extraction scheme is used, since it in effect supplies types for the important objects of the theory. When a theory is structured in this way, the membership tactic is able to prove most, and in some cases all (as in the theory of regular sets described in the next chapter), of the membership subgoals.

Proving such subgoals is tedious, and they arise often, so the use of extractions as previously described is almost a necessity for any serious user. Also, because a reference to an extraction contains the name of the theorem extracted from, tactics examining terms have "hooks" into the library available to them. Certain kinds of facts are naturally associated with defined objects. For example, a defined object which is a function may also be

functional over a domain with a different equality. Using conventions for the naming or placement of such facts can allow tactics to locate required information for themselves without requiring the user to explicitly supply lemma names. The convention could be that such facts are named using a certain extension to the name of the theorem extracted from, or that such facts are located in the library close to that theorem. Obviously this is far from being a solution to the problem of knowledge management; in the design of the future versions of the Nuprl system this problem will receive considerable attention.

There are a few minor points to keep in mind when putting together a substantial theory. First, be careful with the ordering of library objects. Other objects which are referenced in a given object must precede the given object in the library. ML objects can be particularly troublesome: if you modify and recheck an ML function then the change will not be reflected in an ML function which references it. There will be nothing to indicate that the second object should be rechecked, and until it is it will continue to refer to the old version of the modified function. Another problem is that deleting an ML object from the library does not undo the bindings created therein. Thus you can check the entire library (using, say, `check first-last`) without an error but then have errors occur when the same library is loaded into a fresh Nuprl session. Currently, the only way to solve these problems is by being careful of organization and by keeping track of the dependencies yourself.

Chapter 11

Mathematics Libraries

This chapter contains brief descriptions of some of the libraries which have been built with Nuprl. The accounts are intended to convey the flavor of various kinds of formalized mathematics. In much of the presentation which follows the theorems, definitions and proofs shown are not exactly in the form that would be displayed on the screen by Nuprl, although they come from actual Nuprl sessions (via the snapshot feature). The theorems (anything starting with ">>") will have some white space (i.e., carriage returns and blanks) added, while the DEF objects will have additional white space as well as escape characters and many of the parentheses omitted. (Usually the right–hand sides of definitions have seemingly redundant parentheses; these ensure correct parsing in any context.)

11.1 Basic Definitions

This section presents a collection of definitions which are of general use. Most libraries will include these objects or ones similar to them.

It is convenient to have a singleton type available. We call it `one`.

```
one == { x:int | x=0 in int }
```

Most programming languages contain a two–element type of "booleans"; we use the two–element type `two`.

```
two == { x:int | (x=0 in int | x=1 in int) }
```

Associated with the "boolean" type is the usual conditional operator, the "if–then–else" construct.

```
if <x:two> then <t1:exp> else <t2:exp> fi
   == int_eq( <x>; 0; <t1>; <t2> )
```

To make selections from pairs we define the basic selectors as follows.

```
1of( <P:pair> ) == spread( <P>; u,v.u )

2of( <P:pair> ) == spread( <P>; u,v.v )
```

To define some of the most basic types the usual order relations on integers are very useful. These must be defined from the simple < relation. Here are some of the basic order definitions as an example of how to proceed. The first is a definition of $\leq$, and the following two define notation for chains of inequalities.

```
<m:int> <= <n:int> == <n> < <m> -> void

<m:int> < <n:int> < <k:int>
   == <m> < <n> & <n> < <k>

<m:int> <= <n:int> <= <k:int>
   == <m> <= <n> & <n> <= <k>
```

Another useful concept is the idea of a finite interval in `int`; it may be defined as follows.

```
{ <lb:int>, ..., <hb:int> }
   == { i:int | <lb> <= i <= <hb> }
```

The basic logic definitions were discussed in chapter 3, but they are repeated in figure 11.1 for reference in the subsequent sections. To illustrate the kind of parenthesizing which it is wise to do in practice only white space has been added to the text of the actual library object.

11.2 Lists and Arrays

This section presents a few basic objects from list theory and array theory. We start with the idea of a *discrete* type, which is a type for which equality is decidable. Discreteness is required for the element comparisons that are needed, for example, in searching a list for a given element. Also, we need the idea that a proposition is *decidable*, i.e., that it is either true or false.

```
discrete(<T:type>) ==
   all x,y:<T>. (x=y in <T>) | ~(x=y in <T>)

< P: T->prop > is decidable on <T:type> ==
   all x:<T>. P(x) | ~P(x)
```

```
prop == U1

false == void

~<P:prop> == ( (<P>) -> void )

<P:prop> & <Q:prop> == ( (<P>) # (<Q>) )

<P:prop> => <Q:prop> == ( (<P>) -> (<Q>) )

<P:prop> <=> <Q:prop> ==
   ( ( (<P>) -> (<Q>) ) & ( (<Q>) -> (<P>) ) )

all <x:var>:<T:type>.<P:prop> ==
   ( <x>:(<T>) -> (<P>) )

all <x1:var>,<x2:var>:<T:type>.<P:prop> ==
   all <x1>:<T>. all <x2>:<T>. <P>

some <x:var>:<T:type>.<P:prop> ==
   ( <x>:(<T>) # (<P>) )

some <x1:var>, <x2:var> : <T:type> . <P:prop> ==
   some <x1>:<T>. some <x2>:<T>. <P>

<P:prop> vel <Q:prop> == ~ ( ~<P> & ~<Q> )

<P:prop> imp <Q:prop> == ~<P> vel <Q>

exists <x:var>:<T:type>.<P:prop> == ~ all <x>:<T>. ~<P>

<P:prop> equiv <Q:prop> == (<P> imp <Q>) & (<Q> imp <P>)
```

Figure 11.1: Logic Definitions

```
hd(<l:list>) == list_ind( <l>; "?"; h,t,v. h )

>> all A:U1. all l:A list.
      ~(l=nil in A list) => hd(l) in A

>> all A:U1. all x:A. all l:(A list).
      hd(x.l)=x in A

hdd[ <d:default> ]( <l:list> ) ==
   list_ind( <l>; <d>; h,t,v. h )

>> all A:U1. all x:A. all l:A list. hdd[x](l) in A

>> all A:U1. all x:A. all l:A list. all y:A.
      hdd[x](nil)=x in A & hdd[x](y.l)=y in A
```

Figure 11.2: Definition of Head

Figure 11.2 gives two versions of the list *head* function, which differ only on how they treat the empty list. The theorem statements following each definition define the behavior of the respective head function. The definition of the *tail* function is straightforward.

```
tl(<l:list>) == list_ind( <l>; nil; h,t,v. t )
```

Given a function of the appropriate type, the first theorem (named "`generalize_to_list`") listed in figure 11.3 specifies the construction of another function that iterates the given function over a list. The object extracted from the proof is used in a function which sums the elements of a list.

The definition of the membership predicate for lists uses a recursive proposition.

```
<e:element> on <l:list> over <T:type> ==
   list_ind( <l>; void; h,t,v. (<e>=h in <T>) | v )

>> all A:U1. all x:A. all l:A list.
      (x on L over A) in U1
```

The above type of use of `list_ind` to define a proposition is common and is worth a bit of explanation. Suppose that $l = a_1 \cdot \ldots \cdot a_n \cdot$ `nil` is some list whose elements come from a type T, and let e be in T. Then e `on` l `over` T evaluates to the type expression

$$(e = a_1\ in\ T)|(e = a_2\ in\ T)|\cdots|(e = a_n\ in\ T)|void;$$

```
>> all A:U1. all x:A. all f:(A#A)->A.
   some F:(A list)->A. all l:A list.
      F(nil)=x in A
    & F(l) = f( < hdd[x](l), F(tl(l)) > ) in A

>> all A:U1. all l:A list.
      (l=nil in A list) | ~(l=nil in A list)

len(<l:list>) == list_ind( <l>; 0; h,t,v. (v+1) )

sum(<l:list of integers>) ==
   oneof( term_of(generalize_to_list)
             (int) (0) (\p. (oneof(p)+twoof(p))) )
        (<l>)
```

Figure 11.3: Defining the Sum of a List of Integers

this type (proposition) is inhabited (true) exactly when one of the $e = a_i \ in \ T$ is true.

The following is another example of a recursively defined proposition; it defines a predicate over lists of a certain type which is true of list l exactly when l has no repeated elements.

```
<l:list> is nonrepetitious over <T:type> ==
   list_ind( <l>; one; h,t,v. ~(h on t over <T>) and v )
```

Given this definition, the following theorems are provable.

```
>> all A:U1. nil is nonrepetitious over A

>> all A:U1. all x:A. all l:A list.
     ~(x on l over A) & l is nonrepetitious over A
     =>  x.l is nonrepetitious over A
```

The next theorem defines a function which searches an array for the first occurrence of an element with a given property. (Arrays are taken to be functions on segments of the integers.)

```
search:
     >> all A:U1. all n:int.
        all f:{1,...,n}->A. all P:A->prop.
           P is decidable on A
           => all k:int.
                 1<=k<=n
                 =>   all x:{1,...,k}. ~P(f(x))
                    | some x:{1,...,k}. P(f(x))
                        & all y:{1,...,x-1}. ~P(f(y))
```

11.3 Cardinality

This section presents some elementary facts from the theory of cardinality and discusses a proof of the familiar pigeonhole principle.[1]

Figure 11.4 lists the definitions and lemmas used to prove the pigeonhole principle. `lemma_wf_1` is used to prove well-formedness conditions that arise in the other lemmas. We use the lemma `lemma_arith` when we want to do a case analysis on an integer equality. `lemma_hole` finds for some `f` and some `y` in the codomain of `f` an `x` in the domain of `f` for which `f(x)=y` (or indicates if no such `x` can be found). This lemma can be considered as a computational procedure that will be useful in the proof of the main lemma, `lemma_1`. This lemma is a simplified version of the theorem, with the domain of `f` only one element larger than the range. The theorem `pigeonhole_principle` follows in short order from `lemma_1`.

Using the pigeonhole principle one can prove a more general form of the principle. Figure 11.5 lists the definitions needed to state the theorem and the statement of the theorem.

The proof of `lemma_hole` is a straightforward induction on `m`, the size of the domain of `f`. It is left as an exercise.

We consider in depth the proof of `lemma_1`. In the proof given below we elide hypotheses which appear previously in order to condense the presentation. The system performs no such action. After introducing the quantifiers we perform induction on the eliminated set type `n` (we need to get an integer, `n1` below, to do induction on). This induction is over the size of the domain and range together.

[1] This principle states that any function mapping a finite set into a smaller finite set must map some pair of elements in the domain to the same element in the codomain.

```
     {1..<m:int>} == { s:int | 0<s # s<<m>+1 }

     P == { p:int | 0<p }

     <x:int><><y:int> == ~(<x>=<y> in int)

     <x:term>.=z.<y:term> == (<x>=<y> in int)

lemma_wf_1:
     >> all i:int.all j:int.all f:({1..i}->{1..j}).
            all k:{1..i}.(f(k) in int)

lemma_arith:
     >> all x:int.all y:int. x.=z.y | ~ x.=z.y

lemma_hole:
     >> all m:P.all n:P.all f:({1..m}->{1..n}).all y:{1..n}.
            (some x:{1..m}. f(x).=z.y)
          | (all x:{1..m}. f(x)<>y)

lemma_1:
     >> all n:P. all f:({1..n+1}->{1..n}). some i:{1..n+1}.
           some j:{1..n+1}. ( i<>j # f(i).=z.f(j) )

pigeonhole_principle:
     >> all n:P. all m:P. n<m -> all f:({1..m}->{1..n}).
          some i:{1..m}. some j:{1..m}.
             ( i<>j # f(i).=z.f(j) )
```

Figure 11.4: Proving the Pigeonhole Principle

```
     <T1:type> is equipollent with <T2:type>  ==
        some f:(<T1>-><T2>). some g:(<T2>-><T1>).
           all x:<T1>. ( g(f(x))=x in <T1> )
           # all y:<T2>. ( f(g(y))=y in <T2> )

     >> all A:U1.  A is equipollent with A

     >> all A:U1.all B:U1.
          A is equipollent with B -> B is equipollent with A

     >> all A:U1. all B:U1. all C:U1.
           A is equipollent with B # B is equipollent with C
           -> A is equipollent with C

general_php:
     >> all A,B:U1. all f:(A->B). all n,m:P.
              n<m
           -> A is equipollent with {1..m}
           -> B is equipollent with {1..n}
           -> some i,j:A. ~(i=j in A) # (f(i)=f(j) in B)
```

Figure 11.5: The General Pigeonhole Principle

```
+-----------------------------------------------------------+
|EDIT THM lemma_1                                           |
|-----------------------------------------------------------|
|# top 1 2                                                  |
|1. n:P                                                     |
|2. n1:int                                                  |
|[3]. 0<n1                                                  |
|>> all f:({1..n1+1}->{1..n1}).some i:{1..n1+1}.            |
|   some j:{1..n1+1}.  (i<>j#f(i).=z.f(j))                  |
|                                                           |
|BY elim 2 new ih,k                                         |
|                                                           |
|1# {down case}                                             |
|                                                           |
|2* 1...3.                                                  |
|   >> all f:({1..0+1}->{1..0}).some i:{1..0+1}.            |
|      some j:{1..0+1}.  (i<>j#f(i).=z.f(j))                |
|                                                           |
|3# 1...3.                                                  |
|   4. k:int                                                |
|   5. 0<k                                                  |
|   6. ih:all f:({1..k-1+1}->{1..k-1}).some i:{1..k-1+1}.   |
|      some j:{1..k-1+1}.(i<>j#f(i).=z.f(j))                |
|   >> all f:({1..k+1}->{1..k}).some i:{1..k+1}.            |
|      some j:{1..k+1}.  (i<>j#f(i).=z.f(j))                |
|                                                           |
+-----------------------------------------------------------+
```

The "up" case is the only interesting case. First we intro `f`, and then since we want to use the lemma `lemma_hole` a proper form of it is sequenced in and appears as hypothesis 9 below. We use this lemma on `f:{1..k}->{1..k}`. We then do a case analysis on the lemma.

The left case is trivial. We have some `x` such that `f(x)=f(k+1)`, and `x` is in `{1..k}`, so `x<>k`; thus we can intro `x` for `i` and `k+1` for `j`. Let us concentrate on the more interesting right case. Using our induction hypothesis requires a function `f:{1..k}->{1..k-1}`; our function will possibly map a value of the domain to `k`. We thus make a new function by taking the value of `f` that would have mapped to `k` to whatever `k+1` maps to. Since by the hole lemma (hypothesis 10) no `f(x)` has the same value as `f(k+1)`, our new function will just be a permutation of the old function.

```
+-----------------------------------------------------------+
|EDIT THM lemma_1                                           |
|-----------------------------------------------------------|
|# top 1 2 3 1 2 2 2                                        |
```

```
|1...6.                                                          |
|7. f:({1..k+1}->{1..k})                                         |
|8. all y:{1..k}. (some x:{1..k}.f(x).=z.y)                      |
|                 |(all x:{1..k}.f(x)<>y)                        |
|9.  (some x:{1..k}.f(x).=z.f(k+1))                              |
|   |(all x:{1..k}.f(x)<>f(k+1))                                 |
|10. (all x:{1..k}.f(x)<>f(k+1))                                 |
|>> some i:{1..k+1}.some j:{1..k+1}.(i<>j#f(i).=z.f(j))          |
|                                                                |
|BY elim 6 on \x.int_eq(f(x);k;f(k+1);f(x))                      |
|                                                                |
|1# 1...10.                                                      |
|   >> \x.int_eq(f(x);k;f(k+1);f(x))                             |
|       in ({1..k-1+1}->{1..k-1})                                |
|                                                                |
|2# 1..10.                                                       |
|   11. i:({1..k-1+1})#(j:({1..k-1+1})#((i<>j)#                  |
|         ((\x.int_eq(f(x);k;f(k+1);f(x)))(i)                    |
|         =(\x.int_eq(f(x);k;f(k+1);f(x)))(j) in int)))          |
|   >> some i:{1..k+1}.some j:{1..k+1}.(i<>j#f(i).=z.f(j))       |
|                                                                |
+----------------------------------------------------------------+
```

We can now unravel our induction hypothesis into hypotheses 12, 14, 16 and 17 below. We want to do a case analysis on `f(i)=k | ~f(i)=k` (hypothesis 18); we sequence this fact (proven by `lemma_arith`) in. Consider the case where `f(i)=k`. Then we can prove `f(k+1)=f(j)` by reducing hypothesis 17 (note how we have to sequence in 20 and 21, and then `f(k+1)=f(j)`, in order to reduce the hypothesis).

```
+-----------------------------------------------------------+
|EDIT THM lemma_1                                           |
|-----------------------------------------------------------|
|# top 1 2 3 1 2 2 2 2 1 1 1 2 1 2 2                        |
|1. n:P                                                     |
|12. i:{1..k-1+1}                                           |
|14. j:{1..k-1+1}                                           |
|16. i<>j                                                   |
|17. (\x.int_eq(f(x);k;f(k+1);f(x)))(i)=                    |
         (\x.int_eq(f(x);k;f(k+1);f(x)))(j) in int          |
|18. f(i).=z.k|~f(i).=z.k                                    |
|19. f(i).=z.k                                              |
|20. (\x.int_eq(f(x);k;f(k+1);f(x)))(i)=f(k+1) in int       |
|21. (\x.int_eq(f(x);k;f(k+1);f(x)))(j)=f(j) in int         |
|>> some i:{1..k+1}.some j:{1..k+1}.(i<>j#f(i).=z.f(j))     |
|                                                           |
|BY (*combine 17,20,21*)                                    |
|   seq f(k+1)=f(j) in int                                  |
|                                                           |
|1* 1...19.                                                 |
|   20. (\x.int_eq(f(x);k;f(k+1);f(x)))(i)=f(k+1) in int    |
|   21. (\x.int_eq(f(x);k;f(k+1);f(x)))(j)=f(j) in int      |
|   >> f(k+1)=f(j) in int                                   |
|                                                           |
|2# 1...21.                                                 |
|   22. f(k+1)=f(j) in int                                  |
|   >> some i:{1..k+1}.some j:{1..k+1}.(i<>j#f(i).=z.f(j))  |
|                                                           |
+-----------------------------------------------------------+
```

We can now prove our goal by letting `i` be `k+1` and `j` be `j`. `k+1<>j` follows from hypothesis 14.

```
+-------------------------------------------------------+
|EDIT THM lemma_1                                       |
|-------------------------------------------------------|
|# top 1 2 3 1 2 2 2 2 1 1 1 2 1 2 2 2 2 2              |
|1...13.                                                |
|14. j:{1..k-1+1}                                       |
|15..22.                                                |
|>> (k+1 <>j#f(k+1 ).=z.f(j))                           |
|                                                       |
|BY intro                                               |
|   ....                                                |
|                                                       |
+-------------------------------------------------------+
```

This concludes this part of the proof. We now consider the other case, i.e., `~f(i)=k`. If `f(j)=k` then the conclusion follows by symmetry from above (note that in the current system we still have to prove this; eventually a smart symmetry tactic could do the job). Otherwise, `~f(j)=k`, and we can reduce hypothesis 17 to `f(i)=f(j)` as done previously (see hypotheses 22 and 23 below). The conclusion then follows from hypotheses 16 and 24.

```
+-------------------------------------------------------+
|EDIT THM lemma_1                                       |
|-------------------------------------------------------|
|* top 1 2 3 1 2 2 2 2 1 1 1 2 2 2 2 2 2 2              |
|1...15.                                                |
|16. i<>j                                               |
|17. (\x.int_eq(f(x);k;f(k+1);f(x)))(i)=                |
|        (\x.int_eq(f(x);k;f(k+1);f(x)))(j) in int      |
|18. f(i).=z.k|~f(i).=z.k                                |
|19. ~f(i).=z.k                                         |
|20. f(j).=z.k|~f(j).=z.k                                |
|21. ~f(j).=z.k                                         |
|22. (\x.int_eq(f(x);k;f(k+1);f(x)))(i)=f(i) in int     |
|23. (\x.int_eq(f(x);k;f(k+1);f(x)))(j)=f(j) in int     |
|24. f(i)=f(j) in int                                   |
|>> (i<>j#f(i).=z.f(j))                                 |
|                                                       |
|BY intro                                               |
| ...                                                   |
|                                                       |
+-------------------------------------------------------+
```

This completes the proof of `lemma_1`. The proof of `pigeonhole_principle` is straightforward and is left as an exercise for the reader.

The evaluator, when applied to `term_of(pigeonhole_principle)` on a

```
term_of(lemma_hole)(4)(2)(\x.int_eq(x;2;1;2))(2) -->
inl(<1,axiom>)

term_of(lemma_1)(3)(2)(\x.int_eq(x-1;0;2;x-1)) -->
<1,<3,<\v19.spread(axiom;v20,v21.axiom),axiom>>>

term_of(pigeonhole_principle)(1)(3)(axiom)(\x.1) -->
<1,<2,<\v19.spread(axiom;v20,v21.axiom),axiom>>>

term_of(pigeonhole_principle)
     (2)(3)(axiom)(\x.int_eq(x;3;1;x)) -->
<3,<1,<\v19.spread(axiom;v20,v21.axiom),axiom>>>
```

Figure 11.6: Using the Evaluator on the Pigeonhole Principle

given function `f`, returns the `i` and `j` such that `f(i)=f(j)` and `i<>j`. Figure 11.6 gives some examples of such evaluations performed by the evaluator. The redex on the left of the `-->` evaluates to the contractum on the right of the `-->`.

11.4 Regular Sets

This section describes an existing library of theorems from the theory of regular sets. The library was constructed in part to try out some general–purpose tactics which had recently been written; the most useful of these were the membership tactic (see chapter 9) and several tactics which performed computation steps and definition expansions in goals. The membership tactic was able to prove all of the membership subgoals which arose, i.e., all those of the form H `>>` t `in` T. The other tactics mentioned were heavily used but required a fair amount of user involvement; for example, the user must explicitly specify which definitions to expand. Including planning, the total time required to construct the library and finish all the proofs was about ten hours.

For the reader unfamiliar with the subject, this theory involves a certain class of computationally "simple" sets of words over some given alphabet; the sets are those which can be built from singleton sets using three operations, which are denoted by $r + s$, rs and r^*. There should be enough explanation in what follows to make the theorems understandable to almost everyone.

We begin with a presentation of the library; a discussion of some of the objects will follow. This theory uses extractions for most of the basic objects. For example, the right–hand side of the definition for the function `flatten`

is just `term_of(flatten_)(<l>)`; the actual Nuprl term for the function is introduced in the theorem `flatten_`. As a convention in this theory, a `DEF` name with "_" appended is the name of the corresponding theorem that the defined object is extracted from. Figure 11.7 presents the first part of the library, which contains the definitions of the basic objects, such as basic list operations and the usual operations on regular sets. Objects not particular to this theory, such as the definitions of the logical connectives, are omitted. The alphabet over which our regular sets are formed is given the name `E` and is defined to be `atom` (although all the theorems hold for any type). The collection of words over `E`, `E*`, is defined to be the lists over `E`, and the regular sets for the purposes of this theory are identified with their meaning as subsets of `E*`:

```
E* == (E list)

R(E) == (E*->U1)
```

Note first that we have identified the subsets of `E*` with the predicates over `E*`; for more on representing sets, see the preceding chapter. This means that if x is a word in `E*`, and r is a regular set from `R(E)`, then the assertion that x is a member of r is expressed as the application $r(x)$.

Figure 11.8 shows the next part of the library, which contains some lemmas expressing some elementary properties of the defined objects. It should be noted here that the theorems we are proving are identities involving the basic operations, and they are true over all subsets, not just the regular ones. This allows us to avoid cutting `R(E)` down to be just the regular sets, and we end up proving more general theorems. (It would be a straightforward matter to add this restriction by using a recursive type to define the regular *expressions* and defining an evaluation function.)

Figure 11.9 list some lemmas expressing properties of regular sets, and the main theorem of the library.

We now turn our attention to objects contained in the library. The following descriptions of library objects should give the flavor of the development of this theory. This proof top and `DEF` together give the definition of `append`.

```
>> E* -> E* -> E*
BY explicit intro \x.\y. list_ind(x;y;h,t,v.(h.v))

<l1:A list>@<l2:A list> == (term_of(append_)(<l1>)(<l2>))
```

We use `append` to define the function `flatten`, which takes a list of lists and concatenates them:

```
>> E* list -> E*
BY explicit intro \x. list_ind(x;nil;h,t,v.h@v)
```

```
E                              E_star
DEF E                          DEF E*

l                              R_of_E
DEF [<list element>]           DEF R(E)

append                         append_
DEF <A list> @ <A list>        THM >> E* -> E* -> E*

flatten                        flatten_
DEF flatten(<(A list) list>)   THM >> E* list -> E*

contained                      eq
DEF <R(E)> < <R(E)>            DEF <R(E)> = <R(E)>

plus                           plus_
DEF <R(E)> + <R(E)>            THM >> R(E) -> R(E) -> R(E)

concat                         concat_
DEF <R(E)> <R(E)>              THM >> R(E) -> R(E) -> R(E)

star                           star_
DEF <R(E)>*                    THM >> R(E) -> R(E)
```

Figure 11.7: Definitions of Objects for the Theory of Regular Sets

```
append_lemma_1
THM >> All x:E*. x@nil = x in E*

append_lemma_2
THM >> All h:E. All t,y:E*. (h.t)@y = h.(t@y) in E*

append_assoc
THM >>All u,v,w:E*. (u@v)@w = u@(v@w) in E*

flatten_lemma_1
THM >> All x:E*. flatten([x]) = x in E*

contained_lemma_1
THM >>All r,s,t:R(E). r<s => s<t => r<t

star_lemma_1
THM >> All s:R(E). s*(nil)

star_lemma_2
THM >> All r:R(E). r<r*

star_lemma_3
THM >> All s:R(E). All u,v:E*.
          s(u) => s*(v) => s*(u@v)

star_lemma_3p5
THM >>All s:R(E).All u:E*.
         s*(u)
         => All y:E* list .
         list_ind(y;true;h,t,v.s(h)&v)
         => s*(flatten(y)@u)

star_lemma_4
THM >> All s:R(E). All p,q:E*.
          s*(p) => s*(q) => s*(p@q)
```

Figure 11.8: Basic Lemmas in the Theory of Regular Sets

```
lemma_1
THM >> All r,s:R(E). r+s < r*s*

lemma_2
THM >> All r,s:R(E). r<s => r*<s*

lemma_2p5
THM >> All r,s,t:R(E). r<t* & s<t* => rs<t*

lemma_3
THM >> All s:R(E). s**<s*

lemma_4
THM >> All r,s:R(E). r*s*<(r+s)*

Theorem
THM >> All r,s:R(E). (r*s*)* = (r+s)*
```

Figure 11.9: Lemmas and a Theorem from the Theory of Regular Sets

The definitions of containment, "<", and equality are what one would expect, given the definition of regular sets.

```
<r1:R(E)> < <r2:R(E)> == All x:E*. (<r1>)(x) => (<r2>)(x)

<r1:R(E)> = <r2:R(E)> == <r1> < <r2> & <r2> < <r1>
```

The following three proof tops give the definitions of $r + s$, rs, and r^*, respectively. (Informally, plus corresponds to union, juxtaposition to forming all possible concatenations of a word from the first set with a word from the second set, and star to forming all possible concatenations of words from the set.)

```
>> R(E) -> R(E) -> R(E)
BY explicit intro \r.\s.\x. ( r(x) | s(x) )

>> R(E)->R(E)->R(E)
BY explicit intro
    \r.\s.\x. Some u,v:E*.
      u@v=x in E* & r(u) & s(v)

>> R(E) -> R(E)
BY explicit intro
    \r.\x. Some y:E* list.
```

```
          list_ind(y;true;h,t,v.r(h)&v)
          & flatten(y)=x in E*
```

Note the use of `list_ind` to define a *predicate* recursively.

The presentation concludes with a few shapshots that should give an idea of what the proofs in this library are like. First, consider a few steps of the proof of `star_lemma_3p5`, which states that if u is in set $s*$ then for all sets of words in s the concatenation of all words in the set, followed by u, is also in $s*$. The major proof step, done after the goal has been "restated" by moving things into the hypothesis list, is induction on the list `y`.

```
,-----------------------------------------------,
|* top 1 1 1                                    |
|1. s:(R(E))                                    |
|2. u:(E*)                                      |
|3. (s*(u))                                     |
|4. y:(E* list )                                |
|>> ( list_ind(y;true;h,t,v.s(h)&v)             |
|   => s*(flatten(y)@u))                        |
|                                               |
|BY elim y new p,h,t                            |
|                                               |
|1* 1...4.                                      |
|   >> ( list_ind(nil;true;h,t,v.s(h)&v)        |
|      => s*(flatten(nil)@u))                   |
|                                               |
|2* 1...4                                       |
|   5. h:E*                                     |
|   6. t:(E* list )                             |
|   7. p:( list_ind(t;true;h,t,v.s(h)&v)        |
|      => s*(flatten(t)@u))                     |
|   >> ( list_ind(h.t;true;h,t,v.s(h)&v)        |
|      => s*(flatten(h.t)@u))                   |
'-----------------------------------------------'
```

The first subgoal is disposed of by doing some computation steps on subterms of the consequent of the conclusion. The second subgoal is restated, and the last hypothesis is reduced using a tactic.

```
,-----------------------------------------------,
|* top 1 1 1 2 1                                |
|1...7.                                         |
|8. ( list_ind(h.t;true;h,t,v.s(h)&v) )         |
|>> ( s*(flatten(h.t)@u))                       |
|                                               |
|BY  compute_hyp_type                           |
|                                               |
|1* 1...7.                                      |
|   8. s(h)&list_ind(t;true;h,t,v.s(h)&v)       |
|   >> ( s*(flatten(h.t)@u))                    |
'-----------------------------------------------'
```

The rest of the proof is just a matter of expanding a few definitions and doing some trivial propositional reasoning.

Now consider `Theorem`. The proof is abstract with respect to the representation of regular sets; lemmas `lemma_1` to `lemma_4` and the transitivity of containment are all that is needed. First, the two major branches of the proof are set up.

```
,-----------------------------------------------,
|* top 1                                        |
|1. r:(R(E))                                    |
|2. s:(R(E))                                    |
|>> ( (r*s*)*=(r+s)*)                           |
|                                               |
|BY intro                                       |
|                                               |
|1* 1...2.                                      |
|   >> ((r*s*)*<(r+s)*)                         |
|                                               |
|2* 1...2.                                      |
|   >> ((r+s)*<(r*s*)*)                         |
'-----------------------------------------------'
```

The rest of the proof is just lemma application (using lemma application tactics); for example, the rule below is actually a definition whose right–hand side calls a tactic which computes from the goal the terms with which to instantiate the named lemma, and then attempts (successfully in this case) to finish the subproof.

```
+-----------------------------------------+
|EDIT THM Theorem                         |
|-----------------------------------------|
|* top 1 1 1 1 1                          |
|1. r:(R(E))                              |
|2. s:(R(E))                              |
|3. ( r*s*<(r+s)* => r*s**<(r+s)**)       |
|>> (r*s*<(r+s)* )                        |
|                                         |
|BY Lemma lemma_4                         |
|                                         |
+-----------------------------------------+
```

11.5 Real Analysis

This section presents Nuprl formalizations of constructive versions of some of the most familiar concepts of real analysis. The account here is brief; more on this subject will be found in the forthcoming thesis of Howe [Howe 86].

We begin with a basic type of the positive integers, two definitions that make terms involving `spread` more readable and an alternative definition of `some` which uses the set type instead of the product.

```
Pos ==
  {n:int|0<n}

let <x:var>,<y:var>=<t:term> in <tt:term> ==
   spread(<t>;<x>,<y>.<tt>)

let <w:var>,<x:var>,<y:var>,<z:var>=
              <t1:term>,<t2:term> in <t3:term> ==
   let <w>,<x>=<t1> in let <y>,<z>=<t2> in <t3>

Some <x:var>:<T:type> where <P:prop> ==
   { <x>:<T> | <P> }
```

Figure 11.10 lists a few of the standard definitions involved in the theory of rationals. Note that the rationals, `Q`, are a quotient type; therefore, as explained in the section on quotients in chapter 10, we must use the squash operator ($||\ldots||$) in the definition of $<$ over `Q`.

We adopt Bishop's formulation of the real numbers as *regular* (as opposed to *Cauchy*) sequences of rational numbers. With the regularity approach a real number is just a sequence of rationals, and the regularity condition (see the definition of `R` below) permits the calculation of arbitrarily close rational approximations to a given real number. With the usual approach a

```
preQ == (int#Pos)

Q_eq(<x:preQ>,<y:preQ>) ==
   let p,q,s,t=<x>,<y> in p*t=q*s in int

Q == (x,y):preQ//Q_eq(x,y)

||<T:type>|| == { 0=0 in int | <T> }

<x:Q> < <y:Q> in Q ==
   let p,q,r,s = <x>,<y> in ||p*s<q*r||

<x:Q>+<y:Q> == let p,q,r,s = <x>,<y> in <p*s+q*r,q*s>

|<x:Q>| == let p,q = <x> in less(p;0;<-p,q>;<p,q>)

<x:Q> <_ <y:Q> in Q == <x> < <y> in Q | <x> = <y> in Q

<m:int>/<n:Pos> == <<m>,<n>>
```

Figure 11.10: Defining the Rational Numbers

real number would actually have to be a pair comprising a sequence *and* a function giving the convergence rate. Figure 11.11 lists the definition of the reals and functions and predicates involving the reals. The definition of $<$ is a little different than might be expected, since it has some computational content, i.e., a positive lower bound to the difference of the two real numbers.

The definition of $\leq$ (`<_`) in figure 11.11 is squashed since it is a predicate over a quotient type. However, in the case of the real numbers the use of the squash operator does not completely solve the problem with quotients. For example, if `X` is a discrete type (i.e., if equality in `X` is decidable) then there are no nonconstant functions in `R/= -> X`. In particular, the following basic facts *fail* to hold.

```
All x:R/=. All n:int. Some a:Q. |x-a| <_ 1/n in Q

All x,y,z:R/=. x<y in R/= => ( z<y in R/= | x<z  in R/=)
```

We are therefore precluded from using the quotient type here as an abstraction mechanism and will have to do most of our work over `R`. `R/=` can still be used, however, to gain access to the substitution and equality rules, and it is a convenient shorthand for expressing concepts involving equality.

Consider the computational aspects of the next two definitions. Knowing that a sequence of real numbers converges involves having the limit and the

```
R == { x:(Pos->Q) | All m,n:Pos
                    |x(m)-x(n)| <_ 1/m+1/n in Q }

R_eq(<x:R>,<y:R>) ==
   All n:Pos. |<x>(n)-<y>(n)| <_ 2/n in Q

R/= == (x,y):R//R_eq(x,y)

<x:Q>* == \n.<x>

<x:R>+<y:R> == \ n.( <x>(n) + <y>(n) )

<x:R> <_ <y:R> in R/= ==
   || All n:Pos. <x>(n)-<y>(n) <_ 1/n in Q ||

|<x:R>| == \n.|<x>(n)|

<x:R> < <y:R> in R/= ==
   Some n:Pos where 1/n <_ <y>-<x> in R/=
```

Figure 11.11: Defining the Real Numbers

function which gives the convergence rate. Notice that we have used the set type (in the definition of `some ... where`) to isolate the computationally interesting parts of the concepts.

```
<x:Pos->R> is Cauchy ==
    All k:Pos.
      Some M:Pos where
             All m,n:Pos.
                M <_ m in int => M <_ n in int =>
                |<x>(m)-<x>(n)| <_ (1/k)* in R/=

<x:Pos->R> converges ==
    Some x:R/=. All k:Pos.
      Some N:Pos where
              All n:int. N <_ n in int =>
                 |(<x>)(n)-x| <_ (1/k)* in R/=
```

Figure 11.12 defines the "abstract data type" of compact intervals. Following is the (constructive) definition of continuity on a compact interval. A function in `R+ -> R+` which inhabits the type given as the definition of continuity is called a *modulus of continuity*.

```
CompactIntervals ==
     I_set:U1
   # a:R
   # b:R
   # a <_ b in R
     & I_set = { z:R | a<_z in R/= & z<_b in R/= }  in U1

|<I:CompactIntervals>| == fst(<I>)

a_of <I:CompactIntervals> == fst(snd(<I>))

b_of <I:CompactIntervals> == fst(snd(snd(<I>)))
```

Figure 11.12: Defining Compact Intervals

```
R+ == { x:R | 0<x in R/= }

<f:|I|->R> continuous on <I:CompactIntervals> ==
   All eps:R+.
      Some del:R+ where All x,y:|<I>|.
      |x-y| <_ del in R/=  =>  |<f>(x)-<f>(y)| <_ eps
```

It is now a simple matter to state a constructive version of the intermediate value theorem.

```
>> All I:CompactIntervals. All f:|I|->R.
      f continuous on I  &  f(a_of I) < 0 in R
         &  0 < f(b_of I) in R
      => All eps:R+.
            Some x:|I|. |f(x)| < eps in R
```

A proof of the theorem above will yield function resembling a root–finding program.

11.6 Denotational Semantics

This section presents a constructive denotational semantics theory for a simple program language where meaning is given as a stream of states. The following definitions and theorems illustrate this library.

The data type `Program` is built from a dozen type definitions like those which follow.

```
Atomic_stmt  == Abort | Skip | Assignment

Program      == (depth:N # Statement(depth))

Statement(<d>) ==
     ind(<d>; x,y.void; void; x,T. Atomic_stmt |
         (T#T) *concat* | (expr#T) *do loop* |
         expr#T#T *if then else*)
```

The type for stream of states, `S`, is defined below. Note that the `ind` form in the definition of `Statement` approximates a type of trees, while `N->State` approximates a stream of states. The intended meaning for `S` is that given an initial state, its value on `n` is the program state after executing `n` steps.

```
State*   == Identifier -> Value

Done     == {x:atom | x="done" in atom}

State    == State* | Done
<st:State> terminated ==
     decide(<st>; u.void; u.int)

S        ==
     {f: State -> N -> State |
     All a:State, n:N.
f(a)(n) terminated => f(a)(n+1) terminated}
```

Figure 11.13 presents the various meaning functions as extracted objects. Using these definitions, one can define a meaning function, `M`, for programs. This is done in figure 11.14 in the definition of `M`, the meaning function for programs. With `M` one can reason about programs as well as executing them directly using the evaluation mechanism.

```
Abort_thm:
>> Some s:S. All a:State, n:N. ~(s(a)(n) terminated)

Mabort == 1st(term_of(Abort_thm))

<P:proposition> where <x:var>=<t:term> in <T:type> ==
     (<x>:{x:<T>|<x>=<t> in <T>} # (<P>))

T    == 1
F    == 0

Bool == {x:int | (x=T in int)|(x=F in int)}

if <b:Bool> then <s:value> else <t:value> ==
     int_eq(<b>; T; <s>; <t>)

<st:State> stopped == decide(<st>; u.F; u.T)

mu <x:var>< <k:bound>[<P:N->Bool>] ==
     ind(<k>; a,b.0; <k>; a,b.if <P>(<k>-a) then <k>-a else b)

Concat_thm:
>> All s1,s2:S. Some s:S. All a:State, n:N.
     s(a)(n) =
        decide( s(a)(n); y.s2(a0)(n0); y.inr(y)) in State
     where a0 = s1(a)(n0) in State
     where n0 = mu x<n [\x.s1(a)(x) stopped] -1 in N

<s1>; <s2> == 1st(term_of(Concat_thm)(<s1>)(<s2>))

If_thm:
>> All R:State->Bool, s1,s2:S. Some s:S.
     s = (\a.if R(a) then s1(a) else s2(a)) in S

IF <R> THEN <s> ELSE <t> ==
     1st(term_of(If_thm)(<R>)(<s>)(<t>))

Do_thm:
>> All R:State->Bool, body:S. Some s:S.
     s = IF R THEN (body; s) ELSE Mskip     in S

WHILE <R> DO <body> == 1st(term_of(Do_thm)(<R>)(<body>))
```

Figure 11.13: A Constructive Theory of Denotational Semantics

```
Meaning_thm:
>> Some M:Program->S. All p:Program.
    M(p) = decide(2nd p;
        q. Matomic(q);
        q. decide(q;
            r. ( M(d,1st r); M(d,2nd r) );
            r. decide(r;
                t. WHILE (1st t) DO M(d,2nd t);
                t. IF (1st t) THEN M(d, 1st 2nd t)
                              ELSE M(d, 2nd 2nd t))))
    in S
    where d = (1st p)-1

M == 1st(term_of(Meaning_thm))
```

Figure 11.14: A Constructive Meaning Function for Programs

Chapter 12

Recursive Definition

For anyone doing mathematics or programming, recursive definition needs no motivation; its expressiveness, elegance and computational efficiency motivate us to include forms of it in the Nuprl logic. Current work on extending the logic involves three type constructors: `rec`, the *inductive type constructor*, permitting inductive data types and predicates; `inf`, the *lazy type constructor*, permitting infinite objects; and $\sim$>, the *partial function space constructor*, permitting recursively defined partial functions. This chapter gives the extensions to the system necessary for inductive types and for partial function types; for detailed presentations of these two types see [Constable & Mendler 85].

12.1 Inductive Types

As an introduction to the `rec` types, consider the inductive type of integer trees, defined informally as

$$let\ z\ be\ int \mid (z \# z).$$

In the language of `rec` types this type may be defined as

```
rec(z. int | z#z).
```

Its elements include

```
inl 2,
inr <inl 3,inl 5> and
inr <inr <inl 7,inl 11>,inl 13>.
```

An inductive type may also be parameterized; generalizing the above definition of binary integer trees to general binary trees over a specified type, the example rephrased as

$$let\ z(x)\ be\ defined\ as\ x \mid (z(x)\#z(x));\ consider\ z(int)$$

is denoted

```
rec(z,x. x | z(x)#z(x); int),
```

and the predicate function *dom*,

$$dom(x) = \begin{cases} true & \text{if } f(x) = 0 \\ dom(x+1) & \text{if } f(x) \neq 0 \end{cases},$$

asserting f has a root $\geq x$ is denoted

```
\x. rec(dom,x. int_eq(f(x); 0; true; dom(x+1)); x).
```

The elim form, `rec_ind`, is analogous to the `list_ind` and integer `ind` forms. If `t` is of type `rec(z. int | z#z)` then the following term computes the sum of the values at `t`'s leaves.

```
rec_ind(t;  sum,x.
  decide(x;
    leaf. leaf;
    pair. spread(pair; left_son,right_son.
            sum(left_son)+sum(right_son))))
```

The simpler `rec(`$z.T$`)` form is not formally part of the extension since it can be mimicked by `rec(`z`,`x`.` $T[$`z(0)`$/z]$`;0)`, for example, so consider any `rec(`$z.T$`)` term in what follows to be just an abbreviation. Inductive types can also be defined in a mutually recursive fashion, but we will not pursue that possibility here.

Expressiveness and Elegance

The transfinite W–type of well–founded trees, $(\mathbf{W}x \in A)B$, used to represent Brouwer ordinals is inexpressible in the Nuprl logic discussed in the earlier chapters but can be represented in the logic extended by `rec` types as `rec(w.` x`:`A`#(`B`->w))`. The data type for programs given in chapter 11,

```
d:N#ind(d; x,y.void; void; x,T. Atomic_Stmt | (T#T) |
                                (expr#T) | expr#T#T),
```

can be written more elegantly as

```
rec(T. Atomic_Stmt | (T#T) | (expr#T) | expr#T#T).
```

The `list` type constructor is now redundant because A `list` can be expressed as `rec(l. NIL |` A`#l)`.

Details of the Extension

The following modifications will add inductive types to Nuprl.

Computation System Modification

Add terms

> `rec(`z`,`x`.`T`;`a`)`
> `rec_ind(`r`; `v`,`u`.`g`)`

where u, v, x and z range over variables, a, g, r and T range over terms, and T is restricted as follows:

> No instance of z bound by `rec` may occur in the domain type of a function space, in the argument of a function application or in the principal argument(s) of the remaining elimination forms.[1]

Thus `(A#`z`)->B` or `f(inr `z`)` or `spread(`z`; a,b.a->int)` cannot be subexpressions of T. When closed, `rec` terms are canonical and `rec_ind` terms are noncanonical. These noncanonical terms are redices with principal argument r and contracta

> $g[r, \backslash u$`.rec_ind(`u`; `v`,`u`.`g`)`$/u,\ v]$,

when r is canonical. The rules for binding variables are:

- In `rec(`z`,`x`.`T`;`a`)` the z and x in front of the dot and any free occurrences of z or x in T become bound.
- In `rec_ind(`r`; `v`,`u`.`g`)` the u and v in front of the dot and any free occurrences of u or v in g become bound.

Inductive Type Proof Rules

The proof rules will ensure that the following two assertions hold.

> `rec(`x`,`y`.`A`;`b`)` *type* $\Leftrightarrow$
> $\exists B, k.\ b \in B$ & x`:(`B`->U`k`)->`y`:`B`->`A *type*
>
> $t \in$ `rec(`x`,`y`.`A`;`b`)` $\Leftrightarrow$ `rec(`x`,`y`.`A`;`b`)` *type*
> & $t \in A[\backslash y$`.rec(`x`,`y`.`A`;`y`),\ b/x,\ y]$

That is, `rec(`x`,`y`.`A`;`b`)` is a type exactly when there is a type B and universe level k such that b is in B and for all x which, when instantiated with a value from B, yield a type at universe k, and y of type b, A (with x and y potentially free) is a type. Also, t is of type `rec(`x`,`y`.`A`;`b`)` exactly when `rec(`x`,`y`.`A`;`b`)` is a type and t is a member of the "unwinding" of the recursive type.

[1] This will ensure that the definitions of relations $T = T'$ and $t = t' \in T$ are sensible. A more liberal (and complex) restriction is given in [Constable & Mendler 85].

Formation

A universe intro rule such as "`>> U`i `by intro rec`" cannot be phrased easily in the refinement logic style because of the syntactic restriction on T in the extracted term `rec(`z`,`x`.`T`;`a`)`. One could give an approximate solution (as was done for set elim), but here we settle for no rule at all, thereby forcing the use of the explicit intro rule.

H `>> rec(`z`,`x`.`T`;`a`) in U`i `by intro using` A
z`:`A`->U`i`,`x`:`A `>>` T `in U`i
`>>` a `in` A

Intro

H `>> rec(`z`,`x`.`T`;`a`)` `by intro at U`i $[EXT\ \ r]$
`>>` $T[\backslash x$`.rec(`z`,`x`.`T`;`x`),` $a/z,\ x]$ $[EXT\ \ r]$
`>> rec(`z`,`x`.`T`;`a`) in U`i

H `>>` r `in rec(`z`,`x`.`T`;`a`)` `by intro at U`i
`>>` r `in` $T[\backslash x$`.rec(`z`,`x`.`T`;`x`),` $a/z,\ x]$
`>> rec(`z`,`x`.`T`;`a`) in U`i

H `>> rec_ind(`p`;` h`,`p_1`.`g`)` `in` $G[p/p_2]$
`by intro [over` p_2`.`G`] using` y`:`A`#rec(`z`,`x`.`T`;`y`)` `new` $Z[,p_2]$
p_2`:(`y`:`A`#`Z`(`y`))->(`p_2 `in` y`:`A`#rec(`z`,`x`.`T`;`y`)),`
h`:`p_2`:(`y`:`A`#`Z`(`y`))->`G`,`
p_1`:`y`:`A`#(`$T[Z, y/z, x]$`) >>` g `in` $G[p_1/p_2]$
`>>` p `in` y`:`A`#rec(`z`,`x`.`T`;`y`)`

Elim

H`,`r`:rec(`z`,`x`.`T`;`a`),`H' `>>` G `by elim` r $[EXT\ \ t]$
H`,`r`:`$T[\backslash x$`.rec(`z`,`x`.`T`;`x`),` $a/z,\ x]$ `,`H' `>>` G $[EXT\ \ t]$

H`,`r`:rec(`z`,`x`.`T`;`a`)>>` G`[<`a`,`r`>/`p`]` $[EXT$ `rec_ind(<`a`,`r`>;`p`,`h`.`g`)]`
`by elim` r `over` y`:`A`#rec(`z`,`x`.`T`;`y`)` `[using` p`.`G`]` `new` $h, Z[,p]$
p`:(`y`:`A`#`Z`(`y`))->(`p `in` y`:`A`#rec(`z`,`x`.`T`;`y`)),`
h`:`p`:(`y`:`A`#`Z`(`y`))->`G`,`
p`:`y`:`A`#(`$T[Z, y/z, x]$`) >>` G $[EXT\ \ g]$
`>>` a `in` A

Computation

H `>> rec_ind(`r`;` h`,`r_1`.`g`)=`t `in` G `by reduce`
`>>` $g[r, \backslash r_1$`.rec_ind(`r_1`;`h`,`r_1`.`g`)/`r_1`,` $h]$`=`t `in` G

```
1. f:N->N
2. r:D(0)
>> some y:N. f(y)=0 in N    by elim r over y:N#D(y) new h,Z,p

    2.    p:(y:N#Z(y))->(p in y:N#D(y))
    3. h: p:(y:N#Z(y))->(some y:N. f(y)=0 in N)
    4. p: y:N#int_eq(f(y); 0; true; Z(y+1))
    >> some y:N. f(y)=0 in N
            by elim 4 new y,ax, then
            cases f(y)=0 in N | ~(f(y)=0 in N)

case1 6. f(y)=0 in N
      >> some y:N. f(y)=0 in N     by intro y

case2 5. ax:int_eq(f(y); 0; true; Z(y+1))
      6. ~(f(y)=0 in N)
      >> some y:N. f(y)=0 in N     by computation 5

        3. h: p:(y:N#Z(y))->(some y:N. f(y)=0 in N)
        4. y:N
        5. ax:Z(y+1)
        6. ~(f(y)=0 in N)
        >> some y:N. f(y)=0 in N     by elim h on <y+1,ax>
```

Figure 12.1: A Sample Proof Using Recursion

Example Proof

The *dom* predicate defined earlier in the chapter is used here to demonstrate induction. Given the definitions

```
~<T>== (<T>) -> void
some <v>:<A>.<P>== <v>:(<A>)#(<P>)
N== {n:int| ~(n<0)}
true== (0 in int)
D(<x>)== rec(dom,x. int_eq(f(x); 0; true; dom(x+1)); <x>)
```

figure 12.1 sketches the proof of

```
f:N->N, r:D(0) >> some y:N. f(y)=0.
```

Although `D(0)` is inhabited only by `axiom`, evaluating the extracted term produces a root.

12.2 Partial Function Types

We handle partial functions by viewing them as total functions over their domains of convergence. To do so we introduce `dom` terms, which capture the notion of a domain of convergence. Thus, for a partial function application to be sensible one must prove that the domain of convergence predicate is true for the argument, i.e., that the argument is in the domain of convergence of the function. The introduction of `dom` terms introduces additional proof obligations into the rules for reasoning about partial functions.

As an example, the partial function μ producing the smallest root $\geq x$ of f,

$$\mu(x) = \begin{cases} x & \text{if } f(x) = 0 \\ \mu(x+1) & \text{if } f(x) \neq 0 \end{cases}$$

has a domain of convergence defined by the predicate μ' as follows.

$$\mu'(x) = \begin{cases} \text{true} & \text{if } f(x) = 0 \\ \mu'(x+1) & \text{if } f(x) \neq 0 \end{cases}$$

Thus μ is a total function in $\{x \in nat | \mu'(x)\} \rightarrow nat$.

In our notation the type of partial functions from A to B is $A\sim>B$, the application of partial function `f` to `a` is written `f[a]`, recursive functions are expressed as terms of the form `fix(`f`,`x`.`b`)`, so function μ of the preceding example is

```
MU == fix(mu,x. int_eq(f(x); 0; x; mu[x+1])),
```

and μ' is `dom(MU)` , an elim form which turns out to have value

```
\x.rec(mu',x. int_eq(f(x); 0; true; mu(x+1)); MU; x).
```

The domain predicate is derived later; in general, if $f \in A\sim>B$ then `\`x`.`f`[`x`]` $\in$ `{`x`:`A`|dom(`f`)(`x`)}->`B.

As a second example, the "91" function,

$$F(x) = \begin{cases} x - 10 & \text{if } x > 100 \\ F(F(x+11)) & \text{if } x \leq 100 \end{cases}$$

has domain

$$F'(x) = \begin{cases} true & \text{if } x > 100 \\ F'(x+11) \; and \; F'(F(x+11)) & \text{if } x \leq 100 \end{cases}$$

In our notation F and F' are

```
91== fix(F,x. less(100; x; x-10; F[F[x+11]])

\x.rec(F',x.
   less(100; x; true; c:F'(x+11)#F'(F[x+11]); 91; x)
```

Notice that the `rec` forms defined in the preceding section must be extended to include the simultaneous definition of the recursive function. The new proof rules for these `rec` terms are only slight variants of the given rules, so they are not listed here.

Details of the Extension

Computation System Modification

The domain predicate captures a call–by–value order of computation, so the entire computation system must be adjusted slightly to reflect this. In particular, computing the value of a function application (total or partial) dictates that function and argument are normalized before beta reduction, both subterms in pairs are normalized, and for injections into disjoint sums the subterm is normalized. In this extension direct computation rules are not present.

To add $\sim$> types to the logic we add terms

`fix(`f`,`x`.`b`)`
A`~>`B
`dom(`t`)`
t`[`a`]`

where f and x range over variables and a, b, t, A and B range over terms. When closed, the first two kinds of terms are canonical, and the last two are noncanonical.

Let φ stand for `fix(`f`,`x`.`b`)` henceforth. Each closed term φ`[`a`]` is a redex with contractum $b[\varphi,\ a/f,\ x]$, and each closed term `dom(`φ`)` is a redex with contractum

$\backslash x$.`rec(`f'`,`x`.`$\mathcal{E}[\![$`b`$]\!]$`;`φ `;`x`)`,

where $\mathcal{E}$ is defined below. The rule for binding variables is as follows.

- In `fix(`f`,`x`.`b`)`, the f and x in front of the dot and any free occurrences of f or x in b become bound.

Domain Predicate

We define $\mathcal{E}$, a syntactic transformation on terms, as follows.

- $\mathcal{E}[\![f$`[`t`]`$]\!] = \mathcal{E}[\![t]\!]\#f$`'(`$t$`)`, if f had been bound by the the first variable in a surrounding `fix` term. Recall that f' is the characteristic function for the domain of f.
- $\mathcal{E}[\![t$`[`a`]`$]\!] = \mathcal{E}[\![t]\!]\#\mathcal{E}[\![a]\!]\#$`dom(`$t$`)(`$a$`)` if the first clause does not apply.

- $\mathcal{E}[\![t(a)]\!] = \mathcal{E}[\![t]\!]\#\mathcal{E}[\![a]\!]$
- $\mathcal{E}[\![\texttt{spread}(e;\ u,v.t)]\!] = \mathcal{E}[\![e]\!]\#\texttt{spread}(e;\ u,v.\mathcal{E}[\![t]\!])$, and similarly for the other elimination forms.
- $\mathcal{E}[\![\texttt{inl}(a)]\!] = \mathcal{E}[\![a]\!]$
- $\mathcal{E}[\![\texttt{<}a\texttt{,}b\texttt{>}]\!] = \mathcal{E}[\![a]\!]\#\mathcal{E}[\![b]\!]$
- $\mathcal{E}[\![t]\!] = \texttt{true}$, for the remaining terms.

The #'s used here are dependent products and therefore require a left–to–right evaluation order in the predicate.

Partial Function Type Proof Rules

In the following $\mathcal{E}'$ is just $\mathcal{E}$ with the first clause of the definition omitted and φ' is $\texttt{fix}(f,x.b')$.

Formation

H >> $A\sim\texttt{>}B$ in Ui by intro
>> A in Ui
>> B in Ui

Intro

H >> φ in $A\sim\texttt{>}B$ by intro at Ui
>> $A\sim\texttt{>}B$ in Ui
f:$A\sim\texttt{>}B$, x:A >> $\mathcal{E}'[\![b]\!]$ in Ui
f:$A\sim\texttt{>}B$, x:A, $\mathcal{E}'[\![b]\!]$ >> b in B

H >> φ = φ' in $A\sim\texttt{>}B$ by intro at Ui
>> φ in $A\sim\texttt{>}B$
>> φ' in $A\sim\texttt{>}B$
>> {x:A | dom(φ)(x)} = {x:A | dom(φ')(x)} in Ui
x:{x:A | dom(φ')(x)} >> φ[x] = φ'[x] in B

H >> g[a] in B by intro using $A\sim\texttt{>}B$
>> g in $A\sim\texttt{>}B$
>> a in {x:A | dom(g)(x)}

H >> dom(g) in A->Ui by intro using $A\sim\texttt{>}B$
>> g in $A\sim\texttt{>}B$
>> $A\sim\texttt{>}B$ in Ui

Elim

$H, g{:}A \sim\!> B, H'$ >> G by elim g on a new y
$H, g{:}A \sim\!> B, H'$ >> a in $\{x{:}A \mid \mathrm{dom}(g)(x)\}$
$H, g{:}A \sim\!> B, H', y{:}B;\ y{=}g[a]$ in B >> G

Computation

H >> $\varphi[a]$ = t in B by reduce
>> $b[\varphi, a/f, x]$ = t in B

Appendix A: Summary of ML Extensions for Nuprl

General Notes

All universe levels must be strictly positive. Unless otherwise stated the tokens ‘`nil`‘ and ‘`NIL`‘ may be used as identifiers to indicate that no identifier should label the new hypothesis.

Refinement Functions

`refine: rule → tactic.` This function refines a proof according to a given rule.

`refine_using_prl_rule: tok → tactic.` This function parses the token as a rule in the context of the given proof. The proof is then refined via this rule.

Rule Constructors

See chapter 8 for a description of the rule constructors.

Rule Destructors

`rule_kind: rule → tok.` Returns the kind of the rule. Note that at present this is in the internal form of the rule name. There are also predicates of the form `is_universe_intro_void` that correctly translate from the internal names of rules and the names of the rule constructors (as listed above with “is_” prepended). There are also destructors for each of the rules that correspond to the constructor. The names are of the form “destruct_universe_intro_void”.

Term Destructors

`term_kind: term -> tok.` Returns the kind of the term. Possible results are: `UNIVERSE`, `VOID`, `ANY`, `ATOM`, `TOKEN`, `INT`, `NATURAL-NUMBER`, `MINUS`, `ADDITION`, `SUBTRACTION`, `MULTIPLICATION`, `DIVISION`, `MODULO`, `INTEGER-INDUCTION`, `LIST`, `NIL`, `CONS`, `LIST-INDUCTION`, `UNION`, `INL`, `INR`, `DECIDE`, `PRODUCT`, `PAIR`, `SPREAD`, `FUNCTION`, `LAMBDA`, `APPLY`, `QUOTIENT`, `SET`, `EQUAL`, `AXIOM`, `VAR`, `BOUND-ID`, `TERM-OF-THEOREM`, `ATOMEQ`, `INTEQ`, `INTLESS`.

There are corresponding predicates such as `is_universe_term` and corresponding constructors such as `make_function_term` (which has type `tok → term → term → term`).

`destruct_universe: term → int.` The integer is the universe level.

`destruct_any: term → term.`

`destruct_token: term → tok.`

`destruct_natural_number: term → int.`

`destruct_minus: term → term.`

`destruct_addition: term → (term # term).`
The results are the left and right terms.

`destruct_subtraction: term → (term # term).`

`destruct_multiplication: term → (term # term).`

`destruct_division: term → (term # term).`

`destruct_modulo: term → (term # term).`

`destruct_integer_induction: term → (term # term # term # term).`

The results are the value, down term, base term and up term.

`destruct_list: term → term.` Returns the type of the list.

`destruct_cons: term → (term # term).` Returns the head and tail.

`destruct_list_induction: term → (term # term # term).`
Returns the value, base term and up term.

`destruct_union: term → (term # term).`
Results in the left and right types.

`destruct_inl: term → term.`

`destruct_inr: term → term.`

`destruct_decide: term → (term # term # term).`
The result is the value, the left term and the right term.

`destruct_product: term → (tok # term # term).`
The result is the bound identifier, the left term and the right term.

`destruct_pair: term → (term # term).`
Returns the left term and the right term.

`destruct_spread: term → (term # term).`
Returns the value and the term.

`destruct_function: term → (tok # (term # term)).`
The result is the bound identifier, the left term and the right term.

`destruct_lambda: term → (tok # term).`
Returns the bound identifier and the term.

`destruct_apply: term → (term # term).`
Returns the function and the argument.

`destruct_quotient: term → (tok # (tok # (term # term))).`
Returns the two bound identifiers, the left type and the right type.

`destruct_set: term → (tok # (term # term)).`
Returns the bound identifier, the left type and the right type.

`destruct_var: term → tok.`

`destruct_equal: term → ((term list) # term).`
Returns a list of the equal terms and the type of the equality.

`destruct_less: term → (term # term)).`
Returns the left term and the right term.

`destruct_bound_id: term → ((tok list) # term).`
Returns a list of the bound identifiers and the type.

`destruct_term_of_theorem: term → tok.`
Returns the name of the theorem.

`destruct_atomeq: term → (term # (term # (term # term))).`
Returns the left term, right term, if term and else term.

`destruct_inteq: term → (term # (term # (term # term))).`
Returns the left term, right term, if term and else term.

`destruct_intless: term → (term # (term # (term # term))).`
Returns the left term, right term, if term and else term.

`destruct_tagged: term → (int # term).`
The integer returned is zero if the tag is `*`.

For each term type there is a corresponding constructor that is the curried inverse to the destructor given above. For example,

`make_inteq_term: term → term → term → term → term.`

Auto–Tactic

`set_auto_tactic: tok → void.`
Set auto–tactic to be the ML expression represented by the token.

`show_auto_tactic: void → tok.`
Displays the current setting of the auto–tactic.

Miscellaneous Functions

`mm: tactic.`
Prints a reasonable representation of the proof into the snapshot file without modifying the proof. It is intended to be used as a transfomation tactic.

`term_to_tok: term → tok.` Converts a term into a token representation.

`print_term: term → void.` Displays a term.

`rule_to_tok: rule → tok.`

`print_rule: rule → void.` Displays a representation of a rule.

`declaration_to_tok: declaration → tok.`

`print_declaration: declaration → void.` Displays a declaration.

`main_goal_of_theorem: tok → term.`
Returns the term which is the conclusion of the top node of the named theorem.

`extracted_term_of_theorem: tok → term.`
Returns the term extracted from the named theorem.

`map_on_subterms: (term → term) → term → term.`
Replaces each immediate subterm of the term argument by the result of applying the function argument to it.

`free_vars: term → term list.`

`remove_illegal_tags: term → term.` Takes a term and removes the smallest number of tags needed to make the term have a tagging which is legal with respect to the direct computation rules. (See appendix C for a definition of legal tagging.)

`do_computations: term → term.`
Returns the result of performing the computations indicated by the tags in the argument. (See appendix C for a description of the direct computation rules).

`no_extraction_compute: term → term.`
Computes (in the sense of the direct computation rules) the term as far as possible, without treating `term_of` terms as redices; these are treated in the same way as variables.

`loadt: tok → void.`
Loads the named file into the ML environment. The name given must be the name, or path name, of the file, with the extension (e.g., "`.ml`") removed; the appropriate version (binary, Lisp or ML source) will be loaded. Note that ML written in files can only mention other ML objects; in particular, Nuprl definitions cannot be instantiated, and the single quote cannot be used to form Nuprl terms.

`compilet: tok → void.`
Compiles the named file, putting the Lisp and binary code files in the same directory as the named file. The argument should be the full path name for the file with the extension removed.

Appendix B: Converting to Nuprl from Lambda–prl

This appendix discusses the differences in the Lambda–prl and Nuprl logics and user interfaces and ways to simulate Lambda–prl constructs in Nuprl.

Differences in the Logic

The Nuprl proof rules naturally extend the Lambda–prl proof rules and as such are organized along roughly the same lines. However, the logic of Nuprl is considerably more powerful than that of Lambda–prl, and the rules are correspondingly more complex. The Nuprl rules are classified into three broad categories: *introduction*, *elimination* and *equality*; the equality rules are further broken down into *computation rules* and *equality rules* for each of the term constructors. The introduction and elimination rules are analogous to those in Lambda–prl; the equality rules are a new feature of the Nuprl logic.

The differences between the logic of Lambda–prl and Nuprl can be briefly summarized as follows:

- Strengthening the type structure.
  The Lambda–prl logic is based on two primitive types, integers and lists of integers. The Nuprl logic extends this to a much richer type structure.

- Identifying propositions with types.
  In Lambda–prl propositions were specific syntactic entities. In Nuprl propositions are simply certain kinds of types. Demonstrating the truth of a proposition amounts to presenting an object of that type so that a proposition is treated as the type of its justifications.

- Well-formedness.
  In Lambda–prl well–formedness of propositions was checked by the system. The richness of the Nuprl type structure, however, precludes the possibility of automatically checking whether or not an expression is a type. By the identification of propositions with types this property extends to propositions as well.

- Explicit treatment of equality and simplification.
  In Lambda–prl equality and simplification were by and large taken care of by the `arith` rule, which handled simple arithmetic, substitutivity of equality and simplification of terms (such as `hd[1,2]=1`). The Nuprl logic, however, is sufficiently rich that such a treatment is no longer possible. Consequently there are rules for equality of terms and for computation.

We now discuss the effect of these aspects of the Nuprl logic on the formulation of the rules.

Introduction Rules

For each of the type constructors of the theory there are two introduction rules, one of the form H `>>` A and one of the form H `>>` a `in` A. For instance, we have

H `>>` $A \mid B$ `by intro at U`i `right`
  H `>>` A
  H`>>` B `in U`i

and

H `>>` `inl(`a`)` `in` $A \mid B$ `by intro at U`i
  H `>>` a `in` A
  H `>>` B `in U`i

These rules have essentially the same content to the extent that the judgements are completely reflected in the equality types, but the former leaves the inhabiting object, `inl(`a`)`, suppressed whereas the latter treats it explicitly. In the first rule the extracted object is `inl(`a`)`, while in the latter it is the token `axiom`. Intuitively, the first of the two is useful when we are regarding $A \mid B$ as the proposition expressing the disjunction of A and B; the latter is useful when we are thinking of $A \mid B$ as a data type, in which case the right–hand side above expresses the proposition that `inl(`a`)` is a member of the type $A \mid B$.

Since types, and hence propositions, are those terms which can be shown to inhabit a universe type, this duality extends to the type (proposition) formation rules — they are simply the introduction rules for the universe

types. Note, however, that the rules are organized in such a way that one does not need to construct a type explicitly before showing it to be inhabited. Each rule is sufficient to show not only that a type is inhabited but also that it is in fact a type. In most cases we go to no special trouble to achieve this, but certain rules contain subgoals whose only purpose is to assist in demonstrating that the consequent is a type. For instance, the second subgoal in the rules above exists solely to ensure that $A \mid B$ is a type. That A is a type is ensured by the first subgoal; the second provides the additional information necessary to form the union type.

Elimination Rules

There is also a dual relationship between the elimination rules and an introduction rule for the corresponding elimination form. For instance, the union elimination rule has a goal of the form H `>>` T; the extracted code is a term of the form `decide(`z`;`$u.a$`;`$v.b$`)`. Correspondingly there is an introduction rule with a goal of the form H `>> decide(`e`;`$u.a$`;`$v.b$`) in` $T[e/z]$. This rule states the general conditions under which we may introduce a `decide`. Notice that any `decide` introduced by an elimination rule will have a variable in the first argument position. However, the additional generality of the introduction is obtained at a price; it is not possible to produce the subgoals of the rule instance from the goal alone. Two parameters are needed: one to specify the union type, written `using` $A \mid B$, and one to specify the range type, written `using` $z.T$. This additional complexity is part of the price of the increased expressiveness of the Nuprl logic.

Equality and Simplification Rules

Another major difference between the Lambda–prl rules and the Nuprl rules is the treatment of equality and simplification. The logic of Lambda–prl is sufficiently simple that all equality reasoning can be lumped into a single rule. Similarly, simplification of terms is handled by a single rule. The two are combined, together with some elementary arithmetic reasoning, into a decision procedure called `arith` which handles nearly all of the low–level details of Lambda–prl proofs. However, the additional expressiveness of the Nuprl logic prevents such a simple solution. For each term constructor there is a rule of equality, for each elimination form there is a computation rule, and there is a general rule of substitution.

For instance, the equality rule for `inl` is as follows.

H `>> inl(`a`) = inl(`b`) in` $A \mid B$ `by intro`
  H `>>` a `=` b `in` A
  H `>>` B `in U`i

For `decide` there are two equality rules: the basic equality rule for the constructor and a computation rule. The goals are, respectively,

$$H \texttt{ >> decide(}e\texttt{;}u.s\texttt{;}v.t\texttt{) = decide(}e'\texttt{;}u.s'\texttt{;}v.t'\texttt{) in } T[e/z]$$

and

$$H \texttt{ >> decide(inl(}a\texttt{);}u.s\texttt{;}v.t\texttt{) = } s[a/u] \texttt{ in } T[\texttt{inl(}a\texttt{)}/z]$$

Equations obtained with these rules can be used by the equality and substitution rules.

Differences in the User Interfaces

The Lambda–prl and Nuprl user interfaces are nearly identical, although there are two noticeable differences.

- In Nuprl the `eval` command invokes an interactive evaluation facility, into which one enters terms and bindings to be evaluated. EVAL objects residing in the library may contain def refs.[1] In Lambda–prl the `eval` command has the form `eval` *term*, and *term* may not contain definition references.
- In Lambda–prl, identifiers can include the pound sign ("`#`") in addition to the alphanumeric characters and the underscore ("`_`"). In Nuprl, the at sign ("`@`") is used instead of the pound sign. Note that users are encouraged not to use at signs in their own identifiers, since the system implementers have reserved them for their own purposes.

Simulating Lambda–prl Constructs in Nuprl

Logical Operators

The logical operators are not part of Nuprl; they can be simulated by using DEF's described in chapter 3. We will use these simulated logical operators below in the examples.

Lists

The `hd` and `tl` built-in functions of Lambda–prl do not exist in Nuprl. They can be simulated with the following definitions.

```
hd(<a:list>)==list_ind(<a>; "?"; h,t,v.h)
tl(<a:list>)==list_ind(<a>; nil; u,v,w.v)
```

[1] See chapter 7 for more on the `eval` command.

Given these definitions, it is straightforward to prove in Nuprl that `hd` satisfies

```
>> All A:U1. All L:A list. hd(L) in A => ~(L=nil in A list)
```

and

```
>> All A:U1. All x:A. All L:A list. hd(x.L)=x in A
```

Once the two goals above are proved they can be used in other proofs as lemmas. Similar facts for `tl` are also easy to prove.

Primitive Recursive Functions

Since primitive recursive function objects are not available in Nuprl, they have to be simulated. Consider the primitive recursive function

$$\begin{array}{lcl} F(0,y) & = & G(y) \\ F(n+1,y) & = & H(n,F(n,y),y) \end{array}$$

Shown below is a DEF that serves as a template for defining such functions.[2]

```
(F such that F(0,y)=(<G:base>)(y),
             F(n+1,y)=(<H:body>)(n,F(n,y),y)
)
==
\n.(\y.ind(n;
           u,v."?";
           (\_y_.(<G>)(_y_));
           u,v.(\_y_.(<H>)(u)(v(_y_))(_y_))
          )
          (y)
   )
```

The DEF above could be used to define an exponentiation DEF in the following way.

```
<y:int>**<n:int>
==
((F such that F(0,y)=(\z.z)(y),
              F(n+1,y)=(\a.\b.\c.(b*c))(n,F(n,y),y)
 )(<n>)
)(<y>)
```

[2] The variable `_y_` contains underscores because we want a name unlikely to appear in the surrounding text.

Extraction Terms

To use the extracted term `E` from a (complete) theorem `T` in another theorem, make `E` a DEF:

```
E==term_of(T).
```

The evaluation mechanism may be used to evaluate `term_ of(T)` and to bind its value to a variable.

Appendix C: Direct Computation Rules

This appendix gives a more precise description of the direct computation rules than the one given in chapter 8. It should probably be ignored by those who find the previous description adequate, for the details are rather complicated.

The direct computation rules allow one to modify a goal by directing the system to perform certain reduction steps within the conclusion or a hypothesis. The "direct" in "direct computation" refers to the fact that no well–formedness subgoals are entailed. The present form of these rules, involving "tagged terms" as described in chapter 8, was chosen to provide the user with a high degree of control over what reductions are performed. The tagged terms may be somewhat inconvenient at the rule–box level, but it is expected that the vast majority of uses of these rules will be by tactics.

First, we define what it means to *compute* a term for n steps (for n a nonnegative integer). To do this we define a function $compute_n$ on terms t. Roughly speaking, $compute_n(t)$ is the result of doing computation (as described in chapter 5) on t until it can be done no further or until a total of n reductions have been done, whichever comes first. The precise definition is as follows. If t is not a `term_of` term and not a noncanonical term, or if n is 0, then $compute_n(t)$ is t. If t is `term_of(`*name*`)` then $compute_n(t)$ is $compute_{n-1}(t')$, where t' is the term extracted from the theorem named *name*. If t is a noncanonical term with one principal argument e then replace e in t by $compute_n(e)$ to obtain t', and let k be the number of replacements of redices by contracta done by $compute_n(e)$. If $n > k$, and t' is a redex, then $compute_n(t)$ is $compute_{n-k-1}(t'')$, where t'' is the contractum of t'. If t is noncanonical with two principal arguments (e.g., `int_eq`) then replace the leftmost one, e, with $compute_n(e)$ to obtain t', and let k be the number of reductions done by $compute_n(e)$. If $compute_n(e)$ is not a canonical term of the right type then $compute_n(t)$ is t'; otherwise, it is $compute_{n-k}(t')$, where t' is treated as having one (the second) principal argument.

Having now defined $compute_n(t)$ we can define what it means to do the

computations indicated by a tagging. If T is a term with tags then the computed form of T is obtained by successively replacing subterms of T of the form `[[`n`;`t`]]` or `[[*;`t`]]`, where t has no tags in it, by $compute_n(t)$ or $compute_\infty(t)$, respectively, where $compute_\infty$ is defined in the obvious way.

It seems very plausible that one should be able to put tags anywhere in a term. Unfortunately, it has not yet been proved that this would be sound, so there is at present a rather complicated restriction, based on technical considerations, on the set of subterm occurrences that can legally be tagged. Let T be a term and define a relation $\prec$ on subterm occurrences of T by $u \prec v$ *if and only if* u occurs properly within v. A $t \preceq T$ may be tagged if there are u and v with $t \preceq u \preceq v \preceq T$ such that:

1. $v \prec r \preceq T \Rightarrow$ r is a canonical term, and
2. u is free in v (i.e., no free variables of u are bound in v), and
3. $t \prec r \preceq u \Rightarrow$
   (a) r is a noncanonical term with t occurring within the principal argument of r,
   (b) r is a `spread`, `decide`, `less`, `int_eq`, `any` or `atom_eq` term, or
   (c) r is `ind(`a`;`m`,`x`.`b`;`c`;`n`,`y`.`d`)` such that x,y do not occur free in b,d respectively, or
   (d) r is `list_ind(`a`;`h`,`t`,`v`.`b`)` and v does not occur free in b.

For the purposes of the definition above, the notions of canonical and noncanonical are extended in the obvious way to terms with free variables.

Bibliography

[Aczel 77]
Peter Aczel.
An introduction to inductive definitions.
In *Handbook of Mathematical Logic*, J. Barwise, ed.
North–Holland, Amsterdam, 1977, pages 739–782.

[Aczel 78]
Peter Aczel.
The type theoretic interpretation of constructive set theory.
In *Logic Colloquium '77*,
A. MacIntyre, L. Pacholaki, and J. Paris, eds.
North–Holland, Amsterdam, 1978, pages 55–66.

[Aho & Ullman 74]
Alfred V. Aho and J. D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison–Wesley, Reading, MA, 1974.

[Aiello, Aiello, & Weyhrauch 77]
L. Aiello, M. Aiello, and R. W. Weyhrauch.
Pascal in LCF: semantics and examples of proof.
Theoretical Computer Science, v. 5, n. 2 (1977)
pages 135–178.

[Allen 86]
Stuart F. Allen.
The Semantics of Type Theoretic Languages.
Doctoral Dissertation, Computer Science Department,
Cornell University, August 1986 (expected).

[Anderson & Johnstone 62]
John M. Anderson and Henry W. Johnstone.
Natural Deduction.
Wadsworth, Belmont, CA, 1962.

[Andrews 65]
P. B. Andrews.
Transfinite Type Theory with Transfinite Type Variables.
North–Holland, Amsterdam, 1965.

[Andrews 71]
P. B. Andrews.
Resolution in type theory.
J. Symbolic Logic, v. 36 (1971), pages 414–432.

[Artin, Grothendieck, & Verdier 72]
M. Artin, A. Grothendieck, and J. L. Verdier.
Théories des topos et cohomologie étale des schémas.
In *Lecture Notes in Mathematics*, vols. 269 and 270.
Springer–Verlag, New York, 1972 (first appeared in 1963).

[Backhouse 84]
Roland Backhouse.
A Note on Subtypes in Martin-Löf's Theory of Types.
Computer Science Department, University of Essex, England,
CSM–70, November 1984.

[Backhouse 85]
Roland Backhouse.
Algorithm Development in Martin-Löf's Type Theory.
Computer Science Department, University of Essex, England, 1985.

[deBakker 80]
Jaco deBakker.
Mathematical Theory of Program Correctness.
Prentice–Hall, Englewood Cliffs, NJ, 1980.

[Barringer, Cheng, & Jones 84]
H. Barringer, J. H. Cheng, and C. B. Jones.
A Logic Covering Undefinedness in Program Proofs.
Department of Computer Science, University of Manchester, England,
1984.

[Barwise 75]
Jon Barwise.
Admissible Sets and Structures.
Springer–Verlag, New York, 1975.

[Barwise & Perry 83]
Jon Barwise and John Perry.
Situations and Attitudes.
MIT Press, Cambridge, MA, 1983.

[Bates 79]
Joseph L. Bates.
A Logic for Correct Program Development.
Doctoral Dissertation, Computer Science Department
Cornell University, 1979.

[Bates & Constable 85]
Joseph L. Bates and Robert L. Constable.
Proofs as programs.
ACM Trans. Prog. Lang. Sys., v. 7, n. 1 (1985),
pages 113–136.

[Bates & Constable 83]
J. L. Bates and R. L. Constable.
The Nearly Ultimate PRL.
Computer Science Department Technical Report,
TR–83–551.
Cornell University, Ithaca, NY, 1983.

[Beeson 81]
M. J. Beeson.
Formalizing constructive mathematics: why and how?
In *Constructive Mathematics*, F. Richman, ed.,
Lecture Notes in Mathematics.
Springer–Verlag, New York, 1981, pages 146–190.

[Beeson 83]
M. J. Beeson.
Proving programs and programming proofs.
In *International Congress on Logic, Methodology and Philosophy of Science*, Salzburg, Austria. North–Holland, Amsterdam, 1983.

[Beeson 85]
M. J. Beeson.
Foundations of Constructive Mathematics.
Springer–Verlag, New York, 1985.

[Beth 56]
E. W. Beth.
Semantic construction of intuitionistic logic.
Meded. Akad.
Amsterdam, N.R. 19, n. 11, (1956), pages 357–388.

[Bibel 80]
Wolfgang Bibel.
Syntax–directed, semantics–supported program synthesis.

Artificial Intelligence v. 14 (1980),
pages 243–261.

[Bibel 82]
Wolfgang Bibel.
Automated Theorem Proving.
Sohn & Vieweg
Wiesbaden, West Germany, 1982.

[Bishop 67]
Errett Bishop.
Foundations of Constructive Analysis.
McGraw–Hill, New York, 1967.

[Bishop 70]
Errett Bishop.
Mathematics as a numerical language.
In *Intuitionism and Proof Theory*,
J. Myhill, et al., eds.
North–Holland, Amsterdam, 1970, pages 53–71.

[Bishop & Bridges 85]
Errett Bishop and Douglas Bridges.
Constructive Analysis.
Springer–Verlag, New York, 1985.

[Bishop & Cheng 72]
Errett Bishop and H. Cheng.
Constructive Measure Theory.
Mem. Am. Math. Soc. 116, 1972.

[Bledsoe 77]
W. Bledsoe.
Non–resolution theorem proving.
Artificial Intelligence v. 9 (1977), pages 1–36.

[Bledsoe, Boyer, & Henneman 71]
W. Bledsoe, R. Boyer, and W. Henneman.
Computer proofs of limit theorems.
Artificial Intelligence v. 2 (1971), pages 55–77.

[Bourbaki 68]
N. Bourbaki.
Elements of Mathematics, Vol. I: *Theory of Sets.*
Addison–Wesley, Reading, MA, 1968.

[Boyer & Moore 79]
R. Boyer and J. S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer & Moore 81]
R. S. Boyer and J. S. Moore.
Metafunctions: proving them correct and using them efficiently as new proof procedures.
In *The Correctness Problem in Computer Science*,
R. S. Boyer and J. S. Moore, eds.
Academic Press, New York, 1981, pages 103–184.

[Bridges 79]
D. S. Bridges.
Constructive Functional Analysis.
Pitman, London, 1979.

[Brouwer 23]
L. E. J. Brouwer.
On the significance of the principle of excluded middle in mathematics, especially in function theory.
J. fur die Reine und Angewandte Math, v. 154,
(1923), 1–7.
In [vanHeijenoort 67], pages 334–345.

[Brouwer 75]
L. E. J. Brouwer.
Vol. 1, *Collected Works*
A. Heyting, ed.
North–Holland, Amsterdam, 1975.

[deBruijn 70]
N. G. deBruijn.
The mathematical language AUTOMATH, its usage and some of its extensions.
In *Symposium on Automatic Demonstration*,
Lecture Notes in Mathematics, Vol. 125.
Springer–Verlag, New York, 1970, pages 29–61.

[deBruijn 80]
N. G. deBruijn.
A survey of the project AUTOMATH.
In *Essays in Combinatory Logic, Lambda Calculus, and Formalism*,
J. P. Seldin and J. R. Hindley, eds.
Academic Press, New York, 1980, pages 589–606.

[Buchholz et al. 81]
W. Buchholz, S. Feferman, W. Pohlers, and W. Sieg.
Iterated Inductive Definitions and Subsystems of Analysis.
In *Recent Proof–Theoretical Studies*,
Lecture Notes in Mathematics, Vol. 897,
Springer–Verlag, New York, 1981.

[Bundy 83]
Alan Bundy.
The Computer Modelling of Mathematical Reasoning.
Academic Press, New York, 1983.

[Burstall 69]
R. Burstall.
Proving properites of programs by structural induction.
Comp. J. v. 12, n. 1, (1969), pages 41–48.

[Burstall & Lampson 84]
R. M. Burstall and B. Lampson.
A kernel language for abstract data types and modules.
In *Semantics of Data Types*,
Lecture Notes in Computer Science, Vol. 173.
Springer–Verlag, New York, 1984 pages 1–50.

[Burstall, MacQueen, & Sanella 80]
R. M. Burstall, D. B. MacQueen, and D. T. Sanella.
HOPE: an experimental applicative language.
In Proceedings 1980 Lisp Conference,
Computer Science Department,
Stanford University, Stanford, CA, 1980, pages 136–143.

[Cardelli 83]
L. Cardelli.
The functional abstract machine.
Polymorphism: The ML/LCF/Hope Newsletter I (1983).

[Cardelli 84]
L. Cardelli.
A semantics of multiple inheritance.
In *Semantics of Data Types*,
Lecture Notes in Computer Science, vol 173
Springer–Verlag, New York, 1984, pages 51–67.

[Cartwright 82]
R. Cartwright.
Toward a logical theory of program data.

In *Logics of Programs*,
Lecture Notes in Computer Science Vol. 131.
Springer–Verlag, New York, 1982, pages 37–51.

[Chan 74]
Y.–K. Chan.
Notes on constructive probability theory.
Ann. Probability, v. 2 (1974), pages 51–75.

[Charniak & McDermott 85]
Eugene Charniak and Drew McDermott.
Introduction to Artificial Intelligence.
Addison–Wesley, Reading, MA, 1985.

[Chisholm 85]
Paul Chisholm.
Derivation of a Parsing Algorithm in Martin-Löf's Theory of Types.
Department of Computer Science, Heriot–Watt University, Edinburgh, Scotland, 1985.

[Church 40]
A. Church.
A formulation of the simple theory of types.
J. Symbol. Logic v. 5, (1940), pages 56–68.

[Church 51]
Alonzo Church.
The calculi of lambda–conversion.
Annals of Mathematics Studies, No. 6.
Princeton University Press, Princeton, NJ, 1951.

[Clarke & Emerson & Sistla 83]
Edmond M. Clarke, Jr., E. A. Emerson and A. P. Sistla.
Automatic verification of finite state concurrent systems using temporal logic specification: a practical approach.
In *10th ACM Symposium on Principles of Programming Languages*.
ACM, New York, 1983.

[Cleaveland 86]
W. Rance Cleaveland.
Type Theoretic Models of Concurency.
Doctoral Dissertation, Computer Science Department,
Cornell University 1986.

[Clocksin & Mellish 81]
W. F. Clocksin and C. S. Mellish.

Programming in Prolog.
Springer–Verlag, New York, 1981.

[Constable 71]
Robert L. Constable.
Constructive mathematics and automatic programs writers.
In *Proceedings of IFIP Congress*, Ljubljana,
1971, pages 229–233.

[Constable 77]
Robert L. Constable.
On the theory of programming logics.
In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*,
Boulder, Colorado, May 1977, pages 269–285.

[Constable 83]
Robert L. Constable.
Programs as proofs.
Inform. Processing Lett., v. 16, n. 3, (1983),
pages 105–112.

[Constable 85]
Robert L. Constable.
Constructive mathematics as a programming logic I: some principles of theory.
Ann. Discrete Math., v. 24 (1985),
pages 21–38.

[Constable & Mendler 85]
Robert L. Constable and N. P. Mendler.
Recursive definitions in type theory.
In *Proceedings of Logics of Programs Conference*,
Lecture Notes in Computer Science,
Springer–Verlag, New York, 1985, pages 61–78.

[Constable, Johnson, & Eichenlaub 82]
Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub.
Introduction to the PL/CV2 Programming Logic,
Lecture Notes in Computer Science, Vol. 135,
Springer–Verlag, New York, 1982.

[Constable, Knoblock, & Bates 84]
Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates.
Writing programs that construct proofs.

J. Automated Reasoning, v. 1 n. 3,
(1984), pages 285–326.

[Constable & O'Donnell 78]
Robert L. Constable and Michael J. O'Donnell.
A Programming Logic.
Winthrop, Cambridge, MA, 1978.

[Constable & Zlatin 84]
Robert L. Constable and Daniel R. Zlatin.
The type theory of PL/CV3.
ACM Trans. Prog. Lang. Sys., v. 7, n. 1 (1984),
pages 72–93.

[Coquand & Huet 85]
Thierry Coquand and Gerard Huet.
Constructions: A Higher Order Proof System for Mechanizing Mathematics.
EUROCAL 85, Linz, Austria, April 1985.

[Curry, Feys, & Craig 58]
H. B. Curry, R. Feys, and W. Craig.
Combinatory Logic, Vol. 1.
North–Holland, Amsterdam, 1958.

[Curry, Hindley, & Seldin 72]
H. B. Curry, J. R. Hindley, and J. P. Seldin.
Combinatory Logic, Vol. 2.
North–Holland, Amsterdam, 1972.

[van Daalen 80]
Diedrik T. van Daalen.
The Language Theory of AUTOMATH.
Doctoral Dissertation, Technical University of Eindhoven,
1980.

[Davis 65]
M. Davis, ed.
The Undecidable.
Raven Press, New York, 1965.

[Davis & Lenat 82]
Randall Davis and Douglas B. Lenat.
Knowledge–Based Systems in Artificial Intelligence.
McGraw–Hill, New York, 1982.

[Davis & Schwartz 79]
M. Davis and J. T. Schwartz.
Metamathematical extensibility for theorem verifiers and proof checkers.
Comp. Math. with Applications, v. 5 (1979),
pages 217–230.

[Dijkstra 76]
Edsger W. Dijkstra.
A Discipline of Programming.
Prentice–Hall, Englewood Cliffs, NJ, 1976.

[Donahue & Demers 79]
J. E. Donahue and A. J. Demers.
Revised Report on Russell.
Computer Science Department
Technical Report, TR 79–389.
Cornell University, Ithaca, NY, September 1979.

[Donahue & Demers 85]
J. E. Donahue and A. J. Demers.
Data types are values.
TOPLAS, v. 7, n. 3, (July 1985), pages 426–445.

[Donzeau–Gouge et al. 80]
Veronique Donzeau–Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang.
Programming environments based on structured editors: the Mentor experience.
INRIA, Rapports de Recherches, No. 26, July 1980.

[Doyle 79]
John Doyle.
A truth maintenance system.
Artificial Intelligence, v. 12, n. 3.
(1979), pages 231–272.

[Dummet 77]
Michael Dummett.
Elements of Intuitionism, Oxford Logic Series.
Clarendon Press, Oxford, 1977.

[Feferman 79]
S. Feferman.
Constructive theories of functions and classes.
In *Logic Colloquium '78*,
North–Holland, Amsterdam, 1979, pages 159–224.

[Fitch 52]
Frederic B. Fitch.
Symbolic Logic.
Roland Press, New York 1952

[Fitting 83]
M. Fitting.
Proof Methods for Modal and Intuitionistic Logics.
D. Reidel, Dordrecht, The Netherlands, 1983.

[Fortune, Leivant, & O'Donnell 83]
S. Fortune, D. Leivant, and M. O'Donnell.
The expressiveness of simple and second order type structures.
J. ACM, v. 30 (1983),
pages 151–185.

[Frege 1879]
Gottlob Frege.
Begriffsschrift, a formula language, modeled upon that for arithmetic, for pure thought.
In [vanHeijenoort 67], pages 1–82.

[Friedman 73]
Harvey Friedman.
The consistency of classical set theory relative to set theory with intuitionistic logic.
J. Symbolic Logic, v. 56, n. 38 (1973),
pages 315–319.

[Gabby 81]
Dov Gabby.
Semantical Investigations in Heyting's Intuitionistic Logic.
D. Reidel, Dordrecht, The Netherlands, 1981.

[Gentzen 69]
Gerhard Gentzen.
Investigations into logical deduction.
In *The Collected Papers of Gerhard Gentzen*, M. E. Szabo, ed.
North–Holland, Amsterdam, 1969.

[Girard 71]
J. Y. Girard.
Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types.
In *2nd Scandinavian Logic Symposium*,
J. E. Fenstad, ed.
North–Holland, Amsterdam, 1971, pages 63–92.

[Goad 80]
C. Goad.
Proofs as descriptions of computation.
In *Proceedings of the 5th Conference on Automated Deduction*
Les Arcs, France, *Lecture Notes in Computer Science*, Vol. 87,
Springer–Verlag, New York, 1980, pages 39–52.

[Goguen, Thatcher & Wagner 78]
J. A. Goguen, J. W. Thatcher and E. G. Wagner.
Initial algebra approach to the specification, correctness and implementation of abstract data types.
In *Current Trends in Programming Methodology*, Vol. IV, R.
Yeh, ed.
Prentice–Hall, Englewood Cliffs, NJ, 1978.

[Goldblatt 79]
Robert Goldblatt.
TOPI: The Categorial Analysis of Logic.
North–Holland, Amsterdam, 1979.

[Goodman 84]
Nicolas D. Goodman.
Epistemic arithmetic is a Conservative Extension of Intuitionistic Arithmetic.
J. Symbolic Logic, v. 49, n. 1 (1984),
pages 192–204.

[Goodman & Myhill 72]
Nicolas D. Goodman and John Myhill.
The formalization of Bishop's constructive mathematics.
Toposes, Algebraic Geometry and Logic,
Lecture Notes in Mathematics, Vol. 274.
Springer–Verlag, New York, 1972, pages 83–96.

[Gordon, Milner, & Wadsworth 79]
Michael Gordon, Arthur Milner, and Christopher Wadsworth.
Edinburgh LCF: A Mechanized Logic of Computation,
Lecture Notes in Computer Science, Vol. 78
Springer–Verlag, New York, 1979.

[Goto 79]
S. Goto.
Program synthesis from natural deduction proofs.
International Joint Conference on Artificial Intelligence,
Tokyo, 1979.

[Gries 78]
David Gries, ed.
Programming Methodology: A Collection of Articles by Members of IFIP WG2.3.
Springer–Verlag, New York, 1978.

[Gries 81]
David Gries.
The Science of Programming.
Springer–Verlag, New York, 1981.

[Guttag & Horning 78]
J. Guttag and J. Horning.
The algebraic specification of abstract data types.
In *Programming Methodology*, D. Gries, ed.
Springer–Verlag, New York. 1978.

[Harper 85]
Robert W. Harper.
Aspects of the Implementation of Type Theory.
Doctoral Dissertation,Computer Science Department,
Cornell University, June 1985.

[Hehner 79]
E. C. R. Hehner.
do considered **od**: a contribution to the programming calculus.
Acta Informatica, v. 11, n. 4 (1979),
pages 287–304.

[vanHeijenoort 67]
J. van Heijenoort.
From Frege to Gödel: A Sourcebook in Mathematical Logic.
Harvard University Press, Cambridge, MA, 1967.

[Heyting 30]
Arend Heyting.
Die formalen Regeln der intuitionistischen Logik.
Sitzungsber. Preuss. Akad. Wiss., Phys. – Math Kl.,
1930, pages 42–56.

[Heyting 56]
Arend Heyting.
Intuitionism: An Introduction.
North–Holland, Amsterdam, 1956.

[Hoare 72]
C. A. R. Hoare.
Notes on data structuring.
In *Structured Programming*.
Academic Press, New York, 1972, pages 83–174.

[Howard 80]
W. Howard.
The formulas–as–types notion of construction.
In *To H. B. Curry: Essays on Combinatory Logic, Lambda–Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, eds.
Academic Press, New York, 1980, pages 479–490.

[Howe 86]
Douglas J. Howe.
Implementing Analysis.
Doctoral Dissertation,
Computer Science Department,
Cornell University, 1986 (expected).

[Huet 75]
Gerard P. Huet.
A unification algorithm for typed lambda–calculus.
Theoretical Computer Science, v. 1, n. 1 (1975),
pages 27–58.

[Johnson 83]
Scott D. Johnson.
A Computer System for Checking Proofs.
UMI Press, New York, 1983.

[Jutting 79]
L. S. Jutting.
Checking Landau's "Grundlagen" in the AUTOMATH System.
Doctoral Thesis,
Eindhoven University, Eindhoven, The Netherlands, 1977.
Also in *Math. Centre Tracts No. 83*, Math. Centre, Amsterdam, 1979.

[Kay 77]
Alan Kay.
Microelectronics and the Personal Computer.
Scientific American, September, 1977.

[Kay & Goldberg 77]
Alan Kay and Adele Goldberg.
Personal dynamic media.
Computer, v. 31, (March 1977).

[Kleene 45]
Stephen C. Kleene.
On the interpretation of intuitionistic number theory.
J. Symbolic Logic v. 10 (1945),
pages 109–124.

[Kleene 52]
Stephen C. Kleene.
Introduction to Metamathematics.
Van Nostrand, Princeton, NJ, 1952.

[Kleene & Vesley 65]
Stephen C. Kleene and R. E. Vesley.
The Foundations of Intuitionistic Mathematics.
North–Holland, Amsterdam, 1965.

[Knoblock 86]
Todd B. Knoblock.
Formal Metamathematics and Reflection in Constructive Type Theory.
Doctoral Dissertation, Computer Science Department, Cornell University, 1986.

[Knuth 68]
D. E. Knuth.
The Art of Computer Programming, Vol. I.
Addison–Wesley, Reading, MA, 1968.

[Kowalski 79]
Robert Kowalski.
Logic for Problem Solving.
North–Holland, New York, 1979.

[Kozen 77]
Dexter C. Kozen.
Complexity of Finitely Presented Algebras.
Doctoral Dissertation, Computer Science Department,
Cornell University, 1977.

[Krafft 81]
Dean B. Krafft.
AVID: A System for the Interactive Development of Verifiably Correct Programs.
Doctoral Dissertation, Computer Science Department
Cornell University, 1981.

[Kreisel 62]
G. Kreisel.

Foundations of intuitionistic logic.
In *Logic, Methodology and Philosophy of Science*
Stanford University Press, Stanford, CA,
1962, pages 198–210.

[Kreisel 70]
G. Kreisel.
Hilbert's program and the search for automatic proof procedures.
In *Symposium on Automatic Demonstration, Lecture Notes in Mathematics*, Vol. 125.
Springer–Verlag, New York (1970), pages 128–146.

[Lakatos 76]
Imre Lakatos.
Proofs and Refutations.
Cambridge University Press, Cambridge, 1976.

[Lambek 80]
J. Lambek.
From types to sets.
Advances in Mathematics v. 35 (1980).

[Lauchli 70]
H. Lauchli.
An abstract notion of realizability for which intuitionistic predicate calculus is complete.
In *Intuitionism and Proof Theory*, J. Myhill, A. Kino, and R. E. Vesley, eds.
North–Holland, Amsterdam, 1970, pages 227–234.

[Leivant 83]
Daniel Leivant.
Structural semantics for polymorphic data types (preliminary report).
In *Proceedings of the 10th ACM Symposium in the Principles of Programming Languages*, ACM, New York, 1983, pages 155–166.

[Linger, Mills & Witt 79]
R. C. Linger, H. D. Mills and B. I. Witt.
Structured Programming Theory and Practice.
Addison-Wesley, Reading, MA, 1979.

[Loveland 78]
Donald W. Loveland.
Automated Theorem Proving: A Logical Basis.
North–Holland, Amsterdam, 1978.

[MacLane 69]
Saunders MacLane.
Foundations for categories and sets.
In *Category Theory, Homology Theory and Their Applications II*, *Lecture Notes in Mathematics*, Vol. 92.
Springer–Verlag, New York, 1969, pages 146–164.

[MacQueen 85]
David B. MacQueen.
Modules for Standard ML.
Polymorphism Newsletter, v. 2, n. 2 (1985).

[MacQueen, Plotkin, & Sethi 84]
D. B. MacQueen, G. D. Plotkin, and R. Sethi.
An ideal model for recursive polymorphic types.
In *11th ACM Symposium on Principles of Programming Languages*, 1984, pages 165–174.

[Manna & Waldinger 80]
Z. Manna and R. Waldinger.
A deductive approach to program synthesis.
ACM Trans. Prog. Lang. Sys. v. 2 n. 1 (1980), pages 90–121.

[Manna & Waldinger 85]
Z. Manna and R. Waldinger.
The Logical Basis for Computer Programming.
Addison–Wesley, Reading, MA, 1985.

[Martin-Löf 70]
Per Martin-Löf.
Notes on Constructive Mathematics.
Almqvist & Wiksell, Stockholm, 1970.

[Martin-Löf 71]
Per Martin-Löf.
Hauptsatz for the intuitionistic theory of iterated inductive definitions,
In *Proceedings of the Second Scandinavian Logic Symposium*, J. E. Fenstad ed.
North–Holland, Amsterdam, 1971, pages 179–216.

[Martin-Löf 73]
Per Martin-Löf.
An intuitionistic theory of types: predicative part.

In *Logic Colloquium '73*,
H. E. Rose and J. C. Shepherdson, eds.
North–Holland, Amsterdam, 1973, pages 73–118.

[Martin-Löf 82]
Per Martin-Löf.
Constructive mathematics and computer programming.
In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*.
North–Holland, Amsterdam, 1982, pages 153–175.

[Martin-Löf 84]
Per Martin-Löf.
Intuitionistic Type Theory.
Studies in Proof Theory Lecture Notes, *BIBLIOPOLIS*, Napoli, 1984.

[The Mathlab Group 77]
The Mathlab Group.
MACSYMA Reference Manual.
MIT, December 1977.

[McAllester 82]
David A. McAllester.
Reasoning Utility Package User's Manual, Version One.
AI Lab. MIT, AIM–667, April 1982.

[McCarthy 62]
J. McCarthy.
Computer programs for checking mathematical proofs.
In *Proceedings of the Symposia in Pure Mathematics*, Vol. V, *Recursive Function Theory*.
American Mathematical Society, Providence, RI, 1962.

[McCarthy 63]
J. McCarthy.
A basis for a mathematical theory of computation.
In *Computer Programming and Formal Systems*.
P. Braffort and D. Herschberg, eds.
North–Holland, Amsterdam, 1963, pages 33–70.

[McCarty 84]
David C. McCarty.
Realizability and recursive mathematics.
Doctoral Dissertation, Computer Science Department,
Carnegie–Mellon University, 1984.

[McGettrick 78]
A. D. McGettrick.
Algol 68, A First and Second Course.
Cambridge University Press, Cambridge, 1978.

[Mendler 86]
N. P. Mendler.
Recursive Types and Infinite Objects.
Doctoral Dissertation, Computer Science Department
Cornell University, 1986 (expected).

[Metakides & Nerode 79]
G. Metakides and A. Nerode.
Effective content of field theory.
Ann. Math. Logic, v. 17 (1979),
pages 289–320.

[Meyer 82]
Albert R. Meyer.
What is a model of the lambda calculus?
Information and Control, v. 52, n. 1 (1982),
pages 87–122.

[Milner 85]
R, Milner.
The standard ML core language.
Polymorphism Newletter, v. 2, n. 2 (1985),
pages 1–28.

[Milner & Weyhrauch 72]
R. Milner and R. Weyhrauch.
Proving computer correctness in mechanized logic.
In *Machine Intelligence* Vol. 7, B. Meltzer and D. Michie, eds.),
Edinburgh University Press,
Edinburgh, Scotland, 1972, pages 51–70.

[Minsky 81]
Marvin Minsky.
A framework for representing knowledge.
In *Mind Design*, J. Haugeland, ed.
MIT Press, Cambridge, MA, 1981, pages 95–128.

[Mitchell & Plotkin 85]
John Mitchell and Gordon Plotkin.
Abstract types have existential type.
In *Proceedings of 12th ACM Symposium on*

Principles of Programming Languages.
ACM, New York, 1985, pages 37–51.

[Morse 65]
A. Morse.
A Theory of Sets.
Academic Press, New York, 1965.

[Moschovakis 74]
Yiannis N. Moschovakis.
Elementary Induction on Abstract Structures.
North–Holland, Amsterdam, 1974.

[Mulmuley 84]
K. Mulmuley.
Mechanization of existence proofs of recursive functions.
In *Proceedings 7th International Conference on Automated Deduction.*
Lecture Notes in Computer Science, Vol. 170.
Springer–Verlag, New York, 1984, pages 460–475.

[Myhill 75]
J. Myhill.
Constructive Set Theory.
J. Symbol. Logic, v. 40 (1975),
pages 347–383.

[Nederpelt 80]
R. P. Nederpelt.
A system of natural reasoning based on a typed λ–calculus.
Bull. Eur. Assoc. Theoret. Comp. Sci., v. 11 (1980),
pages 130–131.

[Nelson & Oppen 79]
Greg Nelson and Derek C. Oppen.
Simplification by cooperating decision procedures.
ACM Trans. Prog. Lang. Sys. v. 1, n. 2 (1979),
pages 245–257.

[Nepeivoda 82]
N. N. Nepeivoda.
Logical approach to programming.
In *Sixth International Congress for Logic, Methodology and Philosophy of Science*
North–Holland, Amsterdam, 1982, pages 109–122.

[Nordstrom 81]
Bengt Nordstrom.

Programming in constructive set theory: some examples.
In *Proceedings 1981 Conference on Functional Programming Languages and Computer Architecture*.
Portsmouth, England, 1981, pages 141–153.

[Nordstrom & Petersson 83]
Bengt Nordstrom and Kent Petersson.
Types and specifications.
In *Proceedings IFIP '83*, R. E. A. Mason, ed.
North–Holland, Amsterdam, 1983, pages 915–920.

[Nordstrom & Smith 84]
Bengt Nordstrom and Jan Smith.
Propositions, types, and specifications in Martin-Löf's type theory.
BIT, v. 24, n. 3 (1984), pages 288–301.

[Nuprl Staff 83]
The Nuprl Staff.
PRL: Proof Refinement Logic Programmer's Manual (Lambda PRL, VAX Version).
Computer Science Department, Cornell University, New York, 1983.

[O'Donnell 85]
Michael J. O'Donnell.
Equational Logic as a Programming Language.
MIT Press, Cambridge, MA, 1985.

[Paterson & Wegman 78]
M. S. Paterson and M. N. Wegman.
Linear unification, *J. CSS*, v. 16 (1978),
pages 158–197.

[Paulson 83a]
Lawrence Paulson.
Tactics and Tacticals in Cambridge LCF.
Technical Report 39,
Computer Laboratory,
University of Cambridge, July 1983.

[Paulson 83b]
Lawrence Paulson.
A higher–order implementation of rewriting.
Sci. Comp. Programming, n. 3 (1983),
pages 119–149.

[Paulson 84]
Lawrence Paulson.

Verifying the Unification Algorithm in LCF.
Technical Report 50,
Computer Laboratory,
University of Cambridge, March 1984.

[Paulson 85]
Lawrence Paulson.
Lessons learned from LCF: a survey of natural deduction proofs.
Comp. J. v. 28 n. 5 (1985).

[Petersson 82]
Kent Pertersson
A Programming System for Type Theory.
Memo 21, Programming Methodology Group,
University of Götberg/Chalmers,
Götberg, Sweden, March 1982.

[Petersson & Smith 85]
Kent Petersson and Jan Smith.
Program Derivation in Type Theory: The Polish Flag Problem.
Computer Science Department, University of Götberg/Chalmers,
Götberg, Sweden, January 1985.

[Platek 66]
Richard Alan Platek.
Foundations of Recursion Theory.
Doctoral Dissertation, Stanford University, 1966.

[Plotkin 77]
Gordon D. Plotkin.
LCF considered as a programming language.
Theoretical Computer Science, v. 5 (1977),
pages 223–257.

[Poincaré 82]
Henri Poincaré.
The Foundations of Science.
University Press of America, Washington, DC, 1982.

[Post 43]
E. Post.
Formal reductions of the general combinatorial decision problem.
American J. Math, v. 65 (1943)
pages 197–215.

[Prawitz 70]
D. Prawitz.

Ideas and results in proof theory.
In *Proceedings of the Second Scandinavian Logic Symposium*,
J. E. Fenstad, ed.
North–Holland, Amsterdam, 1970, pages 235–308.

[Quine 63]
Willard Van Orman Quine.
Set Theory and Its Logic.
Belknap Press, Combridge, MA, 1963.

[Reps 84]
Thomas W. Reps.
Generating Language–Based Environments.
MIT Press, Cambridge, MA, 1984.

[Reps & Alpern 84]
Thomas W. Reps and B. Alpern.
Interactive proof checking.
In *11th ACM Symposium on Principles of Programming Languages*.
ACM, New York, 1984.

[Reps & Teitelbaum 85]
Thomas W. Reps and Tim Teitelbaum.
The Synthesizer Generator Reference Manual,
Computer Science Department,
Technical Report, TR 84–619, Cornell University, Ithaca, NY, August 1985.

[Reynolds 74]
John C. Reynolds.
Towards a theory of type structure.
In *Proceedings Colloque sur la Programmation*,
Lecture Notes in Computer Science Vol. 19.
Springer–Verlag, New York, 1974, pages 408–423.

[Reynolds 83]
John C. Reynolds.
Types, abstraction, and parametric polymorphism.
In *Information Processing 83*.
North–Holland, Amsterdam, 1983, pages 513–523.

[Richman 81]
F. Richman, ed.
Constructive Mathematics.
Lecture Notes in Mathematics, Vol. 873,
Springer–Verlag, New York, 1981.

[Robinson 65]
J. A. Robinson.
A machine–oriented logic based on the resolution principle.
J. ACM v. 12 (1965),
pages 23–41.

[Robinson 71]
J. A. Robinson.
Computational logic: the unification algorithm.
In *Machine Intelligence* Vol. 6,
B. Meltzer and D. Michie, eds.
Edinburgh University Press,
Edinburgh, Scotland, 1971, pages 63–72.

[Russell 08]
Bertrand Russell.
Mathematical logic as based on a theory of types.
Am. J. of Math., v. 30 (1908),
pages 222–262.

[Russell 56]
Bertrand Russell.
In *Logic and Knowledge*, R. D. Marsh, ed.
George Allen & Unwin, London, 1956.

[Sasaki 85]
James T. Sasaki.
The Extraction and Optimization of Programs from Constructive Proofs.
Doctoral Dissertation, Computer Science Department, Cornell University,
1985.

[Scherlis & Scott 83]
W. L. Scherlis and D. Scott.
First steps toward inferential programming.
In *Proceedings IFIP Congress*, Paris, 1983.

[Schutte 77]
Kurt Schutte.
Proof Theory.
Springer–Verlag, New York, 1977.

[Scott 70]
Dana Scott.
Constructive validity.
In *Symposium on Automatic Demonstration,*
Lecture Notes in Mathematics, Vol. 125.
Springer–Verlag, New York, 1970, pages 237–275.

[Scott 76]
Dana Scott.
Data types as lattices.
SIAM J. Computing, v. 5 (1976), pages 522–587.

[Shankar 84]
N. Shankar.
Towards Mechanical Metamathematics.
Institute for Computing Science
Technical Report 43,
University of Texas at Austin, 1984.

[Shultis 85]
Jon Shultis.
INTUIT: A System for Program Synthesis Using
Intuitionistic Logic (draft).
Computer Science Department
University of Colorado, July 1985.

[Siekmann & Wrightson 83]
Jorg Siekmann and Graham Wrightson.
Automation of Reasoning, Classical Papers on Computational Logic:
Vol. 1, 1957–1966;
Vol. 2, 1967–1970.
Springer–Verlag, New York, 1983.

[Smith 84]
Jan Smith.
An interpretation of Martin-Löf's type theory in a type–free theory of propositions.
J. Symbol. Logic, v. 49, n. 3 (1984),
pages 730–753.

[Smullyan 68]
Raymond M. Smullyan.
First–Order Logic.
Springer–Verlag, New York, 1968.

[Sokolowski 83]
S. Sokolowski.
A Note on Tactics in LCF.
Department of Computer Science, Report CSR–140–83,
University of Edinburgh, Edinburgh, Scotland, 1983.

[Stansifer 85]
Ryan D. Stansifer.

Representing Constructive Theories in
High–Level Programming Languages.
Doctoral Dissertation, Computer Science Department,
Cornell University, 1985.

[Statman 79]
R. Statman.
Intuitionistic Propositional Logic is Polynomial–space Complete,
Theoretical Computer Science, v. 9, (1979), pages 73–81.

[Steele 84]
Guy L. Steele.
Common LISP – The Language.
Digital Press, Burlington, MA, 1984.

[Stenlund 72]
Sören Stenlund.
Combinators, λ–terms, and Proof Theory.
D. Reidel, Dordrecht, The Netherlands, 1972.

[Stoy 77]
Joseph E. Stoy.
Denotational Semantics: The Scott–Strachey
Approach to Programming Language Theory.
MIT Press, Cambridge, MA, 1977.

[Suppes 81]
P. Suppes.
University–Level Computer–Assisted Instruction at Stanford: 1968–1981.
Institute for Mathematical Studies in the Social Sciences, Stanford University, Stanford, CA, 1981.

[Tait 67]
William W. Tait.
Intensional interpretation of functionals of finite type.
J. Symbol. Logic, v. 32, n. 2 (1967),
pages 187–199.

[Takasu 78]
A. S. Takasu.
Proofs and programs,
In *Proceedings of the Third IBM Symposium on the Mathematical*
Foundations of Computer Science,
Kansai, 1978.

[Takeuti 75]
Gaisi Takeuti.

Proof Theory.
North–Holland, Amsterdam, 1975.

[Teitelbaum & Reps 81]
R. Teitelbaum and T. Reps.
The Cornell program synthesizer:
a syntax–directed programming environment.
Commun. ACM, v. 24, n. 9 (1981),
pages 563–573.

[Toledo 75]
Sue Ann Toledo.
Tableau systems for first order number theory and certain higher order theories.
In *Lecture Notes in Mathematics* Vol. 447
Springer–Verlag, New York, 1975.

[Troelstra 73]
Ann S. Troelstra.
Metamathematical Investigation of Intuitionistic Arithmetic and Analysis,
Lecture Notes in Mathematics, Vol. 344.
Springer–Verlag, New York, 1973.

[Troelstra 77]
Ann S. Troelstra.
Choice Sequences: A Chapter of Intuitionistic Mathematics.
Claredon Press, Oxford, 1977.

[Turner 84]
Raymond Turner.
Logics for Artificial Intelligence.
Halsted Press (John Wiley & Sons), New York, 1984.

[Weyhrauch 80]
R. Weyhrauch.
Prolegomena to a theory of formal reasoning.
Artificial Intelligence, v. 13, (1980),
pages 133–170.

[Whitehead & Russell 25]
A. N. Whitehead and B. Russell.
Principia Mathematica, Vol. 1.
Cambridge University Press, Cambridge, MA, 1925.

[Wos et al. 84]
Larry Wos, R. Overbeek, L. Ewing, and J. Boyle.

Automated Reasoning.
Prentice–Hall, Englewood Cliffs, NJ, 1984.

[Zucker 75]
J. Zucker.
Formalization of classical mathematics in AUTOMATH.
In *Colloques Internationaux du Centre Nationale de la Recherche Scientifique*, No. 249,
1975, pages 136–145.

Index