Representing Constructive Theories in High-Level Programming Languages (Ph.D. Thesis)

Ryan Stansifer

TR 85-664
March 1985

Department of Computer Science
Cornell University
Ithaca, New York 14853

REPRESENTING CONSTRUCTIVE THEORIES IN
HIGH-LEVEL PROGRAMMING LANGUAGES

Ryan Dale Stansifer, Ph.D.
Cornell University, 1985

This thesis concerns certain constructive theories we call programming logics. A programming logic is a strongly typed, functional programming language. Its semantics can be defined by a set of rewrite rules. We consider only programming logics in which predicate logic can be embedded, thus justifying the use of the term "logic." Furthermore, we single out those programming logics for which proofs of existential formulas have a canonical form explicitly revealing the witness. Thus, only constructive theories can be represented in these programming logics, and proofs in programming logics are executable.

Three programming logics are described in this thesis. The first is based on HA^{ω} (Heyting arithmetic of the omega order). The second is based on a Martin-Löf style type theory. The third is based on intuitionistic set theory. These programming logics have been implemented in the programming language ML (described in detail in this thesis). The core of each implementation is the same, and this logic engine, written in ML, is described in an appendix.

Programming logics would be woefully inadequate as a basis for automatic deduction without provision for reasoning at a higher plane. We show how to implement proof strategies for programming logics. As an example, we show how PROLOG, or more precisely, linear input resolution could be implemented as a proof strategy for a programming logic. Finally, we demonstrate how certain elements of classical logic can be used in proof development and then eliminated in these cases.

The experience gained in implementing these programming logics and described in this thesis contributes to the design of theories in which proofs are to be executable. The techniques of implementation demonstrated in this thesis can be used to build prototypes of a wide variety of theories. The ease in experimenting with new theories and the clarity with which the underlying mechanisms can be discussed are due to the representation of these theories at the level of abstract syntax and the directness by which the representation has been implemented in ML.

REPRESENTING CONSTRUCTIVE THEORIES IN HIGH-LEVEL PROGRAMMING LANGUAGES

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by

Ryan Dale Stansifer

June 1985

Acknowledgements

The origins of this undertaking go back to the PL/CV project. Some of the people connected with the project helped me in starting out. In particular, I would like to mention Scott Johnson, Dean Krafft, and Dan Zlatin.

This thesis would not have been possible without the PRL project. I have been helped directly and indirectly by the entire PRL staff: Stuart Allen, Joe Bates, Mark Bromley, Jim Cremer, Bob Harper, Todd Knoblock, Nax Mendler, and Jim Sasaki. The PL/C project, the PRL project, and my thesis have all been directed by the same person, Bob Constable, to whom I owe the most for his guidance and instruction. His unfailing optimism and "pep talks" are also greatly appreciated. Thanks also belong to my other committee members: Alan Demers and Richard Platek. Several conversations with Richard Platek and David Sutherland made me think about what I was doing.

I wish to thank Ralph Johnson for his careful readings of drafts of this thesis. He did his best to suggest ways to make sense out of nonsense.

A very important person in this enterprise is my wife Betsy. She provided editorial support and had to put up with a great deal while I agonized over this or that. I owe her a special thanks for love and comfort.

Contents

Chapter I

Introduction

The constructive theories we wish to consider in this thesis are called programming logics. These theories are interesting because proofs in them can be executed just like a program in, say, LISP. By representing these mathematical theories in the programming language ML we can actually write the interpreter for a programming logic. We have implemented three programming logics, and they are described in this thesis.

Programming logics are both programming languages and logics. An important application of this synthesis is found in the area of program verification. Writing a program in a programming logic is like finding a proof of the specification. This is harder than writing a quick and dirty program, but the result is a provably correct program.

We believe that, despite the fact that proving is harder than programming, practical programming (or proving) systems can be built. The reason for this optimism is that it is possible to interface the logic with a language to express proof strategies, theorem-proving techniques and decision procedures. It is not unreasonable to assert that "real" mathematics takes place at these higher levels. An ideal system supporting higher-level proof construction would make a good mathematician's apprentice or theorem-proving engine. And, if it were built on constructive principles, it would make a practical development environment for correct programs.

1.1 Programming logic

A programming logic is a strongly typed, functional programming language. Its semantics can be defined by a set of transformation rules. The rules convert expressions in the language to other expressions. This is like pure LISP, the semantics of which can be described by functions from S-expressions into other S-expressions. We are only interested in certain sets of transformation rules. We have in mind that the transformation rules should reduce the complexity of expressions, for some complexity measure (a function from expressions to a well-founded set). Hence, all well-formed expressions in a programming logic have a normal form. The semantics of a programming logic can be viewed as a rewrite system. In the language of rewrite systems [Huet and Oppen 1980], we say the semantics is noetherian or finitely terminating. We further restrict the semantics of programming logics to have the Church-Rosser property, thus assuring unique normal forms. We will not deal with these issues in this thesis, but rather rely on known results in the literature [Howard 1970, Stenlund 1972, Troelstra 1973].

One language that meets the requirements above is typed lambda-calculus. Typed lambda-calculus has just one transformation or reduction rule. The so-called beta-reduction rule is:

$$(\lambda x : T.b)(t) \quad \longmapsto \quad [x/t]b$$

where x is a variable of type T, t is a term of type T, and $[x/t]b$ means that t is substituted for x in the body b.

Typed lambda-calculus is a language for expressing algorithms; it is not adequate to express reasoning. If we are to model reasoning, we must have constructs that correspond to propositions: $A \,\&\, B$, $\mathit{Some}\, x \,.\, P(x)$, and so on. So we require that predicate logic be embedded. We mean by this that the rules of inference for the predicate calculus should be constructs in a programming logic. Hence proofs can be expressed in a programming logic, and they will have a normal form, just like every other expression in the language. Some proofs will be proofs of existential propositions. We require that the normal forms of existential propositions $\mathit{Some}\, x \,.\, P(x)$ be of such a form that it is possible to pick out a t and a proof $P(t)$. We call this the *constructivity requirement.*

The result of the constructivity requirement is that proofs will be programs. The idea that proofs could be used to develop programs in modern programming languages emerged in the early 1970s [Constable 1971, Bishop 1970]. For further discussion see [Bates and Constable 1983, Martin-Löf 1982].

It was around the same time in the early 1970s that a procedural interpretation for predicate calculus was developed, leading to PROLOG and a different approach to programming [Colmerauer et al. 1972]. Programming logics differ greatly from PROLOG, yet they are strangely similar in that they each give a semantics to predicate logic. We will see the exact relationship between programming logics and PROLOG in Section 4.2.

1.2 Programming logic as a verification methodology

An issue of much practical interest is the reliability of software. Although software is used in many critical applications, little of it has been formally verified to meet its specifications. The work in this thesis has a bearing on this topic, for programming logics offer a new paradigm for program verification.

A common method for proving that programs meet specifications is based on Hoare logic. In Hoare logic the objects of interest are triples: $\{p\}\ S\ \{q\}$, where p is called the precondition, q the postcondition and S is some program fragment. The precondition and postcondition are usually first-order formulas of predicate logic and are intended to characterize the state of the machine before and after performing the operation S. The meaning of the Hoare triple $\{p\}\ S\ \{q\}$ is that q is guaranteed to be true about the state of the machine after doing S if p was true before and S terminates. To verify a program we must have rules describing how the statements of the language affect the state of the machine. Here, for example, is the typical rule for the if-statement.

$$\frac{\{p \,\&\, b\}\ S_1\ \{q\}, \quad \{p \,\&\, \neg b\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

The sequential execution of the machine gives rise to a proof obligation for composing program statements. The postcondition of one statement must imply the precondition of the immediately following statement.

The whole process is nearly automatic. If the program is annotated with loop invariants, the proof obligations or verification conditions can be generated automatically. A theorem prover for first-order logic can (some of the time) prove these verification conditions. The Stanford PASCAL verifier is built along these lines [Stanford verification group 1979]. In this way the partial correctness of the program is verified. At least, that is how it is supposed to happen. In practice, there are some drawbacks. Not only is human intervention (in the form of lemmas) required to untie knotty proofs in first-order logic, but also the reasoning takes place at a low level. This is especially true in regard to pointers, which are used to implement higher level constructs like lists and trees. We quote from members of the Stanford verification group [Luckham and Suzuki 1979]:

> The main problems encountered in specifying pointer programs ... result from the use of pointers to implement complicated structures (e.g., balanced binary trees). Specifications require the use of high level concepts describing properties of structures that do not have a standard axiomatization. So verification of pointer programs quite often depends on finding an elegant set of specification concepts and their axioms. Of course, the need to formalize high level concepts can also occur in verifying array programs.

It appears to be a waste of intellectual effort to first transform the high-level notion of an algorithm conceived in the mind to a low-level implementation using pointers, and then find an elegant set of specification concepts to verify the program. If the two processes could be combined, a substantial savings of effort might accrue.

Besides the axiomatic method of the Stanford verifier, there is another approach to program verification based on the theory of fixpoints in complete partial orders. This theory was proposed by Dana Scott [Scott 1970] and developed by Robin Milner [Milner 1972]. This method, the method of denotational semantics [Stoy 1977, Tennent 1976], views programs (recursive functions) as fixpoints of equations. For example, let f_P be the least fixpoint of the program

$$P: F(x) \Leftarrow \textbf{if } p(x) \textbf{ then } x \textbf{ else } F(F(h(x))).$$

Properties of this function can be proved; for instance, one can show that $f_P f_P = f_P$ regardless of the underlying functions p and h. The primary method of proof is fixpoint induction.

In a formal theory, like the theory of fixpoints, the proofs can be written out in complete detail and checked by machine. This has been done for the theory of fixpoints in the LCF (logic of computable functions) system [Gordon et al. 1979]. The fixpoint theory is formalized in the system, and the user is assisted in deriving proofs about functions.

A programming logic can also be viewed as verification methodology. The underlying philosophy of a programming logic is that reasoning and correct programming are the same process. The ultimate goal is to find a formal language which facilitates expression of both logical and algorithmic content. Traditionally these two modes of expression have had their separate languages: classical first-order logic on the one hand, and the plethora of programming languages on the other. A programming logic is a single language to meet both demands. Since a programming logic can express the desired functionality, as in "integers q and r such that $x = y * q + r$," it is a specification language. Since a proof in a programming logic can be executed like an algorithm, it is a program. There is then no question that the program meets the specification.

In this thesis we will see that a programming logic can be designed that will permit reasoning about data structures like lists and trees. Using a language like PASCAL requires reasoning at the level of the pointers that implement these data structures. Reasoning about these data structures directly is an improvement.

1.3 Automated deduction

Deduction is an important part of program verification. It is an important part of any activity that requires mathematical certitude. One area of great interest to which this certainly applies is expert systems, where the knowledge of experts is used to deduce the answers to specialized questions. For best effect, both program verification and expert systems require that this deduction be performed automatically, or with as much assistance as possible by computer.

Systems that do automatic deduction can be classified along a spectrum ranging between two extreme poles. On the one end is theorem proving. Here the goal is to make deduction totally automatic, not just for making inferences in response to queries of data bases or expert systems, but also for new results in mathematics [Wos 1982]. This goal is unattainable, so theorem provers need guidance provided by either users or heuristics. At the other end of the spectrum lies verification. Any reasonable formal system uses a definition of deduction that permits a test for a single step in the deduction. A completed deduction or a proof can then be checked. Strictly taken, verification is overly cautious, and any practical system will do at least some of the details automatically.

Verification should not be dismissed too soon, as it is not without its challenges. A mathematician's apprentice that checks all the details has not been realized yet. Such a system would expand the range of theorems the working mathematician can prove. The tedious proof of the four-color theorem is evidence of both the need for a mathematician's apprentice and the difficulty in building one. Although there have been systems [Suppes 1981] that "know" some mathematics and can assist those trying to learn it, there are few (PRL is one) that are open-ended. The challenge lies in finding the way to let mathematical insight (at which humans do well) control the management of details (at which computers excel). The crafting of a verification system to be user-friendly, efficient, natural and powerful is a goal that may produce man-machine systems that prove more "interesting" theorems than theorem-proving systems. It is a race to the center of the spectrum.

The basic underlying question which unifies the study and construction of systems of automated deduction is: how does one do mathematics? As soon as the question is asked, one is confronted with another question: what is mathematics? There have been many proposed answers.

Is it set theory? This is one of the first answers that come to mind. Many books on mathematical subjects begin with a review of the basic concepts of set theory. These authors are tacitly adopting the view that everything they wish to express could, in principle, be said in set theory. More often than not this is false, as the biggest deficiency in classical set theory is the absence of a fundamental notion of algorithm. Some of the greatest results in mathematics benefit by being expressed in the language of algorithms: the greatest common divisor, fundamental theorem of arithmetic, uncountability of the continuum, Bolzano-Weierstrass theorem, and so on.

Is mathematics number theory? The natural numbers are well understood and could serve as the basis from which the edifice of mathematics is constructed. Rational numbers were constructed from integers, real numbers from rationals, maybe everything of interest can be reached starting with the natural numbers. This brings to mind the German mathematician Leopold Kronecker's quip, "God made the integers, all the rest is the work of man."

Is mathematics type theory? Perhaps mathematics is built of building blocks different from those in set theory or number theory. Some have suggested that mathematics is logic itself. This school of mathematical thought, known as "logicism," is associated with Bertrand Russell [Putnam 1967]. This view has now fallen into disfavor, but more recently others have suggested that logic and algorithms may be built from a common notion, that of a type.

Whatever the building blocks—sets, numbers, types—we believe that the notion of construction is important.

We cannot wait for the answer to the question of what mathematics is to be resolved—if it can ever be resolved. We must plunge right in and look at what deduction is in practice—as a creative process, a human endeavor.

One thing is certain, mathematics is not significantly advanced by striking out in all directions one deduction at a time. No new ideas are introduced this way; just never-ending streams of theorems. It has been obvious from the start [Gelernter 1963] that important theorems were not going to be found from a low-level approach. New ideas like diagonalization or the proof of the Bolzano-Weierstrass theorem emerge from the murky recesses of intuition. How do we program computers to find these insights? There is some literature to guide the search for answers [Polya 1954, Lakatos 1976], but it is difficult to put into practice. One notable effort to guide discovery in mathematics by gauging "interestingness" is described in [Lenat 1982]. This continues to be a big area of research in the field of artificial intelligence.

An important lesson must be drawn from this. If automatic deduction is to be useful, it must have the capability to work at a higher level. Something more than just symbol manipulation, more than just checking proofs, is needed. The ability to capture proof techniques and manipulate proofs is needed. This is applying metamathematics to automated deduction. We show in this thesis how to apply some metamathematics the end of assisting proof development in the context of programming logics.

1.4 Designing theories

Different programming logics may be suited for different purposes. It is by no means clear just what form a programming logic should take. Different constructs will be appropriate to meet different goals. Foundational questions may require a few elegant constructs. Demands for efficient execution may require numerous, ugly constructs that reflect the underlying hardware. One shudders to think of the data types for machine, integer arithmetic, to say nothing of floating-point arithmetic. Even when the right constructs are found it is not clear what form they should take. It is not surprising that the experience of program language design is of importance here.

In time, new functional programming constructs for non-determinism and non-terminating processes may be developed. These constructs may be brought into the fold of programming logics. Experimentation is needed to help design these programming logics. It is not always clear what constructs are needed and useful. Rapid prototyping of programming logics facilitates the evaluation of new constructs. This thesis contributes to this end by offering the experience of prototyping several diverse programming logics.

We have chosen a representation for programming logics at the level that John McCarthy [McCarthy 1963] has termed abstract syntax. This level has worked out well. It is high enough that the constructs of a programming logic can be represented naturally. This thesis relates the experience gained in building these constructs. The abstract representation is also at a low enough level that implementation details can be comprehended. All the relevant implementation details are contained in this thesis. Everything is included which is needed to build a programming logic and start executing proofs as programs.

The abstract representation circumvents the irrelevant details about the concrete appearance of a programming logic. We can then concentrate on developing proofs in programming logics. Important ideas for developing proofs—refinement, tactics, theorem proving—all fit in well with this abstract representation. We will thus be able to easily connect disparate ideas. Yet the actual implementation in ML is never far away. As an example of a sophisticated proof development technique, we present in this thesis the ML program that develops a programming logic expression from proofs in classical mathematics.

1.5 Background

Of great relevance to this thesis is the PRL project at Cornell University. The PRL (Program Refinement Logic) project has the goal of forming a programming language out of the constructive part of mathematics. This programming language will be unlike previous ones. It will be a new generation programming language. These ideas have inspired the programming logics of this thesis. The history and goals of the PRL project are discussed in [Constable 1983].

The ambitious aims of the PRL project include managing a library of accumulated theorems and proof techniques, extracting efficient code [Sasaki], and providing a proof development environment that is easy to use and understand. Already one version of PRL (Lambda PRL) has been implemented and is running at Cornell; documentation is found in the programmer's manual [PRL staff 1984]. A new version (Nu PRL) has just been finished [Constable and Bates 1984].

PRL is the latest system to be developed out of the ongoing research at Cornell University. One of the early landmarks in this research is *A Programming Logic* [Constable and O'Donnell 1978]. This design influenced the PL/CV and AVID verifiers [Johnson 1980, Krafft 1981]. In these verifiers a PL/C program was built by introducing the imperative language constructs. The assertion parts of the logic were shielded in comments, so that the text could be compiled without change by a PL/C compiler. A different method is used by PRL to obtain a program. In PRL a program is extracted from a proof. That is to say, the computational content is compiled into a LISP program. The programming logics of this thesis, on the other hand, *are* themselves programs. There is no other internal part or structure that holds the computational content. There is an interpreter that takes an expression $f(x)$ and returns the answer y. Of course, one may want to translate the computational content of programming logic expressions into other languages. LISP is the obvious and easiest choice for a target language. The extraction process of [Bates 1979] describes the necessary details. But the techniques for compiling LISP code [Allen 1978] can be used to compile expressions of programming logics (a functional programming language like pure LISP) into any language desired.

Also of great relevance to this thesis is LCF (Logic of Computable Functions). This formal system for reasoning about programs introduced the programming language ML as its meta-language. LCF is important because it used the type security of ML to represent a (nonconstructive) logic and because it introduced the idea of tactics, or proof strategies.

1.6 What has been implemented?

This thesis represents the experience of implementing three programming logics. All three programming logics have been implemented on the same logic engine written in the ML programming language. The logic engine is composed of ML type definitions and functions. The constructs of each programming logic are ML function definitions that make use of the logic engine. By running ML, loading the logic engine, and loading the appropriate construct files, the programming logic is simulated. All the constructs of the particular programming logic are available to use. Expressions can be built up, evaluated and printed out.

The logic engine is the foundation for building different systems, tailored to the needs and wishes of the theory designer. For example, programming logics are strongly typed, so the available types are important to a system's utility. The logic engine permits primitive types and type operators to be introduced. If the designer of a theory is interested in natural numbers, a type for natural numbers can be introduced. If the interest is in sets, then a type for sets can be introduced.

1.7 Organization

This thesis can be logically divided into four parts. The first part is a description of the programming language ML. This programming language will provide the formalism that will unify all other discussions. The second part deals with representing logic. Metamathematics is the concern of the third part. The fourth part concerns implementation details and has been relegated to the appendices.

The first topic in this thesis is a description of the programming language ML. Chapter Two is devoted to this purpose. In place of set theory, the traditional lingua franca of mathematics, we have chosen another, more suitable, language—ML. Since ML is a new and unfamiliar programming language to many, it is described in detail. This provides the background necessary to thoroughly understand the implementations presented in this thesis.

But those are not the only benefits of studying ML. We will discover that ML is a language which has more than a few similarities with programming logics. Also, ML is convenient as a meta-language for formal systems. ML, as the name suggests, was named for meta-language.

The second part is the heart of the thesis, and it is contained in the third and fourth chapters. These chapters describe three programming logics that cover a wide range of applications. The first is suited for number theory. The second is similar to a type theory of Martin-Lõf. The third is a formalized set theory.

The majority of the effort will go toward describing AHA, the programming logic for number theory. The constructs of AHA will be introduced, and the interpreter which evaluates proofs will be described. We will show how to compute with the interpreter. The other two programming logics introduced show that the same logic engine can be used for other theories.

The subject of the third part is applied metamathematics. Applied metamathematics studies how programs can be written to manipulate mathematical objects such as proofs. We will be most interested in programs that assist in finding proofs.

In this third part we will see how to implement a completely different style of proof development, refinement style. We will see how to integrate automatic theorem proving with programming logic. In particular we will examine the relationship PROLOG has to programming logics. Finally we look at a method for using classical methods to build constructive proofs. We describe the ML program that implements this method.

The appendices make up the fourth part of this thesis. The first appendix contains the ML code for the logic engine. The other appendices list the rules of inference and the reduction rules for the three programming logics.

Chapter II

Description of the programming language ML

The programming language ML was originally designed by Robin Milner for use as the metalanguage in the LCF verification system. The language is interesting in its own right and is versatile enough to compete with LISP. It is, in fact, much like LISP—but better in some ways. It is strongly typed and has a more readable syntax. ML descends from the languages POP-2 [Burstall et al. 1971], ISWIM [Landin 1966] and HOPE [Burstall et al. 1980]. The syntax of ML has not been completely standardized. The ML syntax described here is that of the UNIX version of ML written by Luca Cardelli at Bell Laboratories [Cardelli 1983]. There are several other versions of ML, and the language is gradually evolving to a standard.

In this chapter we describe as much of ML as appears in the rest of this thesis. This amounts to almost all of ML except for the non-functional part: references and arrays. We discuss the other dialects of ML in the last section.

The distinguishing characteristics of ML are

- an interactive dialog with the user;
- a system of strong typing;
- its functional style;
- an exception-trap mechanism;
- a mechanism for defining abstract data types;
- provision for separately compiled modules.

2.1 Description of ML

Like LISP, ML is an interactive programming language. An ML session consists of a dialog between the user and the system. The user types an expression terminated by a semicolon. The value of that expression and its type are returned by the ML system. ML prompts the user to input an expression with the "-" character, and responds on the following line. Here are some examples:

```
- (3 + 5) * 2;
  16 : int
- "this is a string";
  "this is a string" : string
- (), true, 1;
  (),true,1 : unit * bool * int
- if 1=2 then 3 else 4;
  4 : int
```

The "*" symbol is used for multiplication and as the cartesian product operator for data types. ML has just the four primitive data types used in the examples above: **unit** (with **()** as the only element of the type), **bool** (the data type of boolean values), **int**, and **string**.

Homogeneous lists of elements can be built and manipulated in ML as in the next examples. Lists are written in square brackets; **hd** and **tl** are the built-in operators for the head and tail functions on lists; and the LISP **CONS** function is the build-in, infix operator :: in ML. We also introduce comments, which in ML are enclosed in curly braces.

```
- [1;2;3;4];
  [1;2;3;4] : int list

- hd [1;2;3;4];  tl [1;2;3;4];      { 2 expressions on 1 line. }
  1 : int
  [2;3;4] : int list

- [8,true;5,false];  1 :: [2;3;4];
  [(8,true);(5,false)] : (int * bool) list
  [1;2;3;4] : int list
```

The last two lines typed in to the ML system each had two expressions. The system prints the response to each expression on a separate line. There is no prompt after the first line, should an input expression extend beyond one line. Notice that elements of list are separated by semicolons. The empty list is **[]**. Elements of tuples are delimited by commas. The type of a tuple is the cartesian product of the types of the elements.

Bindings. Next we illustrate how variables are given values. At the top level this is done using the keyword **val**, but there are various ways of making local bindings as well. In those rare cases where there is some confusion over which value of a variable to use, ML uses lexical scoping (sometimes called static binding).

```
- val a = 3;                       { Bind the value 3 to a.  }
  val a = 3 : int
- val a = 4 and b = 27,true;       { Make two bindings.      }
> val a = 4 : int
| val b = 27,true : int * bool
- val a,b = 4,(27,true);           { Make same 2 bindings.   }
> val a = 4 : int
| val b = 27,true : int * bool
- a + 6 where val a = 5 end;  a;   { N.B. no change in a.    }
  11 : int
  4 : int
- let val a = 5 in a+6 end;  a;    { Again no change in a.   }
  11 : int
  4 : int
```

The "let" and "where" constructs accomplish the same purpose—factoring out a local value. Such a local binding has no effect on the global value of the variable. Notice that when more than one binding is made the ML system prints each binding on a separate line; the first binding is preceded by the ">" character; after the first binding, the lines the system prints out begin with the "|" character.

Functional programming style. ML is a functional programming language. Functions in ML are created using the keyword `fun`. (LISP uses `LAMBDA` for the same purpose.) What is customarily written $(\lambda x.b)$ in the lambda notation is rendered `(fun x.`*b*`)` in ML. Functions are first class objects; they can be arguments to other functions and can be returned as values of functions. Functions can be bound to variables at the top level just as any other value can: `val f = (fun x.`*b*`)`. This way of defining the function `f` is given an alternate syntax in keeping with the customary way of writing definitions: `val f(x) =` *b*. Here is a simple function definition.

```
- val f (n) = n + 1;         { Function definition. }
  val f : int -> int

- f;                         { A function object. }
  fun : int -> int
```

```
- f (2);                     { Function application & evaluation. }
  3 : int

- f 2;                       { Parens not needed, if arg a token.  }
  3 : int
```

Notice that the printing representation of all functions in ML is the same, namely just the keyword **fun**.

Here are some more interesting examples of functions.

```
- val g (x,y) = (x+y) div 2;  g(f(7),2+2);
  val g : (int * int) -> int
  6 : int

- val h (x) = (fun y. x(y+1)+1);  h(f);  h(f)(2);
  val h : (int -> int) -> (int -> int)
  fun : int -> int
  5 : int

- val rec fact (n) = if n=0 then 1 else n*fact(n-1);
  val fact : int -> int

- val f (n) = g (fun x.n+1) (34)
                where val g (h) (n) = h(n) end;
  val f : int -> int

- f(1);
  2 : int
```

The keyword **rec** is always needed in a recursive definition. In the example of the factorial function above, the identifier **fact** would be unbound in the body of the definition, if the keyword **rec** had been omitted. Mutually recursive definitions are made using the **and** construct like in the next example.

```
- val rec Odd (n) =
    if n=0 then false else Even (n-1)
  and Even (n) =
    if n=0 then true else Odd (n-1);
```

```
> val Odd : int -> bool
| val Even : int -> bool
```

Strong typing. An important feature of ML is its polymorphic type system. Where the type of an argument does not matter, it need not be specified. Hence one need not define one identity function exclusively for integers and another for strings. One identity function will do—one for any type.

```
- val id (x) = x;               { Polymorphic identity function }
  val id : 'a -> 'a
- val switch (x,y) = (y,x);
  val switch : ('a * 'b) -> ('b * 'a)
- val comp (f,g) (x) = f (g (x));     { Functional composition }
  val comp : (('a -> 'b) * ('c -> 'a)) -> ('c -> 'b)
```

Identifiers beginning with a single quote always denote type variables in ML.

Exceptions. Programs in ML can raise exceptions and then trap them. This is similar to the catch and throw mechanism in LISP. A function, instead of returning a value, may signal some abnormal condition. For example, the built-in division function signals division by zero whenever the divisor is zero. Further execution halts and ML prints a message to the user at the top level.

```
- 1 div 0;           { Build-in integer division function div }
  Exception: div
- val f (n) = if n<0 then escape "Neg arg" else fact (n);
  val f : int -> int
- f(3);
  6 : int
- f(-3);
  Exception: "Neg arg"
```

Exceptions can be trapped before they reach the top level. This permits the computation of an alternate value for an expression should it raise an exception. The syntax of the trapping mechanism calls for a question mark after the expression that may signal an exception and before the alternate expression to be evaluated in the event an exception is signaled.

```
- (1 div 0) ? 45;
  45 : int
- val g (n) = f(n) ? f(-n);      { Uses definition of f above }
  val g : int -> int
- g(3);
  6 : int
- g(-3);
  6 : int
```

This is a good place to stop and discuss type and type inferencing in ML. We have already seen the primitive types in ML: unit, bool, int and string, and the type operators: `*`, `->` and `list`. (Notice that there is no union operator, as one might expect. We will see how to define one shortly.) It is never necessary to specify the type of any expression in ML (although it is permitted), since the type of any expression can be inferred, if the expression has a type. (Some expressions do not have a type, like `fun x. x(x)`. These expressions cause a type error to occur and are not legal ML expressions.) The method of type inference is spelled out in [Milner 1978].

Definition of abstract data types. Next we will introduce ways to build new types from old types. In these type definitions we will need to be able to write type expressions. We have already seen many type expressions, because ML prints one for every expression the user types to the system. They are formed from the primitive types using the three type operators listed above. For example, `bool list -> int` is a type expression.

ML has the capability to define new types. This can be done in two ways. The functions that construct the elements of a new type can be specified. This is a concrete type. An abstract type is defined by giving the constructors as well, but then all the functions that will ever make use of the constructors must be defined on the spot because the constructors will not be available outside the scope of the type definition. This insures controlled access to the type. We first give examples of a concrete types.

```
- type Color = Red | Blue | Yellow;
> type Color = Red | Blue | Yellow
| con Red : Color
| con Blue : Color
| con Yellow : Color
```

Color is an enumerated type with three elements. **Red, Blue, Yellow** are constructors; they are really 0-ary functions or constants. Constructors need not have arity zero, as in the next example of a tree. Furthermore notice in this example that recursive types can be defined, but they require the keyword **rec** in the definition.

```
- type rec Tree = Leaf of int | Node of int * Tree * Tree;
> type rec Tree = Leaf of int | Node of int * Tree * Tree
| con Leaf : int -> Tree
| con Node : (int * Tree * Tree) -> Tree
```

Leaf is one of two constructors defined here for the type **Tree**. Given any integer n, **Leaf**(n) is a tree. After the keyword **of** in the definition of type comes the domain of the constructor. The range of the constructor is always the type being defined. The constructors in a type definition are separated by a vertical bar. The next examples use the definition of tree just given.

```
- val l1 = Leaf (4);
  val l1 = Leaf 4 : Tree

- val l2 = Leaf (5);
  val l2 = Leaf 5 : Tree

- val tr = Node (6, l1, l2);
  val tr = Node(6,Leaf 4,Leaf 5) : Tree
```

An important part of the ML language which has not yet been mentioned is pattern matching. This is often used in conjunction with the case statement to break apart the structure of a type. An element of a type is taken apart in the case statement. It is matched against the pattern consisting of variables and constructors in each branch of the case statement. This is how destructuring is accomplished and explains the lack of "destructors" or "selectors" in the language. We give several examples of this using the definition of the type **Tree** above.

```
- val rec Preorder (t: Tree): int list =
    case t of
      Leaf n       .      [n]   |
      Node (n, lt, rt). n :: (Preorder lt) @ (Preorder rt);
  val Preorder : Tree -> (int list)
```

The infix operator @ is the ML append function.

```
- val f (t: Tree): int =
    case t of
        Leaf n  .                      n    |   { A leaf }
        Node (n, Leaf _, Leaf _).  n    |   { Two leaves }
        Node (n, Node _, Leaf _).  n    |   { A skewed tree }
        Node (n, _, _). n;              { Everything else }
  val f : Tree -> int

- val rec EqTree (t: Tree, s: Tree): bool =
    case (t,s) of
      (Leaf n, Leaf m)              . n=m |
      (Node (n,t1,t2), Node (m,s1,s2)).
        if n=m
          then if EqTree (t1,s2) then EqTree (t2,s2) else false
          else false                     |
      (_,_) . false;
  val EqTree : Tree * Tree -> bool
```

In each branch of the case statement the object is matched against a pattern. If the match succeeds, then any variables in the pattern are bound to the appropriate subparts of the object. The "_" character is the wild card pattern which matches any object. The cases are expected to be disjoint and cover all the possibilities, but this is not checked. If at run time no case is found to match, then a "match" exception is raised.

The next example is an abstract type definition of a tree. The constructors **Leaf** and **Node** will not be available outside the scope of the abstract type definition.

```
- abstype rec Tree = Leaf of int | Node of int * Tree * Tree
  with val MakeLeaf n = Leaf n;
  val MakeNode (n,t1,t2) = Node (n,t1,t2);
  val Label t =
    case t of
      Leaf n. n |
      Node (n,t1,t2). n;
  val RightSubTree t =
    case t of
```

```
    Leaf n. escape "Leaf" |
    Node (n,t1,t2). t2
  end;
> abstype Tree
| val MakeLeaf : int -> Tree
| val MakeNode : (int * Tree * Tree) -> Tree
| val Label : Tree -> int
| val RightSubTree : Tree -> Tree
```

Here are some examples of trees.

```
- val l1 = MakeLeaf (4);
  val l1 = - : Tree

- val l2 = MakeLeaf (5);
  val l2 = - : Tree

- val tr = MakeNode (6, l1, l2);
  val tr = - : Tree
```

ML uses a hyphen as the print representation of an abstract type.

The user can define new type operators. One of these type operators that can be defined is the union operator.

```
- type ('a,'b) Union = InL of 'a | InR of 'b;
> type ('a,'b) Union = InL of 'a | InR of 'b
| con InL : 'a -> (('a,'b) Union)
| con InR : 'b -> (('a,'b) Union)

- InL (3);  InR (true);  { Elements of different union types. }
  InL 3 : (int,'a) Union
  InR true : ('a,bool) Union
```

Modules. It is possible to gather a collection of relevant functions and type definitions together into what is called a module in ML. These can then be compiled and used in other modules. We give a sketch of a hypothetical module in order to introduce the syntax of a module.

```
module Graphs
body

  abstype Vertex = ...
  abstype Edge = ...
  abstype Label = ...
  abstype Graph = ...
  val FormEdge (v1: Vertex, v2: Vertex): Edge = ...
  val Label (v: Vertex): Label = ...
  val Endpoints (e: Edge): Vertex * Vertex = ...
  val OutEdges (v: Vertex, G: Graph): Edge list = ...
  ...
```

This module might be imported for use in another module of graph algorithms.

```
module GraphAlgorithms
includes Graphs
body
  val IsConnected (G: Graph): bool = ...
  val ShortestPath (G: Graph, v1: Vertex, v2: Vertex):
          Edge list = ...
  val CongruenceClosure (G: Graph, R: Vertex*Vertex->bool):
          Vertex*Vertex->bool = ...
  ...
end
```

2.2 Summary of the language syntax

We conclude the description of the ML programming language by giving a brief BNF review of the basic syntax of the the language.

The following metasyntax is used in the description. We use $\{A\}^+$ to mean one or more instances of A. We use $\{A\ /\ B\}^*$ to mean zero or more instances of A separated by B. (The separator B only occurs if there are two or more instances of A.) We use $\{A\ /\ B\}^+$ to mean one or more instances of A separated by B. We use $\{A\ /\ B\}^2$ to mean two or more instances of A separated by B.

Declaration ::=

val [**rec**] *ValueBinding* |

type [**rec**] *TypeBinding* |

abstype *[* **rec** *] TypeBinding* **with** *Declaration* **end** |

export {*ExportList*}$^+$ **from** *Declaration* **end** |

Declaration ; *Declaration*

ValueBinding ::=

Pattern = *Expression* |

ValueBinding **and** *ValueBinding*

TypeBinding ::=

{*ConstructorIdentifier [* **of** *Type]* / | }$^+$ |

TypeBinding **and** *TypeBinding*

Pattern ::=

SimplePattern |

ConstructorIdentifier SimplePattern |

{*Pattern* / , }^{2}

SimplePattern ::=

_ |

Identifier |

ConstructorIdentifer |

[{*Pattern* / ; }*] |

(*Pattern*)

Expression ::=

SimpleExpression |

Expression SimpleExpression |

escape *Expression* |

if *Expression* **then** *Expression* **else** *Expression* |

{*Expression* / , }^{2} |

let *Declaration* **in** *Expression* **end** |

Expression **where** *Declaration* **end** |

case *Expression* **of** *Match* |

Expression ? *Expression* |

fun *Match*

SimpleExpression ::=

Identifier |

ConstructorIdentifier |

[{*Expression* / **;** }* **]** |

(*Expression* **)**

Type ::=

TypeVariable |

TypeIdentifier |

{*Type* / *****}^{2} |

Type **->** *Type* |

(*Type* **)**

Match ::=

{*Pattern* **.** *Expression* / **|** }$^+$

ExportList ::=

abstype {*TypeIdentifier*}$^+$ |

type {*TypeIdentifier*}$^+$ |

val {*Identifier*}$^+$

Module ::=

module *Ident*

[**includes** {*Ident*}$^+$ *]*

body *Declaration* **end**

2.3 Dialects of ML

The ML that we have described in this chapter differs significantly from the language used as the meta-language in LCF [Gordon et al. 1979]. We will call this original dialect of ML *Edinburgh ML*, to distinguish it from the ML we have described in this chapter, *UNIX ML*. An implementation of Edinburgh ML in LISP has been developed at Institut National de Recherche en Informatique et en Automatique (INRIA) in France. This implementation forms the basis of Cambridge LCF and is used in the PRL project at Cornell University [Constable et al. 1984].

There are a fair number of syntax differences in Edinburgh ML, enough so that it is confusing to switch back and forth. In Edinburgh ML the key word `let` is used instead of `val`, `letrec` instead of **val rec**, **absrectype** instead of **abstype rec**, and **token** instead of **string**. Back quotes (') delimit strings instead of double quotation marks. Expressions in Edinburgh ML end with two semicolons not one. The type with one element is . (a period) and not **unit**. The cross product operator is **#** instead of *****. Edinburgh ML has a built in union operator **+** for types.

The biggest difference between the two dialects lies in abstract type definitions. In Edinburgh ML when an abstract type T is defined, two functions **abs**T and **rep**T are declared for the scope of the type definitions. We give an example; first in UNIX ML and then in Edinburgh ML.

```
abstype Time = mkTime of int * int
  with val maketime (hrs,mins) =
    if hrs<0 or 23<hrs or mins<0 or 59<mins
      then escape "fail"
      else mkTime (hrs,mins);
  val hours (mkTime (hrs,mins)) = hrs;
  val minutes (mkTime (hrs,mins)) = mins
end;
```

Here is the same type definition in Edinburgh ML. Notice the two implicitly defined functions **abstime** with type **int # int -> time** and **reptime** with type **time -> int # int**. These are maps between the type and its representation.

```
abstype time = int # int
  with maketime (hrs,mins) =
    if hrs<0 or 23<hrs or mins<0 or 59<mins
      then fail
      else abstime (hrs,mins)
  and hours t = fst (reptime t)
  and minutes t = snd (reptime t);;
```

After this definition of the abstract type **time** the functions

```
maketime : int # int -> time
hours : time -> int
minutes : time -> int
```

are defined.

Another of the substantial differences between Edinburgh ML and UNIX ML is the more extensive matching facility in UNIX ML that has been borrowed from HOPE. Since Edinburgh ML has no constructor functions, most destructuring is done explicitly (as in the destructors **hours** and **minutes** above). The only exception is for the built-in types lists and tuples.

Another important difference is modules. Edinburgh ML has no modules. Modules are the area of current development.

At this time a standard for the core of ML (no modules) has been set [Milner]. The standard is not significantly different from what we have presented as UNIX ML. UNIX ML greatly influenced the standard and has itself evolved toward the standard.

Chapter III

The AHA programming logic

In this chapter we represent a variant of number theory in ML. Representing a formal theory assigns objects in the corporeal world to ideas in the Platonic world. We pick a representation at the level of abstract syntax. Abstract syntax captures the essential elements of the constructs, while ignoring the details. The programming language ML provides the mechanism to realize directly the abstract syntax of a formal theory.

The starting point of this chapter is the formal theory PPLAMBDA. This classical theory was implemented in ML as part of the LCF system. We will examine how this was accomplished, because the technique used in that implementation was used in the implementation of the three programming logics discussed here. We will describe these programming logics in this chapter and the next. The first programming logic is called AHA, and is a theory concerning natural numbers. We will describe this programming logic in detail and discuss the interpreter for it. Several examples are worked out in AHA. The integer division function is used to demonstrate how a proof can be executed like a program. The chapter ends with an example of a program to compute the maximum segment sum.

3.1 PPLAMBDA

The LCF system is a proof checker for proofs about computable functions [Gordon et al. 1979, Milner 1979]. The proofs are written in a formal system called PPLAMBDA (which stands for polymorphic predicate lambda calculus). The PPLAMBDA theory formalizes in detail the fixpoint theory in complete partial orders as found in, for example, [Manna 1974] or [Stoy 1977]. PPLAMBDA includes all the rules of reasoning and facts about partial orders which are necessary to prove properties of functions. For instance, there is a rule of inference to prove the conjunction of formulas:

$$\frac{A, \quad B}{A \,\&\, B}$$

This rule means that if one has a proof of A and a proof of B one can combine the premises to get a proof of the conjunction $A \,\&\, B$. Such a rule is a common law of reasoning. We shall see this rule again. Naturally, PPLAMBDA contains some facts about the domains. Here is an axiom concerning the flat domain of boolean values:

$$All\, b \,.\, b = \perp \mid b = Tr \mid b = Fl$$

The symbol representing the bottom element of a partial order is "$\perp$". The symbols "Tr" and "Fl" are two values in the boolean domain. (Actually, to be perfectly accurate, LCF uses UU, TT and FF respectively.)

Central to the theory is the fixpoint induction rule which provides the means to prove certain properties P about the fixpoint of functionals F. This is what the rule looks like:

$$\frac{P(\perp), \quad All\, f \,.\, P(f) \Rightarrow P(F\, f)}{P(FIX\, F)}$$

The symbol "FIX" stands for the fixpoint operator. Most useful theorems in PPLAMBDA use this rule.

It is worth pointing out that in the presence of the axiom

$$All\, b \,.\, b = \perp \mid b = Tr \mid b = Fl$$

certain formulas are provable that we shall wish to eschew in other theories. For instance,

$$All\, f \,.\, All\, x \,.\, f(x) = \perp \mid f(x) = Tr \mid f(x) = Fl$$

is provable in PPLAMBDA. Since the fixpoint operator permits the definition of arbitrary partial recursive functions ϕ, the implication is that

$$All\, x \,.\, \phi(x) = \perp \mid \phi(x) = Tr \mid \phi(x) = Fl$$

although there is no way of knowing which case holds. This makes it impossible to write an interpreter for proofs in PPLAMBDA as we shall do for programming logics. There appears to be no obvious way to remedy this point, as the cases rule is necessary to the PPLAMBDA theory.

The complete set of rules in PPLAMBDA is not of use here, but can be found in [Paulson 1983] where they are given in a style that facilitates comparison with the programming logic presented later in this chapter. It is the method of representing PPLAMBDA in ML that is of interest here, since this approach can also be taken to implement a programming logic. Although PPLAMBDA was not implemented strictly in ML, it can be. The terms, formulas and proofs of PPLAMBDA are all represented as ML abstract data types. The rules of the PPLAMBDA theory are then represented as constructors of the type theorem. The crucial point is that the only way to create an element of the type theorem is through these constructors which, presumably, implement valid rules of logic. In order to best understand how the implementation could be accomplished, we will present a fairly complete example of the propositional logic.

Propositional logic. We will now represent a part of propositional logic in ML. Propositions, formulas and proofs will be represented by elements of three different ML data types. These type definitions will be given shortly. The ML types for formulas and proofs will be recursive just like their definitions as syntactic classes in mathematics.

The actual implementation of a proposition is not important here, but to be concrete in this example of propositional logic we take propositions to be represented by ML strings. The following type definition in ML accomplishes this.

```
- type Proposition = MakeProp of string;
> type Proposition = MakeProp of string
| con MakeProp : string -> Proposition
```

The ML function `MakeProp` is a constructor of propositions (representations of propositions actually). Next we create a particular proposition for use later.

```
- val P = MakeProp ("P");    { A proposition }
> val P = MakeProp "P" : Proposition
```

Given a data type representing propositional statements we first define one for the formulas. The definition of formula follows the usual definition in mathematical logic. Every propositional statement is an atomic formula, and a pair of formulas can be made into another formula using the implication connective. Of course, we might be interested in other connectives or at least in a formula to represent falsehood, but implication shall suffice for this example. So we first need a constructor of formulas from propositions. Since these particular propositions are the only atomic propositions we name the constructor `AtomicFormula`. It is a function from the ML data type `Proposition` to data type `Formula`. We name the constructor that forms the compound formula out of two formulas `Imply`.

```
- type rec Formula =
    AtomicFormula of Proposition | { A proposition is a formula. }
    Imply of Formula * Formula;    { Make implication.          }
> type Formula = ...
| con AtomicFormula : Proposition -> Formula
| con Imply : (Formula * Formula) -> Formula
```

The type **Formula** is more substantive than the type **Proposition**. We will have need for some functions to manipulate formulas: one to extract the hypothesis and conclusion subformulas from an implication and another function to test for syntactic equality.

```
- val Hypothesis =                { Get hypothesis of implication. }
    case f of
      AtomicFormula _    . escape "Not implication" |
      Implication (f1,f2). f1;
  Hypothesis : Formula -> Formula

- val Conclusion =                { Get conclusion of implication. }
    case f of
      AtomicFormula _    . escape "Not implication" |
      Implication (f1,f2). f2;
  Conclusion : Formula -> Formula

- val rec EqFormula (f1, f2) =   { Syntactic equality of formulas. }
    f1=f2;                        { = on some types is allowed.     }
  EqFormula : (Formula * Formula) -> bool
```

We could have made **Formula** an abstract data type (possibly adding the recognizers **IsAtomic** and **IsImply**), but there is no particular reason to do so.

Finally we define the calculus of propositional logic by defining an abstract type representing proofs. An element of type theorem can be constructed only as an instance of one of the two axioms or as a result of applying *modus ponens* to theorems. Hence the ML type for theorem will have three constructors. We name the two axioms constructors K and S (from the theory of combinators), and the rule of inference *modus ponens*. In order to actually give the ML definition of the abstract data type representing proofs, a useful auxiliary function must be defined along the way. This function **ProofOf** takes an element of type theorem and returns the formula of which the theorem is a proof.

```
- abstype rec Thm =                { How an instance of ... }
    AK of Formula * Formula          |  { K axiom is represented. }
    AS of Formula * Formula * Formula |  { S axiom is represented. }
    MP of Thm * Thm                     { modus ponens is repr.   }
  with
    val AxiomK(p,q)=AK(p,q); val AxiomS(p,q,r)=AS(p,q,r);
    val rec ProofOf thm =
      case thm of
        AK(p,q).   Imply (p, Imply (q,p))                      |
        AS(p,q,r). Imply (Imply (p,q),
                     Imply (Imply(p,Imply(q,r)), Imply(p,r))) |
        MP(t1,t2). Conclusion (ProofOf t1);

    val ModusPonens (t1,t2) =
      if EqFormula (Hypothesis (ProofOf t1), ProofOf t2)
        then MP(t1,t2) else escape "Fail"
  end;
> abstype Thm
| val AxiomK : (Formula * Formula) -> Thm
| val AxiomS : (Formula * Formula * Formula) -> Thm
| val ProofOf : Thm -> Formula
| val ModusPonens : (Thm * Thm) -> Thm
```

If we did not already know what the axioms K and S, and rule of inference *modus ponens* meant, we could read it off the definition of the abstract type **Thm**. The K axiom is a proof or a one-line theorem of the form

$$\text{K:}\quad P \Rightarrow (Q \Rightarrow P)$$

for any formulas P and Q. (So technically K is an axiom *schema*.) Similarly for the S axiom; we have

$$\text{S:}\quad (P \Rightarrow Q) \Rightarrow ((P \Rightarrow (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R)).$$

The rule of inference *modus ponens* is the recursive case of the **Thm** abstract type definition. Given two theorems, **ModusPonens** constructs another theorem, but it does not do so unconditionally. We see in the definition above that it checks that the theorems it is given are proofs of formulas that stand in a certain relationship to each other. In a natural deduction style calculus we write this rule of inference:

$$\text{ModusPonens}\quad \frac{A \Rightarrow B,\quad A}{B}.$$

The first theorem must be a proof of an implication formula; the second theorem must be a proof of the hypothesis.

This completes the formalization of a fragment of propositional logic. An element of the ML data type **Formula** represents a formula in propositional logic. An element of **Thm** represents proof in the propositional calculus, because the only way such an element can be produced is to use one of the three constructors. Each constructor represents a valid step of a proof in the propositional logic. An element of type **Thm** can be created in no other way. This makes crucial use of type security of ML abstract data types. It is this type security that makes it possible to build up a system in which all elements of type **Thm** are true theorems. We call an element of **Thm** a *proof expression*.

If we had defined **Thm** as a concrete type, and not an abstract type, then the constructor **MP** would be available outside the scope of the type definition. **MP** could then be misused to form elements of type **Thm** that are not proofs, as in **MP(AK(A,A),AK(A,A))**.

An alternate implementation. We have chosen a representation for **Thm** that keeps the whole proof tree. Because only valid theorems are derived using the constructors one could throw out the proof trees [Gordon 1982]. The abstract type definition to implement this approach looks like this:

```
- abstype Thm = Theorem of Formula
  with
    val AxiomK (p,q) = Theorem (Imply (p, Imply (q,p)));
    val AxiomS (p,q,r) = Theorem (Imply (...,...));
    val ProofOf (Theorem p) = p;
    val ModusPonens (Theorem p, Theorem q) =
      if EqFormula (Hypothesis p, q)
        then Theorem (Conclusion p) else escape "Fail"
  end;
> abstype Thm
| val AxiomK : (Formula * Formula) -> Thm
| val AxiomS : (Formula * Formula * Formula) -> Thm
| val ProofOf : Thm -> Formula
| val ModusPonens : (Thm * Thm) -> Thm
```

We will not use this sort of implementation for programming logics, since we intend to write an interpreter for proof expressions of programming logics. The interpreter requires that information in the proof tree be available. As a consequence, we cannot throw the proof tree out.

An example proof. Using the implementation of propositional logic given above (either one), we can now prove that a proposition P implies itself. We first give this proof in its natural deduction style proof tree.

$$\dfrac{\dfrac{S:\cdots \qquad K:P \Rightarrow (P \Rightarrow P)}{(P \Rightarrow ((P \Rightarrow P) \Rightarrow P)) \Rightarrow (P \Rightarrow P)} \qquad K:P \Rightarrow ((P \Rightarrow P) \Rightarrow P)}{P \Rightarrow P}$$

This proof translates into the following element of the ML type `Thm`.

```
- ModusPonens (
      ModusPonens (
          AxiomS (P, Imply (P,P), P),
          AxiomK (P, P)
      ),
      AxiomK (P, Imply (P,P))
  );
  ... : Thm
```

This way of presenting a proof expression as a tree is too cumbersome—the parentheses are hard to match up and it is difficult to consider individual subexpressions. So we shall adopt a more linear presentation that makes use of the ML meta-language to define intermediate proof expressions. Here is the same example in the linear style.

```
val prex1 = AxiomK (P,P);                    {ProofOf(prex1) = P=>(P=>P)}
val prex2 = AxiomS (P, Imply(P,P), P);       {ProofOf(prex2) = many P's }
val prex3 = ModusPonens (prex2,prex1);       { [P=>((P=>P)=>P)]=>(P=>P) }
val prex4 = AxiomK (P, Imply(P,P));          { P=>((P=>P)=>P)           }
val prex5 = ModusPonens (prex3,prex4);       { ProofOf(prex5) = P=>P    }
```

It is possible to make more interesting use of the ML meta-language than defining intermediate proof expressions. For example, we could define a theorem prover to relieve the tedium of writing proofs in the propositional calculus like the one above. A theorem prover would be an ML function, call it `TheoremProver`. It would take one argument, a formula, and return a proof of that formula.

```
- val TheoremProver (phi) = ...
  val TheoremProver : Formula -> Thm
```

Properly defined, this function would have the property

`ProofOf (TheoremProver (phi)) = phi`

for true formulas `phi`. The theorem proving function acts as a kind of inverse to the `ProofOf` function. Such a function can be written in ML, since it is decidable which propositional formulas are true. Those formulas which are not theorems would cause the function to raise an exception.

3.2 Description of the programming logic AHA

What we want to describe in this section is a programming logic, not a very practical one as it is presented in abstract syntax here, but one that should appear very familiar. Its constructs are taken directly from mathematics, from predicate calculus, and from Peano arithmetic. Later we will add an interpreter that understands these constructs as constructs of a programming language.

The programming logic of this chapter is fully implemented in ML, and it automates a portion of mathematics resembling Heyting arithmetic (HA), hence its name AHA (Automated Heyting Arithmetic). As a piece of mathematics, AHA has included a number of "extras" which are worth mentioning beforehand. It has a general hierarchy of data types including all finite types as introduced by [Gödel 1958] or [Troelstra 1973]. In addition, quantification is allowed not only over variables of these types, but also over propositions and propositional functions. Except for this quantification over propositions, AHA is identical to the theory N-HA$^{\omega}$ in [Troelstra 1973].

Myth and reality. AHA is implemented in ML as the propositional calculus was in an earlier example. We will pretend in this section that there are ML data types for the classes of syntactic objects: terms, types, formulas, and proof expressions. Describing AHA will consist of describing the elements of these syntactic classes. There is no reason in principle why there could not be such an implementation with separate ML data types for the syntactic classes. In fact earlier versions of AHA implemented in ML had such an organization. The practical experience with various implementations has shown that there is a much more economical implementation. This implementation has just one ML abstract type for terms, types, formulas and proof expressions.

Because this implementation is more difficult to motivate, we defer discussing it until a type theory representation is brought up. At that time the justification comes up fortuitously on its own accord.

Because AHA is a programming logic it shares two characteristics with the ML programming language. One, all ML and AHA expressions have types. Two, ML and AHA each have an evaluation mechanism. We have already seen that the dialog between the user and ML takes on the following pattern:

```
-   ML-expression
    Eval (ML-expression) :    TypeOf (ML-expression)
```

It is easy to imagine an AHA system that has a dialog of the same pattern. Where convenient we wish to adopt the fiction that we have such an AHA system. We shall occasionally write:

```
AHA-   AHA-expression
       Eval (AHA-expression) :    TypeOf (AHA-expression)
```

Of course, in reality we have an implementation of AHA in ML which includes ML functions that implement the *Eval* and *TypeOf* functions. We might speak of this implementation as a ML-AHA hybrid system. We will see shortly that we most emphatically do not want to limit the use of ML to defining an abstract data type for AHA expressions and the functions *Eval* and *TypeOf*. We will find many uses for ML as the meta-language of AHA.

Now we turn to a presentation of AHA by example.

Terms. One constant in AHA is the number zero. It is denoted in AHA by `Zero`; entering `Zero` to an AHA system produces the response `0 : Nat`. The `Nat` after the colon indicates that zero is one of the class of natural numbers.

```
AHA- Zero;
AHA  0 : Nat
```

Notice that our representation of zero which we typed to the AHA system, namely `Zero`, is different than the representation the system prints out, namely `0`. This is somewhat arbitrary. It may seem capricious to have different representations, but in some cases there are technical reasons for doing this. In the case of zero, the numeral "0" has been reserved by ML as *its* zero constant of type integer. However, once the initial reluctance to have different representations is overcome, this becomes an advantage, since it emphasizes the difference between representation and concept as well as providing appropriate representations for the different needs of entering and reading expressions.

Actually, zero, just like any constant, is a 0-ary function symbol, There is one 1-ary function symbol provided—the successor function.

```
AHA- SuccFunc
AHA  succ : Nat->Nat
```

(One can introduce function symbols into the AHA system; the method for doing this will be shown later.) Naturally there must be a way to apply functions to arguments. This is done thus:

```
AHA-  Application (SuccFunc, Zero);
AHA   succ(0) : Nat
```

Since this soon gets to be illegible if there are many applications of the successor function, we introduce the abbreviation `Succ`.

```
AHA- Succ (Zero);
AHA  succ(0) : Nat
```

Of course, in mathematics there is an abbreviation for `Succ(Zero)` as well, namely the numeral "1". Such abbreviations can easily be obtained in AHA due to its felicitous marriage with ML. These two abbreviations are easily rendered using the function defining mechanism of ML.

```
- val Succ (t) = Application (SuccFunc, t);
  val Succ : Term -> Term
- val One = Succ (Zero);
  val One = Succ (Zero) : Term
```

The ML type of `One` is `Term`; that is, the meta-type, or the syntactic kind of `One` is `Term`. In AHA the type of `One` is `Nat`.

In use, these abbreviations produce results no different than the longer forms.

```
AHA- One;
AHA  succ(0) : Nat
AHA- Succ (One);
AHA  succ(succ(0)) : Nat
```

Besides constants, there are variables in the theory, some for every type. To get a variable one must designate its printing representation and specify the type. Variables of the same type with the same representations are the same.

```
AHA- Variable ("x", Nat);
AHA  x : Nat
AHA- Variable ("g", Arrow(Nat,Nat));
AHA  g : Nat->Nat
```

Since the type of every variable is required to be explicitly given, we see that we are building an explicitly typed system, as opposed to an implicitly typed system. The ML programming language is an implicitly typed system. The types of the variables need never be given, since their types are inferred by their use. AHA avoids the need for type inference by requiring the types be given when declaring a variable. It would, of course, be of practical interest to build a type inference mechanism for AHA to relieve the drudgery of always providing the types of variables.

Function symbols and variables are examples of basic terms. There are also ways of building new terms from old terms. We have already seen one method: application. There is, of course, abstraction as well. We demonstrate application by defining an AHA term corresponding to the identity function for natural numbers.

```
AHA- Abstract (Variable ("x", Nat), Variable ("x", Nat));
AHA  (\x: Nat. x) : Nat->Nat
```

(The AHA system uses a backslash \ for the print representation of the abstraction operator to remind one vaguely of the Greek letter lambda.) Arbitrary typed lambda-expressions can be built this way.

Predefined in the AHA system are two boolean constants **Tr** and **Fl**.

```
AHA- Tr;
AHA  Tr : Bool
AHA- Fl;
AHA  Fl : Bool
```

Like the constant **Zero**, these are AHA terms. There are other methods of forming terms besides abstraction and application: **Pair**, **First**, **Second**, **InL**, **InR** and **Out**. But they will be rarely used here.

```
AHA- Pair (Zero, One);
AHA  (0,Succ(0)) : Nat#Nat
AHA- InL (Zero, Nat);
AHA  inl(0) : Nat+Nat
AHA- InR (Bool, Zero);
AHA  inr(0) : Bool+Nat
```

We could continue to write out AHA terms completely. But as seen from the simple lambda-expression above, this will become unreadable. Constructing a complete proof in this way would be awkward, to say the least. A more perspicuous method would be possible if we had a way to refer to terms we have previously created. This is easily done in the ML meta-language, so we adopt this method as illustrated in the lambda term below.

```
- val var = Variable ("x", Nat);
  val var = - : Term

AHA- var;
AHA  x : Nat

- val term = Abstract (var, var);
  val term = - : Term

AHA- term;
AHA  (\x. x) :  Nat->Nat

AHA- Application (term, Zero);
AHA  0: Nat
```

Since we are forced to be more explicit than is usual in mathematical logic, a little mental gymnastics is required to switch back and forth between the meta-language and the AHA system.

At this time it is worth mentioning how we get a kind of parametric polymorphism for free. This is done by using defining ML functions that take types as arguments and create AHA terms.

```
- val Identity (ty) =
    let val var = Variable ("x", ty) in
      Abstract (var, var);
  val Identity : Type -> Term

AHA- Identity (Nat);
     (\x:Nat. x) : Nat->Nat

AHA- Application (Identity (Bool), Tr);
     Tr : Bool
```

Types. Now we will say a few words about types. We have already encountered **Nat**—the type of the natural numbers, and **Bool**—the type of boolean values. These are examples of primitive types. Other primitive types could be introduced in AHA, but none will be considered here. Using the natural numbers we can build types for function spaces, cross products and unions with the type operators **Arrow**, **Cross** and **Union**. We have already seen how to build terms in function spaces. It is also possible to make variables with any one of these types.

```
AHA- Variable ("F", Arrow (Arrow (Nat,Nat), Nat));
AHA  F : (Nat->Nat)->Nat
AHA- Variable ("X", Cross (Arrow (Nat,Nat), Nat));
AHA  X : (Nat->Nat)#Nat
AHA- Variable ("U", Union (Arrow (Nat,Nat), Cross (Nat,Nat)));
AHA  U : (Nat->Nat)+(Nat#Nat)
```

The hierarchy of types closed under **Arrow**, **Cross** and **Union** is called the *simple type hierarchy*.

One can have variables that range over propositions in AHA. These variables have type **Prop**. Propositional function variables of type **Arrow(Nat,Prop)** are allowed as well. In fact, variables and function symbols may range over another type hierarchy. This type hierarchy, the *propositional type hierarchy*, is simply the closure under the three type operators of the primitive types *and* **Prop**. These are exactly the types over which quantification is permitted.

In this thesis little use will be made of the rich propositional type hierarchy just introduced. But it is worthwhile to take a break in the development of AHA and connect the propositional type hierarchy to its origins.

Such a type theory originated with Bertrand Russell. We have chosen to have a complex and flexible hierarchy of data types, the data type hierarchy. Let us suppose for a minute that we had chosen just to have **Nat**, a special case of the arrow operator **Arrow**(T,**Prop**), and the cross product operator. Now we can compare the propositional type hierarchy with that of Russell [Hatcher 1982]. Russell began with one domain of discourse D. We will equate it to **Nat**.

type symbol	*order*	*AHA type*	*mathematical object*
o	0	`Nat`	basic domain
(o)	1	`Arrow(Nat,Prop)`	sets
(o,o)	1	`Cross(Nat,Nat)`	ordered pairs
$((o,o))$	2	`Arrow(Cross(Nat,Nat),Prop)`	binary relations
$((o))$	2	`Arrow(Arrow(Nat,Prop),Prop)`	sets of sets

The order of a type is the greatest depth of parentheses in the type symbol and the order of a term is the order of its type.

We will never have need to go very high in this type hierarchy. Nevertheless the same rules of logic that apply to first order logic will be adopted for the AHA calculus with this expressive type hierarchy.

Variables of any type are permitted in AHA. This permits impredicative terms; impredicative terms are those with rank less than the maximum rank of any variable occurring in them. Here is an example of a predicative and an impredicative term.

```
val x = Variable ("x", Nat);
val P = Variable ("P", Arrow (Nat, Prop));
val impred = Abstract (x, Application (P, x));
val pred = Abstract (P, Application (P, x));
```

Formulas. We resume the presentation of AHA with the formulas in the logic. Now we must add the familiar logical apparatus to our theory. We start with the two formulas representing true and false.

```
AHA- True;
AHA  True : Prop
AHA- False;
AHA  False : Prop
```

The type **Prop** is not to be confused with the type **Bool**. The type **Bool** is the definite data type with two particular fixed elements. It is always the case that an element of type **Bool** is equivalent to one of these two canonical terms. The type **Prop** is something else entirely. It is related to propositions and formulas. We do not want to conclude that all elements of type **Prop** are equal (in some sense) to the element **True** or to **False**. This would commit us to the law of excluded middle.

The other atomic formulas are equations (made with the function **Equal**), propositional symbols and propositional functions.

```
- val x = Variable ("x", Nat);
  val x = - : Term

AHA- x;
AHA  x : Nat

AHA- Equal (x, Zero);
AHA  x=0 : Prop

- val P = PredicateSymbol ("P");
  val P = - : Term

AHA- P;
AHA  P : Prop

- val H = PropFunction ("H", Nat);
  val H = - : Term

AHA- H;
AHA  H : Nat->Prop

AHA- Application (H, x);
AHA  H(x) : Prop
```

The AHA expressions above of type **Prop** are all atomic formulas.

At this point it is convenient to demonstrate how to make additional function symbols. These function symbols are uninterpreted; that is, the AHA system knows nothing about them (as it does about the successor function). The difference between interpreted and uninterpreted function symbols will become clear after the discussion on evaluation.

```
- val f = FunctionSymbol ("f", Arrow(Nat,Nat));
  f: Nat->Nat
```

The symbol **f** so introduced is an uninterpreted function symbol. The second argument to **FunctionSymbol**—the type of the symbol introduced—can be any simple type, **Prop**, or any other propositional type. This permits the functions that introduce predicate symbols and propositional functions to be just special cases of the **FunctionSymbol** function.

```
- val PredicateSymbol (s) = FunctionSymbol (s, Prop);
  val PredicateSymbol : string -> Term
- val PropFunction (s,t) = FunctionSymbol (s, Arrow (t,Prop));
  val PropFunction : string * Type -> Term
```

To summarize: atomic formulas are made from terms with type **Prop** and from terms of the same type using **Equal**.

We have already remarked that the ML abstract types **Term** and **Formula** are an expository fiction. If these were really two separate types, a minor technical problem would arise here. A *term* of type **Prop** would not be an atomic *formula* without the aid of an ML constructor making a formula out of a term, thus converting it from one abstract type to the other. The function **MakeProp** served a similar purpose in the context of the propositional calculus. One might wish that the ML type structure would accept that terms of type **Prop** *are* formulas. However, the type inferencing mechanism of ML could not be extended to handle with sort of type definition.

To relate formulas there are the usual connectives: **Not**, **And**, **Or** and **Imply**. We give some examples.

```
- val Q = Not (Equal (Application (f,x), Zero));
  val Q = - : Formula
```

```
AHA- Q;
AHA  ~f(x)=0 : Prop

- val R = And (Not P, Application (F,x));  { Uses P,F above. }
  val R = - : Formula

AHA- R;
AHA  ~P&F(x)

AHA- Or (Q,R);
AHA  (~f(x)=0 | (~P&F(x)))

AHA- Imply (Q,R);
AHA  (~f(x)=0 => (~P&F(x)))
```

There is no need to introduce **Not** as a primitive connective, since it is easy enough to introduce it as an abbreviation. The unary connective for logical negation **Not**(x) will stand for **Imply**(x,**False**). Likewise the biconditional is defined as usual: **Iff**(x,y) is short for **And**(**Imply**(x,y),**Imply**(y,x)). Abbreviations are easy to implement since they correspond directly to ML functions. For example, the previous two abbreviations correspond to the following ML functions.

```
- val Not (x: Formula): Formula = Imply (x, False);
  val Not : Formula -> Formula
- val Iff (x: Formula, y: Formula): Formula =
    And (Imply (x,y), Imply (y,x))
  val Iff : Formula * Formula -> Formula
```

Finally, come the quantifiers. **Some** and **All** are two new ways of building formulas out of other formulas and variables.

```
AHA- Some (x, Application (F,x));
     Ex:Nat. F(x)
AHA- All (x, R);
     Ax:Nat. (~P&F(x))
```

Proof expressions. Now that the formulas have been described we want to define a calculus with which we can derive true facts about arithmetic. In its overall makeup this calculus is a natural-deduction style calculus [Anderson and Johnstone 1962, Prawitz 1965, Dummett 1977] with introduction and elimination rules. We shall call these proofs, proof expressions of formulas because we will describe how to make them just as we have already described how to make terms of types.

Of all the formulas introduced above, **True** is the one we most certainly would like to derive. It has, as one would hope, a trivial derivation consisting of simply the proof expression **Truth**.

```
AHA- Truth;
AHA  Truth : True
```

Certain equations are most assuredly true by accepted laws of equality. These rules, reflexivity, symmetry, congruence, etc., will be introduced later. We examine now the introduction and elimination rules for **False**. Written out in standard natural deduction style, they look like this:

$$\frac{A \& \neg A}{False} \qquad \frac{False}{A}$$

These rules correspond to two constructors, **FalseIntro** and **FalseELim**, in the AHA representation of the logic. **FalseIntro** takes an argument which must be a proof of a contradiction and the result is a proof expression proving false. (Actually this rule is not needed at all. Given that $\neg A$ is an abbreviation for $A \Rightarrow False$, this rule is a special case of **ImpElim** which will be discussed shortly.) Surprisingly **FalseElim** needs two arguments. Besides the proof of false, **FalseElim** needs the formula A as an argument to indicate what the proof expression proves. Thus **FalseElim** $(prex:$ **False**$, A)$ is a proof expression proving A (where by $prex:$ **False** we mean $prex$ is a proof expression in AHA proving $False$). Hence, arguments to do correspond one-to-one with premises. Arguments other than premises are there to provide additional information about the particular form the rule of inference is to take.

The **AndIntro**, **AndElimR** and **AndElimL** rules may be written:

$$\frac{A, \quad B}{A \& B} \qquad \frac{A \& B}{A} \qquad \frac{A \& B}{B}$$

These natural deduction rules must be understood as definitions of constructors in AHA which will fail if the arguments are not in the form that the rule prescribes. **AndIntro** is a constructor that takes two arguments, proof expressions, and forms a proof expression of the conjunction of the arguments. **AndElimR** is a constructor that takes one argument, a proof expression proving a conjunction, and is a proof expression of the left conjunct. The constructor **AndElimL** is similar. For disjunction there are two introduction rules and one elimination rule. The introduction rules **OrIntroR** and **OrIntroL** look like:

$$\frac{A}{A \mid B} \qquad \frac{A}{B \mid A}$$

where B is a formula and not a proof expression. The constructor **OrIntroR** has two arguments: a proof expression which proves A and a formula B. Together these supply all the information necessary to form a proof expression of the disjunction. The **OrElim** is slightly more complicated and takes three arguments. The first must be a proof expression of a disjunction; the other two arguments must be proofs of implications with special forms:

$$\frac{A \mid B, \quad A \Rightarrow C, \quad B \Rightarrow C}{C}$$

The proof of an implication in a natural deduction style calculus requires assuming A, then proving B under that assumption. The result is $A \Rightarrow B$. The rule is usually written something like:

$$\frac{\begin{array}{c} A \\ \vdots \\ B \end{array}}{A \Rightarrow B}$$

In AHA we accomplish the proof of an implication using the **Assume** construct and the **ImpIntro** rule as follows:

```
- val hyp = Assume ("hyp", A);          { Assume A. }
      ⋮
- val prex1 = ... hyp ... ;             { Derive proof of, say, B. }
- val prex2 = ImpIntro (hyp, prex1); { Proof of A ⇒  B. }
```

The string **"hyp"** is the name of the assumption. This is to distinguish it from other assumptions of the same formula. Do not confuse the ML (meta-language) variable **hyp** with its contents, the AHA proof expression **Assume ("hyp", A)**, even though we often call AHA expressions by the ML identifiers to which they are bound. The **ImpIntro** construct is said to discharge the assumption **hyp**.

Finally, there is the corresponding elimination rule **ImpElim** which is just the familiar rule of *modus ponens*:

$$\frac{A \Rightarrow B, \quad A}{B}$$

With the rules governing the basic connectives out of the way, we have rules primarily for the quantifiers and arithmetic left over. We list here the rules that are most self-evident in the more compact "turnstile" form.

Truth	$\vdash True$
PeanoPostulate7	$n' = m' \vdash n = m$
PeanoPostulate8	$\vdash \neg\, n' = 0$
BetaReduction	$\vdash (\lambda x.b)(t) = [x/t]b$
AllIntro	$P(x) \vdash All\ x\,.\,P(x)$
AllElim	$All\ x\,.\,P(x) \vdash P(t)$

Two rules originate from Peano's axiomization of arithmetic, axioms seven and eight in Peano's original list. His remaining axioms are either logical in nature or can be derived in AHA.

Although the **AllIntro** rule is straightforward, we pause to examine it in order to introduce the notion of a binding. The **AllIntro** rule looks like this:

$$\textbf{AllIntro}\ (x,\ prex : P(x))$$

The first argument to **AllIntro** is a variable x. This is another example of a kind of argument different from a premise. This is a particular kind of argument, and it stands in a special relation to the second argument. All free occurrences of the variable x in *prex* are bound by the **AllIntro** construct. More precisely, we view the pair x, *prex* as a *binding*. This has special meaning for substitution. Substituting t for x in this binding has no effect, because no free occurrence of x is possible. Substituting t for y (different from x) works as expected except when x occurs free in t. In this case, if y occurs *prex*, an error message is raised.

We have come across a binding before in the construct **Abstract**(x,b), and we shall shortly come across some more. In general a construct can have more than one binding as arguments, and each binding can bind more than one variable in an expression. (See Appendix A for the implementation details of the logic engine.)

We conclude this discussion on proof expressions in AHA by describing the three most complicated ones: some-introduction, some-elimination, and induction. Each has many arguments.

The natural deduction style rule for some-introduction looks like this:

$$\frac{P(t)}{Some\ x\,.\,P(x)}$$

The **SomeIntro** proof expression constructor requires three arguments. The first argument is a "some" formula. This provides the formula to be proved, since determining it from $P(t)$ is not trivial and not unique. Also the formula indicates the name of the bound variable which may be convenient for renaming variables. The second argument is a proof expression. It must prove the scope of the "some" formula P for a particular term. The third argument must be the particular term t for which one showed in the second argument that $P(t)$. Put together, the use of the **SomeIntro** constructor looks like this:

$$\texttt{SomeIntro (Some(}x\texttt{,}P\texttt{),}\ prex : P(t)\texttt{,}\ t\texttt{)}$$

The some-elimination rule looks like

$$\frac{\mathit{Some}\ x \,.\, P(x), \quad \begin{array}{c} P(x_0) \\ \vdots \\ Q \end{array}}{Q}$$

The constructor **SomeElim** has four arguments: the proof expression of some existentially quantified formula, the variable used to refer to the term postulated, an assumption, and a proof of Q from the assumption.

$$\texttt{SomeElim (}prex_1 : \texttt{Some(}x\texttt{,}P\texttt{),}\ x_0\texttt{,}\ shyp : P(x_0)\texttt{,}\ prex_2 : Q\texttt{)}$$

The variable x_0 is bound in the proof expression $prex_2$. The assumption $shyp$ in $prex_2$ is discharged.

Induction looks like:

$$\frac{P(0), \quad \begin{array}{c} P(n) \\ \vdots \\ P(n') \end{array}}{\mathit{All}\ x \,.\, P(x)}$$

The constructor **Induction** has five arguments: the "all" formula to prove, the base case $P(0)$, the induction variable, the induction hypothesis $P(n)$, and a proof of $P(n')$ from the induction hypothesis.

$$\texttt{Induction (All(}m\texttt{,}P\texttt{),}\ prex_1 : P(0)\texttt{,}\ n\texttt{,}\ ihyp : P(n)\texttt{,}\ prex_2 : P(n')\texttt{)}$$

The variable n is bound in the proof expression $prex_2$, and the assumption $ihyp$ in $prex_2$ is discharged.

Restrictions. As one would expect from experience with first-order predicate logic, there are some restrictions governing the use of the rules for manipulating quantifiers and induction as well. We will first look at all-elimination. From what we have said so far, one might be tempted to prove $\mathit{Some}\ y \,.\, D(y, y)$ for a binary predicate D from the assertion $\mathit{All}\ x \,.\, \mathit{Some}\ y \,.\, D(x, y)$ using all-elimination. We hope, of course, that this is not permitted and, in fact, the deduction is not possible. The reason is that the exact formulation of the all-elimination rule requires that the proof expression `AllElim (`$prex$`: All(`x`,`$P(x)$`), `t`)` is a proof of the formula $[x/t]P(x)$, where $[x/t]P(x)$ is the formula $P(x)$ with all free occurrences of x replaced by t. The precise definition of substitution prevents the capture of free variables. Hence, $[x/y]Some\ y\ .\ D(x,y)$ yields $Some\ y\ .\ D(x,y)$—that is, no change at all—and not $Some\ y\ .\ D(y,y)$. A similar problem for the some-introduction rule is avoided in the same way.

A slightly bigger problem appears in the case of all-introduction. The solution is to restrict the choice of variables to those that are not free in any undischarged assumptions. In the some-elimination rule

`SomeElim (`$prex_1$` : Some(`x`,`P`), `x_0`, `$shyp : P(x_0)$`, `$prex_2 : Q$`)`

the variable x_0 cannot be free in `Some(`x`,`P`)` or Q. In the induction rule, the induction variable n cannot occur free in `All(`m`,`P`)`.

Finally, the rules concerning equality should be mentioned. They are the typical rules one would expect.

`Reflexivity`	$\vdash n = n$
`Symmetry`	$n = m \vdash m = n$
`Transitivity`	$n = m, m = p \vdash n = p$
`Congruence`	$f = g, x = y \vdash f(x) = g(y)$
`Substitution`	$x = y, P(x) \vdash P(y)$
`EqualityElim`	$P = Q, Q \vdash P$

Example of subtraction. Now that we have the syntax of a sample programming logic, it is time to give an example of a proof. We give a proof of the following theorem in AHA:

$$All\ x\ .\ (x = 0\ \mid\ Some\ y\ .\ y' = x)$$

It has a very simple proof by induction which in AHA is built up as follows:

```
- val x,y = Variable ("x", Nat), Variable ("y", Nat);
- val SOME (term) = Some (y, Equal (Succ y, term));
- val EQO (term) = Equal (term, Zero);
- val ALL = All (x, Or (EQO (x), SOME (x)));
```

This first line declares two variables for use in the proof. The remaining lines set abbreviations which are used to make formulas that will come up in the proof. The theorem to be proved is expressed in AHA as the last of these formulas, ALL. The base case of the induction requires proving

$$0 = 0 \mid Some\, y \,.\, y' = 0$$

or expressing in the notation of AHA, `Or (EQO (Zero, SOME (Zero))`. This is proved easily by reflexivity and or-introduction.

```
- val base = OrIntroL (Reflexivity Zero, SOME (Zero));
```

The induction step requires slightly more work. Notice that the induction hypothesis is not required in any material way in the proof of

```
Or(EQO(Succ x),SOME(Succ Zero))
```

```
- val indhyp = Assume ("indhyp", ALL);
- val prex1 = Reflexivity (Succ x);
- val prex2 = SomeIntro (SOME (Succ x)), prex1, x);
- val prex3 = OrIntroR (EQO (Succ x), prex2);
- val proof = Induction (ALL, base, x, indhyp, prex3);
```

In the last line, the base case and the induction step are combined to yield a proof expression corresponding to the desired proof.

Definitions. The use of ML to define or abbreviate AHA terms, types, formulas, and proof expressions suffers from one practical drawback. Every time the AHA expression is used it is type checked again. If the AHA expression appears more than once in another expression, it can be evaluated more than once. For large expressions saving this extra work would be a real boon. What is needed is a definition facility which works inside the AHA system, one where the definiendum is an AHA expression whose type and value can be determined in constant time. Such a definition facility is provided. The AHA expression `Definition(`*name*`,` *exp*`)` evaluates *exp* once and saves its type and its value. For example, suppose a lemma is used several times in the course of the development of some theory. To avoid unnecessary calculations it can be saved as a definition.

```
val Lemma = Definition ("Lemma", prex);
...
val Theorem1 = ... Lemma ...;
...
val Theorem2 = ... Lemma ...;
```

3.3 Cantor's theorem

In the previous section we described a formal system, AHA, which can be used to prove theorems in number theory. In this section we make use of AHA to prove Cantor's theorem. We present this proof as an example proof of a non-trivial theorem. This proof also exercises the higher-order features of AHA.

Cantor originally set out to prove that the set of real numbers is countable. He succeeded in proving that the set of all algebraic numbers is countable. In the paper [Cantor 1874] in which he published this result, he also shows that the set of real numbers cannot be countable. The essence of the proof is now a familiar technique known as diagonalization. The proof is so elegant that the truth of the theorem is irrefragable. Since we do not have a data type corresponding to real numbers, we recast the theorem in terms of functions from natural numbers to natural numbers, or sequences. The theorem then states that there is no enumeration of sequences such that every sequence appears at least once in the enumeration. An enumeration of sequences is a function E with domain `Nat` and range `Arrow (Nat,Nat)`. Here is the statement of the theorem:

$$\neg\, Some\ E\ .\ All\ f\ .\ Some\ n\ .\ E(n) = f$$

First we prove a trivial lemma: $All\ n\ .\ \neg\, n' = n$. We start by declaring a variable and making an abbreviation for the statement of the lemma.

```
val n = Variable ("n", Nat);
val lemma = All (n, Not (Equal (Succ n, n)));
```

The proof is by induction. The base case $0' \neq 0$ is just exactly an instance of Peano Postulate 8.

```
val base = PeanoPostulate8 (Zero);
```

The induction step uses Peano Postulate 7. The point is that $n' \neq n$ implies $n'' \neq n'$.

```
val hyp  = Assume ("hyp", Not (Equal (Succ n, n)));
val raa  = Assume ("raa", Equal (Succ (Succ n), Succ n));
val prex1 = FalseIntro (AndIntro (PeanoPostulate7 raa, hyp));
val prex2 = NotIntro (raa, prex1);
```

Induction combines the base case and the induction step to complete the proof. The proof is then bundled up altogether in a neat package by the definition command.

```
val proof = Induction (lemma, base, n, hyp, prex2);
val Lemma = Definition ("Lemma", proof);
```

Now for the theorem. First the variable declarations.

```
val E = Variable ("E", Arrow (Nat,Arrow(Nat,Nat)));
val f = Variable ("f", Arrow (Nat,Nat));
val n = Variable ("n", Nat);
```

Now for some abbreviations of formulas which are used in the proof.

```
val Eq = Equal (Application(E,n),f);
val A = All (f, Some (n, Eq));
val C = Some (E, A);          { Some E. All f. Some n. E(n)=f }
```

In the proof of the theorem we are going to exhibit a sequence that does not appear in the enumeration E_0. We call it **Diag** and define it here: $\lambda x.E_0(x)(x)'$.

```
val x = Variable ("x", Nat);
val E0 = Variable ("E0", Arrow (Nat, Arrow (Nat, Nat)));
val Diag = Abstract (x, Succ (Application (Application (E0,x),x)));
```

The function **Diag** is function from **Nat** to **Nat**. For any given natural number x, the function **Diag** picks the x-th sequence in the given enumeration E, chooses the x-th element of that sequence, and returns the successor of that element. Notice that **E0** is a free or non-local variable in **Diag**. This is looking ahead to the formation of proof expression **prex6** where **E0** will be bound.

Now the proof starts. Assume for purposes of contradiction that an enumeration of sequences exists.

```
val raa = Assume ("raa", C);
```

Therefore we can assume that all sequences appear in the enumeration.

```
val shyp1 = Assume ("shyp1", A);
```

Then the sequence **Diag** appears in the enumeration somewhere, say at index **d**.

```
val d = Variable ("d", Nat);
val shyp2 = Assume ("shyp2", Equal (Application (E0), Diag));
```

Now we show that $E_0(d)(d) = E_0(d)(d)'$, since $Diag = E_0(d)$ by choice of d, and since $Diag(d) = E_0(d)(d)'$ by the way we constructed *Diag*.

```
val prex1 = Congruence (shyp2, Reflexivity d);   { E0(d)(d)=Diag(d) }
val prex2 =
  Transitivity (prex1, BetaReduction (Application (Diag, d)));
```

Next we use the lemma to show that $E_0(d)(d)' \neq E_0(d)(d)$.

```
val prex3 = AllElim (Lemma, Application (Application (E0,d),d));
```

This contradicts what we have shown above.

```
val prex4 = FalseIntro (AndIntro (Symmetry prex2, prex3));
val prex5 = SomeElim (AllElim (shyp1, Diag), d, shyp2, prex4);
```

In the next step we choose variable E0 to be the enumeration assumed to exist by assumption **raa**. Often one chooses the same variable as in the statement of the theorem to avoid declaring a new variable. Thus the distinction between E and E_0 would merge. We have been pedantic here to make sure the distinction was not overlooked.

```
val prex6 = SomeElim (raa, E0, shyp1, prex5);
```

The proof expression **prex6** is a proof of **False** under the assumption **raa**. This assumption must, therefore, be incorrect.

```
val proof = NotIntro (raa, prex6);
```

This proof is not, as some contend [Bell 1937, page 569], a nonconstructive proof. It has constructive meaning. which can be put to good use. Given any proposed enumeration E of sequences, a counter-example $\lambda x.E(x)(x)'$ is constructed.

This observation leads to the following result. If one can prove

$$\neg\ Some\ x\ .\ All\ y\ .\ P(x,y)$$

in AHA, then one can also prove

$$All\ x\ .\ Some\ y\ .\ \neg\ P(x,y)$$

under certain circumstances. If we have, as in the proof of Cantor's theorem, a proof expression in the form

```
NotIntro (raa, SomeElim (raa, x, shyp, prex))
```

where **prex** is a proof of false using **AllElim (shyp,** t**)**, then we can prove

$$All\ x\ .\ Some\ y\ .\ \neg\ P(x,y)$$

as follows:

```
val S = Some (y, ¬ P(x,y));
val raa' = Assume ("raa'", P(x,t));
val prex' = [raa'/AllElim(shyp,t)]prex;
val proof = AllIntro (x, SomeIntro (S, NotIntro (raa', prex'), t));
```

Now any proposed x_0 such that $All\, y \,.\, P(x_0, y)$ can be debunked by a counter-example y_0 such that $\neg\, P(x_0, y_0)$.

The implications of finding a counter-example like this are worth considering for a moment. Suppose an expert system asserts that there is no such x such that $All\, y \,.\, P(x, y)$. Despite the indubitable reasoning that led to that conclusion, there may persist lingering suspicions in the mind of the user that x_0, say, is really such an x. With a constructive interpretation of proofs, the expert system can be called on to explain why x_0 is not an appropriate x by producing a counter-example.

3.4 Recursion

In order to make AHA practical, we need to consider a number of constructs which we can add to AHA. From the realm of mathematical experience we know that definition by recursion is an easy way of introducing new functions. So in this section we add a recursion construct to AHA that is sufficiently general enough to capture the usual recursion schemes. This construct plus the higher-order nature of AHA makes AHA a powerful programming language.

Definition by recursion. Definition by recursion usually takes the following form:

$$f(0) = t, \quad f(n') = s,$$

where the term s is permitted to mention $f(n)$ and n. The variable n is the induction variable and $f(n)$ is the value that the recursive call to the function f yields. It will prove useful to replace any occurrence of $f(n)$ by a fresh variable, say, v, with the understanding that it represents the value of the function f at the argument n. Now given the terms t and s, and the variables n and v, we have all the information we need to compute the function f. Thus the function f is defined. In AHA the syntax is:

$$\texttt{Recursion}\ (t,\ n,\ v,\ s)$$

A typical definition in mathematics, like that of the "cut-off" subtraction function, looks like this:

$$f(0) = 0, \quad f(n') = n.$$

In AHA the function f is represented by the term `Recursion(Zero,n,v,n)` where n and v are both variables of the type `Nat`. In this case v represents the value of the function $f(n)$ and is not used at all in the term making up the induction step (this term is simply n, the induction variable, in the example).

Another common example in mathematics is the addition function. Since the addition function is binary, this example brings up the issue of parameters in recursive definitions. Typically one argument is picked to do induction on and the other argument is a parameter. The definition of recursion given for AHA makes no provision for parameters. This might at first appear to be a serious deficiency. On the other hand, there is no stipulation that the terms t and s must have a base type—in particular, they can be functions. In this way parameters can be circumvented. Here is the definition of the addition function:

$$f(0) = (\lambda x.x), \quad f(n') = (\lambda x.f(n)(x)').$$

In AHA this becomes:

```
- val n,x = Variable ("n", Nat), Variable ("x", Nat);
> val n = - : Term
| val x = - : Term

- val h = Variable ("h", Arrow (Nat, Nat));
  val h = - : Term

AHA- Recursion (Abstract(x,x), n, h,
                Abstract(x, Succ (Application (h, x))))
     Recursion [(\x.x), (\x.succ(h(x)))] : Nat -> Nat
```

The base case is the "plus 0" function which is just the identity function. In the inductive step the variable h is the "plus n" function, from which the "plus n'" function is easily constructed.

We can assign this term to an ML variable and use it like the curried addition function.

```
- val PlusFunc =
    Recursion (Abstract (x,x), n, h,
               Abstract (x, Succ (Application (h,x))));
  PlusFunc = - : Term

- val Plus (n,m) = Application (Application (PlusFunc, n), m);
  val Plus = fun : Term * Term -> Term
```

The ML function `Plus` is an abbreviation, like `Succ`, which saves writing terms with many application operations in them. What then is meant by `Plus(One,One)`? Is it really the same as `Two`? The answer is no. We will see later in the section on evaluation that **Plus(One,One)** reduces or evaluates to **Two**. But **Plus(One,One)** is just an abbreviation for the following expression:

```
Application (
  Application (
    Recursion (
      Abstract (x,x), n, h,
      Abstract (x, Application (SuccFunc, (Application(h,x))))),
    Application (SuccFunc, Zero)),
  Application (SuccFunc, Zero))
```

The method used above for defining a function with a parameter by induction is completely general. We could add the following recursive scheme to AHA:

$$f(0,x) = t, \quad f(n',x) = s$$

where the term s may have occurrences of the variable n and $f(n,r)$ for any term r, as well as the parameter x. The example of the addition function indicates how this scheme can be reduced to the **Recursion** construct by defining a curried version of the function. The curried version looks like:

$$F(0) = (\lambda x.t), \quad F(n') = (\lambda x.S)$$

where the term S is obtained from s by replacing occurrences of $f(n,r)$ by $F(n)(r)$. The final step is to uncurry the function F: $f = \lambda p.F(\mathtt{First}(p))(\mathtt{Second}(p))$. Now we sketch this development in ML.

```
val n, x = Variable ("n", Nat), Variable ("x", A);
val h = Variable ("h", Arrow (A, B));
val F = Recursion (Abstract (x,t), n, h, Abstract (x, S));
val p = Variable ("p", Cross (Nat, A));
val f = Abstract (p,
        Application (Application (F, First p), Second p));

AHA- f;
     (\p. F(First p)(Second p)) : (Nat, A) -> B
```

Mutual recursion poses no special problem either. The following scheme could easily be added to AHA and reduced to the **Recursion** construct:

$$f(0) = t_1, \quad f(n') = s_1, \quad g(0) = t_2, \quad g(n') = s_2$$

where the s_i may have occurrences of $f(n)$, $g(n)$ and n. This can be accomplished by defining a function h that contains both f and g:

$$h(0) = \mathtt{Pair}(t_1, t_2), \quad h(n') = \mathtt{Pair}(S_1, S_2)$$

where in the S_i occurrences of $f(n)$ and $g(n)$ are replaced by $\mathtt{First}(h(n))$ and $\mathtt{Second}(h(n))$.

Ackermann's function. To demonstrate the power of the AHA system we give an example of a function which is not primitive recursive. We do not deviate from using the tired example of Ackermann's function [Ackermann 1928, McNaughton 1982], which is known to be not primitive recursive. Although definition by recursion does not appear to be a powerful means of recursion, it is not restricted to defining functions into the natural numbers. By defining functions of functions, very fast-growing functions can be defined.

Although somewhat overused, this example should not be taken too lightly. For by virtue of being expressible in AHA, we know that Ackermann's function terminates. A proof of termination in, say, PPLAMBDA, is not easy.

The definition of Ackermann's function was inspired by considering the following sequence of binary functions:

$$x + y, \quad x \times y, \quad x^y, \quad \left.x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\right\}y, \ \ldots .$$

Each function is obtained by iterating the preceding function. We denote the i-th iterate of a function f of one argument by f^i, and define it recursively by $f^0(x) = x$ and $f^{i+1}(x) = f(f^i(x))$. In AHA the iteration operator can be defined by recursion as follows.

```
val x,n = Variable ("x", Nat), Variable ("n", Nat);
val F = Variable ("F", Arrow (Nat, Nat));
val base = Abstract (F, Abstract (x, x));
val Type = TypeOf base;            { (Nat => Nat) => (Nat => Nat) }
val G = Variable ("G", Type);
val ind = Abstract (F, Abstract (x, Application (F,a)))
                where val a = Application (Application (G,F),x) end;
val IterFunc = Recursion (base, n, G, ind);
val Iter (n,f,x) =
  Application (Application (Application (IterFunc, n), f), x);
```

Thus `Iter(n,f,x)` represents the function f iterated n times at the argument x.

In preparation for defining Ackermann's function we need some additional functions. We need the "cut-off" subtraction function.

```
val MonusFunc = Recursion (Zero, n, x, n);
val Monus x = Application (MonusFunc, x);
```

We also need an AHA term to act like an if-statement. It must take three AHA terms X, Y and Z. It must evaluate to Y, if X evaluates to 0; otherwise it must evaluate to Z. We use the ML meta-language to define a function that takes three AHA terms as arguments and returns an AHA term which exhibits the necessary behavior when evaluated.

```
val IfZero (X,Y,Z) =
  Application (Recursion (Y, n, x, Z), X)
    where val Type = TypeOf Y;          { Must be same as TypeOf Z.}
      val x = Variable ("$x", Type);    { Fresh variables; at least}
      val n = Variable ("$n", Nat) end; { not free in Z.           }
```

Now we can proceed to the actual definition of Ackermann's function. Here is the definition:

$$Ack(0, x, y) = y + x,$$

$$Ack(1, x, y) = y * x,$$

$$Ack(2, x, y) = y^x,$$

$$\dots$$

$$Ack(z', x', y) = \text{Iterate } \lambda\, w \,.\, Ack(z, w, y) \ x \text{ times on } y.$$

We will define the curried version of this function in AHA by recursion on the first argument. Thus the function will take a natural number as an argument and return a binary function. First we define the base case, which is just the plus function.

```
val base = PlusFunc;
```

We save the type of the base case for use later, and define a variable H to use in the induction step. H will be the value (a binary function) that Ackermann's function returns on the preceding case.

```
val Type = TypeOf base;                    { Nat => (Nat => Nat) }
val H = Variable ("H", Type);
```

Now we define the induction step:

$$ind(n')(x,y) = \begin{cases} 0, & \text{if } x = 0 \text{ and } n = 0; \\ 1, & \text{if } x = 0 \text{ and } n > 0; \\ (\lambda w.\, ind(n)(w,y))^{x-1}(y), & \text{otherwise.} \end{cases}$$

In AHA this is accomplished as follows.

```
val x,y = Variable ("x", Nat), Variable ("y", Nat);
val w,n = Variable ("w", Nat), Variable ("n", Nat);

val ind = Abstract (x, Abstract (y, IfZero (x, t, s)))
  where val t = IfZero (n, Zero, One);
        val g = Abstract (w, Application (Application (H,w),y));
        val s = Iter (Monus x, g, y) end;
val AckFunc = Recursion (base, n, H, ind);
val Ack (z,x,y) =
  Application (Application (Application (AckFunc, z), x), y);
```

It is perhaps interesting to compare this development of Ackermann's function with that done in Constructive Set Theory (this language is closely related to TT which will be discussed later) [Nordström 1981].

3.5 Computation

Thus far nothing in the description of the AHA programming logic has appeared that would distinguish the implementation of AHA from an implementation of a classical logic. We have described a theory with its language and rules of inference. So we know how to make proofs in the system. This was sufficient for the LCF system. A proof that the least fixpoint of some equation had some property was the goal desired. The rest of the effort in the LCF system could be devoted to assisting the creation of these proofs. In a programming logic, on the other hand, the goal will often require proving a different kind of theorem. Typically one will want to prove a theorem of the form "for all x–there exists y". Then by the nature of a programming logic, an interpreter can be written that when supplied with that proof and a particular x will produce a y with the desired property.

Thus to program the integer division function, one would just prove a theorem in a programming logic:

$$All\ x, y \,.\, 0 < y \Rightarrow Some\ q, r \,.\, x = y * q + r \,\&\, r < y$$

The interpreter for AHA is the subject of this section. In order to write this interpreter some care is needed in choosing and formulating the rules of the programming logic. Fortunately, we can draw on the experience of the constructive school of mathematics for help in devising these rules. Their criticisms of classical mathematics provide insight to the problem.

Constructive mathematics. There is a standard nonconstructive proof which is often brought up to illuminate the motives of the constructivists. Here is the statement of the theorem: there are two irrational numbers a and b such that a raised to the power of b is rational. The proof goes as follows. Consider $\sqrt{2}$. We know that $\sqrt{2}$ is irrational. Now let $x = \sqrt{2}^{\sqrt{2}}$. Is x rational? If it is, then the theorem is proved. Otherwise, consider $x^{\sqrt{2}} = (\sqrt{2})^2 = 2$. That is certainly rational, so either way we have found an appropriate pair of irrational numbers. Or have we? Is x rational or not?

If we had an axiom `ExMid` in the AHA system we could formalize this proof. The axiom of excluded middle has the following form:

$$\text{ExMid} \qquad A \mid \neg A.$$

The proof would look something like this:

`OrElim (ExMid (Rational (`x`)),` $case_1$`,` $case_2$`)`

We claim for the AHA system that we can find the a and b in such proofs in a mechanical, simple way. If we had included the rule of excluded middle such a claim would require the interpreter to know which case to take in this instance. Clearly, that would not be easy here since it amounts to knowing whether $\sqrt{2}^{\sqrt{2}}$ is rational or not.

For the rules as we formulated them, it is easy to see how we can justify the claim that evaluation will be mechanical. Since an "or" formula can be proved only by `OrIntroL` or `OrIntroR` and thus the case is explicitly tagged, there will never be any problem deciding which case is true. It should also be apparent how a particular value can be computed, for a "some" formula can be proved in AHA only by `SomeIntro` and this requires a particular term to be supplied. This value can and will be used by the interpreter.

If we were interested in what theorems are true in classical Peano arithmetic, there is no reason why we could not go ahead and include `ExMid` as an axiom in the logic.

An interpreter for a AHA. As in the programming language ML and LISP, the interpreter for AHA will take an expression (term or proof expression) and return a simplified expression. Since we have lambda-terms in AHA we expect to find at least the beta-reduction rule in the interpreter. In fact there are several such reduction rules in AHA, and we will go through these rules now. Later we will see exactly what expressions we actually have to enter to the AHA system in order to evaluate programs like the integer division function.

The reduction rules dealing with conjunction are particularly simple.

`AndElimR (AndIntro (`$prex_1$`, `$prex_2$`))` $\longmapsto$ $prex_1$

`AndElimL (AndIntro (`$prex_1$`, `$prex_2$`))` $\longmapsto$ $prex_2$

Here is the reduction rule for beta-reduction.

`Application (Abstract (`x`, `b`), `t`)` $\longmapsto$ $[x/t]b$

The notation $[x/t]b$ means that the term t is substituted for the variable x in the term b. If any free variable of t is captured, the result is an error message. There is a rule for `AllElim` proof expressions.

`AllElim (AllIntro (`x`, `$prex$`),t)` $\longmapsto$ $[x/t]prex$

The rule for `ImpElim` is similar.

`ImpElim (ImpIntro (`hyp`, `$prex_1$`), `$prex_2$`)` $\longmapsto$ $[hyp/prex_2]prex_1$

It is worth considering this reduction a little more closely. The `ImpIntro` constructor enforces that the proof expression hyp is in the form `Assume (`$name$`, `A`)` where A is some formula. The `ImpElim` constructor enforces that whatever proof expression it has as its first argument, it is a proof of $A \Rightarrow B$. This much follows from the definition of the proof expression constructors. Clearly, the proof expression `ImpIntro (`hyp`, `$prex_1$`)` is one such proof expression proving $A \Rightarrow B$ for some formula B. But there are others, including some which are not necessarily implication-introduction expressions. The proof expression could be, for instance, a some-elimination or an or-elimination proof expression and still be a proof of $A \Rightarrow B$. One could say that the *type* of the first argument must be $A \Rightarrow B$. Of course, if the evaluation is to continue, then these expressions of type $A \Rightarrow B$ must evaluate to an implication-introduction expression so that the implication-elimination reduction rule can be applied.

The reduction rules for "or", "some" and induction complete the list of reduction rules.

OrElim (**OrIntroR** (A, $prex_1$), $prex_2$, $prex_3$) $\longmapsto$ **ImpElim** ($prex_2$, $prex_1$)

OrElim (**OrIntroL** (A, $prex_1$), $prex_2$, $prex_3$) $\longmapsto$ **ImpElim** ($prex_3$, $prex_1$)

SomeElim (**SomeIntro** (S, $prex_1$, t), y_0, $shyp$, $prex_2$)

$\longmapsto$ $[y_0/t]([shyp/prex_1]prex_2)$

AllElim (**Induction** (A, $prex_1$, n, $prex_2$), **Zero**) $\longmapsto$ $prex_1$

AllElim (**Induction** (A, $prex_1$, n, $ihyp$, $prex_2$), **Succ** x)

$\longmapsto$ $[n/x]([ihyp/$**AllElim** (**Induction** (...), x)$]prex_2)$

To insure that there is no misunderstanding about these evaluation rules, we show the implementation of the some-elimination rule. Like all evaluation rules the implementation is in two parts. First there is recognizing that the rule is applicable, and second there is performing the contraction. These two tasks are implemented in two functions, **IsSomeRedex** and **SomeContraction**. In order to understand these functions, we need to know the selector functions for the constructors **SomeIntro** and **SomeElim**. The arguments of **SomeIntro** are obtained by the selectors **SomeState**, **SomeProof**, and **SomeWitness**. The arguments of **SomeElim** are **Premis1**, **ChooseVar**, **SomeHyp**, and **Premis2**.

```
val IsSomeRedex exp =
  IsSomeIntro (Premis1 exp);

val SomeContraction exp =
  let val so = Premis1 exp in          { Must be some-intro prex. }
    Substitute (ChooseVar exp, SomeWitness so, ss)
      where val ss =
        Substitute (SomeHyp exp, SomeProof so, Premis2 exp)
      end
  end;
```

The collection of reduction rules defines the AHA interpreter completely. The AHA interpreter applies any of these reduction rules until none of them are applicable. The ML function that implements the interpreter is called **Reduce**.

Observe that there is nothing to prevent a proof expression with free variables or undischarged assumptions from being written and evaluated. We would not consider such proof expressions to represent proofs, but that causes no problem for the interpreter.

Subtraction example. With the basic normalizing procedure in mind, consider again the subtraction example. We gave a proof of the formula:

$$All\ x\ .\ (x = 0\ \mid\ Some\ y\ .\ y' = x)$$

The proof was represented by a proof expression of the basic form

Induction (All (x, P), *base*, x, *indhyp*, *indstep*)

Normalization will produce no change in this proof expression. It is already in normal form. Most often this will be the case unless the proof has needless steps, or if lemmas are incorporated.

The interpreter is not limited to the role of proof improver. The interpreter actually does computation. Consider the following proof expression still using the subtraction example.

AllElim (Induction (All x, P), *base*, x, *indhyp*, *indstep*), Zero)

The result of evaluating this proof expression is simply *base* or

OrIntroL (Reflexivity Zero, SOME (Zero))

When we apply the proof by induction to One we get the following chain of proof expressions.

[x/Zero]ImpElim (indstep, AllElim (Induction (...), Zero))

ImpElim ([x/Zero]indstep, OrIntroL (...))

ImpElim (ImpIntro ([x/Zero]indhyp, [x/Zero]prex3, OrIntroL (...)))

[indhyp/OrIntroL (...)]([x/Zero]prex3)

[x/Zero]prex3

OrIntroR (EQO (Succ Zero), *prex*)

The proof expression *prex* is:

SomeIntro (SOME (Succ Zero), Reflexivity (Succ Zero), Zero)

Notice the information we can extract from the proof expression obtained in evaluating the theorem at One. By the fact that the result is an OrIntroR proof expression, we know that there is some y such that $y' = x$. On further examination of the SomeIntro proof expression, *prex*, we see that the y in question is zero.

This theorem proving the existence of a predecessor has no special importance. However, looking carefully at this proof can yield an important moral. To prove that the predecessor existed required a proof that x was non-zero. This amounted to computing the predecessor which was then given to AHA in the some-introduction construct. Thus if one considers this "for-all there-exists" proof as a program, which given a number, computes its predecessor, then we find that we had do all the work ourselves. A programming logic does not find something that is not already there. There is no such thing as a free lunch.

Alpha-equality. We have not yet explained how the system treats alpha-variants: terms, types, formulas and proof expressions that differ only by renaming bound variables. In fact, the system does not distinguish between the alpha-variants of expressions. If the system did, then the rules would be much harder to use as formulas would have to be equal down to the variable names. For example, in the case of *modus ponens* of the propositional calculus the hypothesis of one premis must equal the other premis. But equal in what sense? If one premis is `Some(x,Equal(x,x))` and the other is:

```
Imply (Some (y, Equal (y,y)), True)
```

does *modus ponens* apply? Since the two existential formulas are alpha-variants the system treats them as equals.

Alpha-equality is especially important when the system chooses variables names. This does not happen directly, since the philosophy we have used in implementing these programming logics is to let the user choose the variable names at all times. An example of this is the some-introduction rule. The user must bear the consequences of a clash of variable names by renaming the variables. The system does not automatically rename variables. (A clash manifests itself either by a proof of a different formula than was intended or by an error being raised in substitution—not in an invalid proof!)

However, it is occasionally convenient for the system to choose the variable name in higher-level concepts. An example of this is the "less than" predicate in AHA.

```
val LE (t,s) = Some (x, Equal (Plus (t,x), s))
                   where val x = NewSystemVariable (Nat) end;
```

Now suppose the goal was to prove:

```
Some (z, Implies (LE(z,Zero), Equal (z, Zero)))
```

It would be inconvenient to force the user to know what variable name the system chose in order to prove that goal by some-introduction. Similar situations arise in the programming logics TT and IZF as well. The arrow operator in TT has a hidden variable, as does the subset predicate in IZF. The user should not be burdened with choosing the variable name or knowing it.

Computation rules, computational equality. In order to reason successfully about equality, additional rules are plainly needed. We have seen the need for beta-reduction in the proof of Cantor's theorem. Indeed there are a couple of such rules we have not mentioned yet. These rules are lumped together in the category called computation rules. Roughly these rules equated those terms that in the nature of the logic we expect to denote the same object.

Beta-reduction is one example. We expect that, due to the meaning of lambda abstraction, a lambda term $(\lambda x.b)$ applied to an argument t denotes the same thing as $b[x/t]$. We have seen that Cantor's theorem could not be proved without a rule codifying this fact. So we have a rule in AHA, which looks like this:

BetaReduction (s)**: Equal** $(s,\ b[x/t])$

where s is **Application (Lambda** $(x,\ b)$**,** t**)**. All computation rules have this general form:

CompReduction (s)**: Equal** $(s,\ t)$

where the term s has a pattern which is reduced by the computation rule to t.

Likewise, we expect that a term formed using recursion when applied to the argument zero is identical with the base case. In this case a term of the form

Application (Recursion $(base,\ \ldots)$**, Zero)**

is equal to the term $base$. This fact is codified in the **BaseCaseReduction** rule. Similarly the **IndReduction** rule is for terms of the form:

Application (Recursion $(b,\ n,\ v,\ t)$**, Succ** (s)**)**

If one term reduces to another, then it can be proved that they are equal terms. In light of this one may want to consider a construct in AHA

CompEquality (s)**: Equal** (s,t)

where t is the normal form of s.

We have mentioned that the AHA system uses alpha-equality in type-checking, so that names of bound variables are not important. There is one case where this is not enough and computational equality is used. The case occurs in the some-introduction constructor.

There are two factors that lead to the problem. The interpreter works by first reducing the arguments of any constructor. In the case of a hypothetical two-argument constructor **Constr**, we must have:

$$\texttt{Reduce (Constr } (a,\ b)) = \texttt{Constr (Reduce } a,\ \texttt{Reduce } b)$$

The other factor involved is the way type-checking works. If a and b are such that they pass all the type-checking constraints for the constructor **Constr**, then so must **Reduce**(a) and **Reduce**(b).

In the case of **SomeIntro** there is a problem. Here is an example of a some-introduction construct that illustrates the problem.

$$\texttt{SomeIntro (Some(}x,P(f(x))),\ prex : P(f(0)),\ \texttt{Zero)}$$

The type-checking constraint is

$$[x/0]P(f(x)) \equiv P(f(0))$$

Suppose, furthermore, that $f(0)$ reduces to 1, and, hence, $P(f(0))$ to $P(1)$. Then $prex : P(f(0))$ reduces to $prex' : P(1)$. But since **Some**(x,$P(f(x))$) does not reduce, the arguments cannot be put back together.

$$[x/0]P(f(x)) \equiv P(1)$$

The first equivalence holds for alpha-equality, but the second holds only if we reduce the expressions first. The inescapable consequence is that type-checking must involve the interpreter.

3.6 Extensions to Basic AHA

We have seen that recursion is a construct that aids in programming in AHA. Other programming constructs can be added to AHA by careful choice of rules. As long as rules are constructively valid, they can be implemented and added to AHA.

Axiom of choice. Using definition by recursion one can define new functions and prove properties about these functions. But in a constructive theory like AHA one should be able to take advantage of the fact that "for-all there-exist" proofs implicitly define functions. To this end we introduce a new rule into AHA, one which asserts the existence of functions. In principle we can do without definition by recursion altogether, although in practice one would want it.

Here is the statement of the axiom of choice which appears to be what we need:

$$\text{AC} \qquad \frac{All\ x\ .\ Some\ y\ .\ P(x,y)}{Some\ f\ .\ All\ x\ .\ P(x,f(x))}$$

Intuitively the implementation of the axiom of choice is clear. The value of $f(t)$ for any term t can be computed knowing the proof expression $prex : All\,x\,.\,Some\,y\,.\,P(x,y)$. The proof expression `AllElim` $(prex, t)$ reduces to a some-introduction proof expression:

$$\texttt{SomeIntro(Some(}y, P(t,y)\texttt{))}, prex : P(t,s), s)$$

The value $f(t)$ is s. And the proof that s is the right value for t is $prex : P(t,s)$.

Unfortunately there is not a clean way of implementing AC directly (as there is for Markov's principle below). The problem is that `SomeElim` must know how to reduce any proof of an existential formula. So in particular, when confronted with:

$$\texttt{SomeElim (AC (}prex : All\,x\,.\,Some\,y\,.\,P(x,y)\texttt{)},\ F,\ shyp,\ prex : Q)$$

the interpreter must be able to reduce it to a proof expression for Q without leaving the variable F free and the assumption $shyp$ undischarged.

The key here is to Skolemize the result. This requires introducing a new term constructor, we have called `Extract`:

$$\texttt{Extract (}prex : All\,x : A\,.\,Some\,y : B\,.\,P(x,y)\texttt{)}: \texttt{Arrow (}A,\ B\texttt{)}$$

Now we can rewrite the axiom of choice:

$$\text{AC} \quad \frac{All\,x\,.\,Some\,y\,.\,P(x,y)}{All\,x\,.\,P(x, Extract(x))}$$

or as a constructor:

$$\texttt{AC (}prex : All\,x\,.\,Some\,y\,.\,P(x,y),\ t\texttt{)}: P(t, \texttt{Application (Extract(}prex\texttt{)},t\texttt{)})$$

Markov's principle. Some people hold that the following rule, Markov's principle, should be true in a constructive theory. We will show how to add it to AHA.

$$\text{MR} \quad \frac{\neg\neg\, Some\,x : Nat\,.\,A(x), \quad All\,x : Nat\,.\,A(x)\ |\ \neg A(x)}{Some\,x : Nat\,.\,A(x)}$$

It is intuitively plausible that MR should hold, because we can start at zero and test all the natural numbers, one after another, to see if $A(x)$ holds. (Notice that to generalize MR for other types would still require that we can enumerate all the elements of the type systematically, as we can for the natural numbers.) We test a natural number n using the proof expression $All\,x : Nat.A(x)\ |\ \neg A(x)$. This proof expression applied to a natural number reduces to a proof expression which is either `OrIntroR` or `OrIntroL`. `OrIntroR` indicates that $A(n)$ holds; `OrIntroL` indicates it does not. Here is the ML function that implements the search.

```
val rec MarkovImpl (f: ProofExp, t: Term) =
  let val red = Reduce (AllElim (f,t)) in
    if IsOrIntroR (red)          { Reduce to OrIntroR proof exp?    }
      then (t, Premis1 red)      { Premis1(OrIntroR(prex,phi))=prex }
      else MarkovImpl (f, Succ t)
  end;
```

We know that sooner or later this search must terminate, since assuming there is no x such that $A(x)$ leads to a contradiction.

The recursive function MarkovImpl returns a pair. The first item is the term t for which $A(t)$ is true. The second item is the proof that $A(t)$ holds. These two pieces are exactly what the some-elimination rule must make use of to eliminate a use of the Markov principle.

A some formula introduced using MR

$$\texttt{SomeElim (MR (}prex_1\texttt{, }prex_2\texttt{), }x_0\texttt{, }shyp\texttt{, }prex:Q\texttt{)}$$

is eliminated using the some-elimination construct. This calls for a modification to the interpreter to accommodate the new construct. Here is the pertinent contraction.

```
val MarkovSomeContraction exp =
  let val mr = Premis1 exp;           { Must be MR prex. }
      val (t,Pt) = MarkovImpl (Premis2 mr, Zero) in
        Substitute (ChooseVar exp, t, ss)
          where val ss = Substitute (SomeHyp exp, Pt, Premis2 exp)
          end
  end;
```

Finiteness of floating-point numbers. Now we want consider another example from a completely different context. Suppose that we were interested in floating point calculations done on some data type Float. If we could axiomatize the results of some machine's operations on floating-point numbers we could reason about these calculations. One axiom we might wish to express about floating-numbers is that, unlike the real numbers, there are only a finite number of floating-point numbers. There are several ways of formalizing this. One way is the pigeonhole principle. If there are only a finite number of floating-point numbers, and one has an infinite number of them, then two of them are the same. Here is one formalization of this:

$$\mathit{All}\ f : \mathit{Nat} \to \mathit{Float}\ .\ \mathit{Some}\ i, j : \mathit{Nat}\ .\ i \neq j\ \&\ f(i) = f(j)$$

There appears to be no way to implement such an axiom short of comparing the function f on every pair of numbers. This is hopelessly too inefficient to use in practice. Another way of formalizing what it means for the floating-point numbers to be finite can be expressed in two axioms.

$$\mathit{All}\ f : \mathit{Nat} \to \mathit{Float}\ .\ \mathit{Some}\ i : \mathit{Nat}\ .\ f(i) \geq f(i+1)$$

$$\mathit{All}\ f : \mathit{Nat} \to \mathit{Float}\ .\ \mathit{Some}\ i : \mathit{Nat}\ .\ f(i) \leq f(i+1)$$

The i asserted to exist can be found by testing each successive value of the function. This, too, appears to be of dubious use, but these axioms can be used successfully in practice. Consider for a moment the program which computes the root of a polynomial using Newton's approximation. The sequence of successive approximations $x_0, x_1, \ldots$ is a function of the appropriate type, namely $\mathit{Nat} \to \mathit{Float}$, so that the axioms apply. The i found by the implementation of these axioms is the index of interest in the sequence of successive approximations. Under the appropriate conditions x_i is the closest **Float** number to the root of the polynomial.

3.7 The integer division function

We will prove that for every pair of numbers x and y ($y > 0$) there is a pair of numbers q and r which are the quotient and remainder of dividing x by y. Then we will use the interpreter to find the q and r given particular values of x and y.

Many details are glossed over to keep the exposition moving. Some abbreviations are used without being introduced. Proofs needed about elementary arithmetic are assumed magically to exist or to have been proved somewhere. We note that in the following proof the "$a < b$" relation is defined as an abbreviation for $\mathit{Some}\ v\ .\ a + v = b\ \&\ \neg\ v = 0$.

```
{   All x,y. 0<y => Some q,r. x=y*q+r & r<y   }

val x,y = Variable ("x", Nat), Variable ("y", Nat);
val hyp  = Assume ("hyp", LT(Zero,y)); { 0<y:  Some v 0+v=y & ~ v=0 }
val v0 = Variable ("v0", Nat);
val shyp = Assume ("shyp", LThan (Zero, v0, y));
val prex1 = AndIntro (Magic(Less(x,v0,Add(x,y))), AndElimL shyp);
```

```
val prex2 = SomeIntro (LT(x,Add(x,y)), prex1, v0);
val prex3 = SomeElim (hyp, v0, shyp, prex2);
{ ProofOf (prex3) = x<x+y = Some v. x+v=x+y & ~ v=0 }
val prex4 = AndIntro (prex3, hyp);
val prex5 = ImpElim (nAllElim (DivLemma, [x;x;y]), prex4);
val prex6 = ImpIntro (hyp, prex5);
val proof = AllIntro (x, AllIntro (y, prex6));
val DivThm = Definition ("DivThm", proof);
```

Even after this much code we see that the real work has been left to a lemma which can be proved by induction. For the moment we shall just state the lemma.

$$All\, n\,.\, All\, a,b\,.\, a < n + b \,\&\, 0 < b \;\Rightarrow\; Some\, q,r\,.\, a = b * q + r \,\&\, r < b$$

Now suppose we have that **gt0** is a proof in AHA that two is greater than zero. Now consider the following expression:

```
ImpElim (AllElim (AllElim (DivThm, Three), Two), gt0)
```

The result of evaluating this expression will exhibit two values q and r that are the quotient and remainder of three divided by two. Notice that the AHA expression above is like a function call. The third argument is a proof. This is like passing the proof of the precondition to the function as certification that it is proper to call the function. In this manner the integer quotient of three and two has been computed. Let us see how this done in detail.

The first action the interpreter takes is to replace the variables x and y in **DivThm** with the the terms **Three** and **Two**. If the proof **gt0** had not been passed along with the other two arguments, then the evaluation could proceed essentially unchanged. We will return to this point later. As it is, hyp can be replaced by **gt0** in **prex4**. The proof expression prex4 is a proof of $3 < 3 + 2 \,\&\, 0 < 2$. Proof expressions one, two and three take the witness v0 that $0 < 2$, namely 1, and use it as the witness that $3 < 3 + 2$. The stage is set for a call to **DivLemma** with arguments **Three**, **Three**, **Two**, and **prex4**.

This requires that we take a little deeper look at the lemma. The lemma is proved by induction. The base case follows (relatively easily) by choosing zero for q and a for r.

```
val prex13 = SomeIntro (S0, prex12, a);         { Choose a for r.   }
val prex14 = SomeIntro (SM(a,b), prex13, Zero);{ Choose Zero for q.}
```

```
val prex15 = ImpIntro (hyp, prex14);
val base = AllIntro (a, AllIntro (b, prex15));
```

The induction step runs like this. We need to prove:

$$All\, a,b\,.\, a < n' + b \,\&\, 0 < b \;\Rightarrow\; Some\, q,r\,.\, a = b * q + r \,\&\, r < b$$

So assume $a < n'+b$. Therefore $a \leq n+b$ and so either $a < n+b$ or $a = n+b$. In the first case apply the induction hypothesis to a and b. In the second case, $a = n + b$, apply the induction hypothesis to n and b. From $Some\, q_1, r_1\, .\, n = b * q_1 + r_1 \,\&\, r_1 < b$ one can show $a = b * (q_1 + 1) + r_1 \,\&\, r_1 < b$, so choose $q_1 + 1$ for q and r_1 for r.

```
val di = ImpElim (AllElim (AllElim (Dichotomy, a), Add(n,b)), hyp');
                                                    { a<n+b or a=n+b }
val prex17 = OrElim (di, prex5, prex16);
val prex18 = ImpIntro (hyp, prex17);
val prex19 = AllIntro (a, AllIntro (b, prex18));
val proof = Induction (Thm, base, n, indhyp, prex19);
val DivLemma = Definition ("DivLemma", proof);
```

Case one says that quotient and remainder are computed directly from the values given by the induction hypothesis.

```
val case1 = Assume ("case1", LT(a,Add(n,b)));
val prex3 = AllElim (AllElim (indhyp, a), b);
val prex4 = ImpElim (prex3, AndIntro (case1, AndElimL hyp));
val prex5 = ImpIntro (case1, prex4);  { end of case 1 }
```

This case is more complicated. Given q_1 and r_1 provided by the induction hypothesis one can show that $q_1{}'$ and r_1 are the values of the quotient and remainder.

```
val prex12 = SomeIntro (Some (r,AD (a,b,Succ q1,r)),prex11,r1);
val Ty = Some (q, Some (r, AD (a,b,q,r)));
  { Some q,r. a=b*q+r & r<b }
val prex13 = SomeIntro (Ty, prex12, Succ q1);
```

```
val prex14 = SomeElim (s1hyp, r1, s2hyp, prex13);
  { Some q,r. a=b*q+r & r<b }
val prex15 = SomeElim (prex8, q1, s1hyp, prex14);
val prex16 = ImpIntro (case2, prex15);
```

We now continue computing **Three** divided by **Two**. The division lemma is called with arguments **Three**, **Three**, **Two** and **prex4**. Since the lemma is proved by induction we look for induction reductions. The base case reduction is not applicable. The lemma is called recursively; this time with arguments $n = 2$, $a = 3$ and $b = 2$. This time case two is applicable. We remember that a one must be added to the final quotient, and we call the lemma again, this time with arguments $n = 1$, $a = 1$ and $b = 2$. Case one is applicable and the lemma is called again, with arguments $n = 0$, $a = 1$ and $b = 2$. Now we have arrived at the base case. The result is $q = 0$ and $a = 1$ plus the one we must add to q. This makes the final answer $q = 1$ and $a = 1$.

The resulting proof expression is:

```
SomeIntro (..., SomeIntro (...,...,One), One)
```

The computational content of the integer division function can be summarized in two ML functions.

```
val rec DivLemma (n,a,b) =
  if n=0 then (0,a)
    else if a<(n-1)+b then DivLemma (n-1,a,b)
      else let val q1,r1 = DivLemma (n-1,n-1,b) in
              (q1+1, r1)
           end;

val DivThm (x,y) = DivLemma (x,x,y);
```

One might want to compare the above program with the one derived in [Manna and Waldinger 1980].

3.8 Example: maximum segment sum

It is difficult to be convinced that programming logics have any practical value, if the integer division function is all that can be programmed, and that at tremendous effort. In the next chapter we will launch an assault on the morass of details which chain the proofs we can do to the lowest levels. In this section we wish to present a more convincing example. It is more convincing for two reasons. First, it is intellectually more challenging. We will make good use of higher-order functions. Second, this example will raise the issue of efficiency. A programming logic will fail to be an adequate programming language if efficient algorithms cannot be expressed or if efficiency cannot be determined or affected. There are many factors affecting efficiency. The choice of operators is a prime one. Most of the power of AHA comes from recursion and induction. These are simple and elegant operators which are adequate in theory. Perhaps less general ones may provide the means for more efficient computation. In this section we will see that the careful choice of proofs greatly affects the speed of execution.

The example we choose, the maximum segment sum problem, has been used as an example for PRL [Bates and Constable 1983]. The maximum segment sum problem concerns a finite sequence

$$f(0), \quad f(1), \quad f(2), \quad \ldots, \quad f(n)$$

of *integers* (it is not interesting to limit the problem to just natural numbers). The problem is to find the maximum sum of any contiguous subsequence (or segment). Thus, for the sequence

$$31, \quad -41, \quad 59, \quad 26, \quad -53, \quad 58, \quad 97, \quad -93, \quad -23, \quad 84$$

the answer is $\sum_{i=2}^{6} f(i) = 187$. A complete history of this problem and a full discussion can be found in [Bentley 1984]. See also [Gries 1982].

For the discussion in this section we need to introduce two higher-order functions. These functions are easily programmed by recursion in AHA. The first function computes the maximum of any segment:

$$FuncMax(f,k,l) = \max_{k \leq i \leq k+l} f(i).$$

The second function computes the sum of any segment:

$$FuncSum(f,k,l) = \sum_{k \leq i \leq k+l} f(i).$$

The type of *FuncMax* and *FuncSum* in AHA is

```
Arrow (Arrow (Nat,Int), Arrow (Nat, Arrow (Nat, Int)))
```

where **Int** is a new basic type that we must introduce to AHA.

Now we wish to compute the maximum segment sum given f and n. There are two ways of going about this. The first way is to prove an appropriate "for-all there-exists" theorem and use the axiom of choice to extract the function. (This is how we approached the integer division example.) The necessary theorem might be:

$$All\ f, n\ .\ Some\ M\ .\ Some\ a, b\ .\ All\ p, q\ .$$

$$(p \leq n \,\&\, q \leq n-p) \;\Rightarrow\; M = FuncSum(f, a, b) \,\&\, M \geq FuncSum(f, p, q)$$

A weakened version of the above theorem permits more efficient computation of the maximum segment sum.

$$All\ f, n\ .\ Some\ M, L\ .\ Some\ a, b, c\ .\ All\ p, q\ .\ (p \leq n \,\&\, q \leq n-p) \;\Rightarrow$$

$$M = FuncSum(f, a, b) \,\&\, M \geq FuncSum(f, p, q) \,\&$$

$$L = FuncSum(f, c, n-c) \,\&\, L \geq FuncSum(f, p, n-p)$$

This approach is fully developed in [Bates and Constable 1983].

The second approach is to define the function outright that computes the maximum segment sum. (We used this approach for Ackermann's function.) The term $FuncMax(G_n, 0, n)$ where $G_n(k) = FuncMax(H_k, 0, n-k)$ and $H_k(q) = FuncSum(f, k, q)$ computes the maximum segment sum. How do we know? Using computation rules of the recursion term constructor we can prove the first theorem above. Upon closer examination we may not be satisfied with defining the maximum segment sum this way. The number of induction reductions performed by the functions *FuncMax* and *FuncSum* to compute the maximum segment sum is:

$$\sum_{k=0}^{n} \sum_{q=0}^{n-k} (q+1) = O(n^3).$$

It seems natural to try if we can make weakening apply here. So we start with the following theorem:

$$All\ f, n\ .\ Some\ M\ .\ M = FuncMax(G_n, 0, n)$$

The proof of the theorem is trivial—just introduce the term $FuncMax(G_n, 0, n)$. But a weakened version of the theorem might permit more efficient computation of the maximum segment sum. Consider the following theorem:

$$All\ f, n\ .\ Some\ M, L\ .\ M = FuncMax(G_n, 0, n) \,\&\, L = FuncMax(G'_n, 0, n)$$

where $G'_n(k) = FuncMax(H_k, n-k, n)$. Intuitively L is to stand for the maximum segment sum with the final endpoint at n. The values of M and L can be computed easily by recursion on n. For $n = 0$, we take $M = f(0)$ and $L = f(0)$. If M_n and L_n have already been computed, we take

$$L_{n+1} = \max\{f(n+1), L_n + f(n+1)\}, \qquad M_{n+1} = \max\{M_n, L_{n+1}\}.$$

The function to compute the maximum segment sum implicit in this last theorem is a linear time algorithm (if the plus and compare functions are implemented in constant time).

To arrive at the weakening that computes L along the way takes ingenuity. It is not at all clear how one arrives at such an insight in the programming logic paradigm. David Gries [Gries 1982] develops this insight in the paradigm of while-programs and Hoare logic. Perhaps this is evidence that the science of programming logics has not been developed. Also worth considering is the possibility of introducing constructs from sequential programming like the while-loop to programming logic. This has been anticipated in [Constable and O'Donnell 1978].

We give the beginnings of the proof in AHA of this last theorem. Rather than develop the proof in the linear style we present the resulting proof expression in a modified form.

```
Let H (q) = FuncSum (f, k, q)
Let G (k) = FuncMax (H, 0, n-k)
Let G'(k) = FuncMax (H, n-k, n)

AllIntro
  Let f : (Nat->Nat) be arbitrary
  Induction
    To show : All n. Some M,L. M=FuncMax(G,0,n) & L=FuncMax(G',0,n)
    Base case
      SomeIntro
        To show : M=FuncMax(G,0,0) & L=FuncMax(G',0,0)
        With f(0) for M and f(0) for L
        ?? : f(0) = FuncMax (G,0,0) & f(0) = FuncMax (G',0,0)

    Induction variable   n
    Induction hypothesis
      ihyp : Some M,L. M=FuncMax(G,0,n) & L=FuncMax(G',0,n)
    Induction step
      SomeElim
        Choose Mn,Ln where Mn=FuncMax(G,0,n) & Ln=FuncMax(G',0,n)
        SomeIntro
          To show : Some M,L. M=FuncMax(G,0,n) & L=FuncMax(G',0,n)
          With Max {f(n'), Ln+f(n')} for L and
               Max {Mn, L} for M
          ?? : Max {Mn, L} = FuncMax (G,0,n') &
             Max {f(n'), Ln+f(n')} = FuncMax (G',0,n')
```

The question marks indicate where additional proof expressions must be filled in. These details have to do with the inner workings of *FuncMax* and *FuncSum* and can be found in [Bates and Constable 1983]. Despite the missing pieces, the interpreter correctly computes the maximum segment sum. The pieces do not affect the computational content of the proof.

Chapter IV

The TT and IZF programming logics

In this chapter we will describe two other programming logics, TT and IZF. The AHA programming logic of the last chapter has the natural numbers as its area of application. The two programming logics of this chapter concern entirely different subjects: types and sets. At first types and sets may appear to have nothing in common with natural numbers. Actually, the technique of representation is the same, and the same logic engine used for the AHA programming logic can, and has, been used for these two. The diversity of the theories implemented this way in ML suggests that this technique may be useful for experimenting with theories in which proofs are executable. Furthermore, we can motivate the first of these programming logics by considering the consequences of dissolving the distinction between formulas and types and between terms and proof expressions. This is the subject of the first section in this chapter. The following sections present the programming logic TT and introduce transfinite induction for TT. In Section 4.4 we briefly consider representing logic directly in the type structure of programming languages. The final section of this chapter presents the programming logic for sets, IZF.

4.1 Axiom of choice

When the distinction between formulas and types and between terms and proof expressions is dropped there are some interesting consequences. In this section we consider dropping that distinction and find that a great simplification takes place. This leads to a new programming logic which will be described in the next section. This new theory will have the property that the axiom of choice is provable.

Terms, types, formulas and proof expressions (more commonly called proofs in this context) are syntactic categories used to define the language of logic. We have found it convenient to divide the expressions of the AHA programming logic into terms and proof expressions in keeping with the usual mathematical exposition. AHA is a strongly typed language so these expressions, be they terms or proof expressions, have types. These types appear after the colon in a dialog with an AHA system. Now we must be careful with nomenclature here. The syntactic category of types we shall call *data types* to distinguish them from the types of AHA expressions. The types in AHA of those expressions that are terms are data types, and the types of proof expressions are formulas.

Some of the constructs for terms are similar to constructs for proof expressions. The only reason that two constructs were needed was because having only one construct for both terms and proof expression is not customary in logic. Consider the constructs `Variable` and `Assumption`. `Variable` does for terms what `Assumption` does for proof expressions. They each declare a hypothetical member of a given type, and they give it a printing representation. A variable gets bound and an assumption gets discharged using the `Abstract` and `ImpIntro` constructs respectively. These constructs coalesce. This suggests that the type constructor `Arrow`(A,B) and the connective `Imply`(A,B) are the same. Further simplification occurs when the arrow operator is seen as a special case of the universal quantifier. For example, $All\, x : Nat . Nat$ can be written $Nat \Rightarrow Nat$, since the variable x does not occur free in the scope of the quantifier. The result of this simplification is that induction and definition by recursion become the same construct. We give examples.

Recursive functions. We show how to make a recursive definition using the induction construct. The example we choose is the "plus a" function:

$$f(0) = a, \qquad f(n') = f(n)'.$$

The definition is accomplished as follows:

`Induction (Imply (Nat,Nat), a, n, x, Succ x)`

Notice we have an implication as the first argument where we have usually had an "all" formula. Furthermore, the implication is between data types (`Nat` to be specific). The curried plus function can be defined, too.

```
val x = Variable ("x", Nat);
val base = Abstract (x, x);
val T = TypeOf base;
val F = Variable ("F", T);
val ind = Abstract (x, Succ (Application (F, x)));
val PlusFunc = Induction (Imply (Nat, T), base, n, F, ind);
```

We formulated the `Recursion` construct with just four arguments, because the type of the term can easily be computed. The `Induction` construct has five arguments so that exact form (variable name included) is specified.

Recursive data types. Besides number-theoretic functions, data types can be defined using the `Induction` construct as well. One example is defining the type for lists of integers.

```
val n = Variable ("n", Nat);
val base = Unit;
val TYPE = TypeOf base;
val F = Variable ("F", TYPE);
val ind = Cross (Nat,F);
val ListN = Recursion (Arrow (Nat,TYPE), base, n, F, ind);
```

`ListN` is a function which, given a number n, returns the data type:

$$\texttt{Nat} \times \ldots \times \texttt{Nat} \times \texttt{Unit}$$

Now the type of integer lists can be created using `Some`.

```
val x = Variable ("x", Nat);
val List = Some (x, Application (ListN, x));
```

Of course, this definition of list does not depend in any way on the fact that the base elements are natural numbers. By quantifying over types a new definition of list is achieved.

```
val T = Variable ("T", Universe 1);
val List' = All (T, Some (x, Application (ListN', x));
```

One obtains parametric polymorphism this way. The type of lists of natural numbers is `AllElim (List',Nat)`.

The constant list "nil" can be easily be programmed.

```
{ Unit = Application (ListN, Zero) }
val prex1 = BaseCaseReduction (Application (ListN, Zero));
val prex2 = EqualityElim (prex1, It);
val Nil = SomeIntro (List, prex2, Zero);
```

The functions for CONS, LENGTH and NULL are not much harder. Here we have a pointerless implementation of lists. Using induction all the usual properties of lists can be obtained.

Axiom of choice. Not only can the constructs be used in new ways when the distinction between formulas and types is dropped, but new results also hold. One result is that the axiom of choice in the form of

$$All\,P\,.\,(All\,x\,.\,Some\,y\,.\,P(x)(y)\;\Rightarrow\;Some\,G\,.\,All\,z\,.\,P(z)(G(z)))$$

is provable if we modify the some-elimination rule a little.

To prove the axiom of choice, we must construct a term F such that $All\,z\,.\,P(z)(F(z))$, given a proof expression *Proof* which is a proof of $All\,x\,.\,Some\,y\,.\,P(x)(y)$. The term F that we want is unlike anything we have seen in AHA, because it crosses the boundary between proof expressions and terms. By eliminating the boundary it becomes possible to express such a term. We will build the term F up gradually using some-elimination on the proof expression $Proof(v)$ =`AllElim`($Proof$,v) which is a proof of the "some" formula $Some\,y\,.\,P(v)(y)$. We set

`Wit`(a)=`SomeElim` (a, w, $shyp : \phi$, w)

where the formula ϕ is chosen to satisfy the requirements of the `SomeElim` rule. The variable w has the same type as y. In the present case we have:

`Wit`($Proof(v)$) = `SomeElim` ($Proof(v)$, w, $shyp : P(v)(w)$, w)

We see that the assumption $shyp$ is ignored completely and that the variable w appears in the last argument of the some-elimination rule, where one would expect a proof expression. But we are no longer distinguishing terms and proof expressions, so this is permissible. We expect a `SomeElim` expression to be a proof expression proving a formula. This formula is the one that the last expression is a proof of. In this case, the expression is w and is not a proof of any formula, but is a term of some data type, perhaps `Nat`. This is mixing formulas and types. The conclusion is that the distinction between formulas and types no longer makes any sense, if the distinction between terms and proof expressions is dropped. Now we are ready to give the term F:

F =`Abstract` (v, `Wit` ($Proof(v)$))

Now we ask the question: why does F work? The key requirement is that this term, when applied to some value, reduces to the witness chosen in the given proof. Recall how the some-elimination reduction rule operates:

$$\texttt{SomeElim (SomeIntro } (S,\ exp_1,\ t),\ x_0,\ shyp,\ exp_2) \longmapsto [x_0/t]([shyp/exp_1]exp_2)$$

We have

$$\texttt{SomeElim (SomeIntro } (S,\ exp_1,\ t),\ w,\ shyp,\ w)$$

and it reduces to

$$[w/t]([shyp/exp_1]w)$$

or simply t. In all previous uses of **SomeElim** the result has been a proof expression. Now it is a term. This is the term we want, one that takes a proof expression to the witness.

It should be clear that to complete the proof of the axiom of choice, we will need a proof that $P(z)(F(z))$ is true. The obvious candidate for such a proof is an expression much like **Wit**. Consider the following expression, call it **PrWit**.

$$\texttt{PrWit}(Proof(v)) = \texttt{SomeElim } (Proof(v),\ w,\ p,\ p)$$

where p is **Assume** ("p", $P(v)(w)$). At first glance it appears that this some-elimination expression is normal—it does not cross the boundary between terms and proof expressions. But on further examination we see that the expression violates the restrictions on the use of the some-elimination rule which prohibits using the assumption $P(v)(w)$ to prove anything containing w free. In fact this some-elimination expression appears to prove nothing other than $P(v)(w)$. This matter is resolved by carefully reconstructing the some-elimination rule.

If we ignore the distinction between terms and proof expressions, then we need to construct a some-elimination rule with a little more subtlety. Recall how we have defined the **SomeElim** rule:

$$\texttt{SomeElim } (prex_1 : \texttt{Some}(x,P),\ x_0,\ shyp : P(x_0),\ prex_2 : Q)$$

This rule comes with the restriction that x_0 does not occur free in Q. It is a simple matter to conclude that this some-elimination term should have type Q. It is not as clear what type the expression should have for a more general class of some-elimination constructs:

$$\texttt{SomeElim } (exp_1 : \texttt{Some}(x,P),\ x_0,\ y,\ exp_2 : T)$$

Here T can be more general—with x_0 and y free. The type of this construct cannot be $Q(x_0, y)$, for x_0 and y would be free. We want the expression

$$\texttt{SomeElim (SomeIntro } (S,\ exp_1,\ t),\ x_0,\ y,\ exp_2 : T)$$

to have type $Q(exp_1, t)$, but sometimes exp_1 and t may not be available. For instance, in the case of an assumption

$$\texttt{SomeElim (Assume ("as", } S\texttt{), } x_0\texttt{, } y\texttt{, } exp_2\texttt{)}$$

there is no witness and proof to with which to substitute in for x_0 and y. The obvious solution is to find terms which when supplied with an actual some-introduction expression will reduce to the desired witness and proof. Such terms were already found for the proof of the axiom of choice: `Wit` and `PrWit`. Thus the type of expression

$$\texttt{SomeElim } (exp_1 : \texttt{Some(}x\texttt{,}P\texttt{), } x_0\texttt{, } y\texttt{, } exp_2 : T\texttt{)}$$

is $T(\texttt{PrWit}(exp_1), \texttt{Wit}(exp_1))$.

We continue the proof of the axiom of choice. Since `PrWit` $(Proof(v))$ is a proof of

$$P(v)(\texttt{Wit}\,(Proof(v)))$$

and $F(v) =$`Wit` $(Proof(v))$, it is just a matter of ironing out the details to complete the proof of the axiom of choice.

We now give all the details to the proof of the axiom of choice. Since the proof holds for binary propositional functions of any type, we give the proof in that setting.

```
val T1 = Variable ("T1", Universe 1);        { Type variable  }
val T2 = Variable ("T2", Universe 1);        { Type variable  }
val P = Variable ("P", Arrow (T1, Arrow (T2,Prop)));
```

Next we declare all the variables we need in the proof.

```
val x = Variable ("x", T1);                 { More variables.  }
val y = Variable ("y", T2);
val z = Variable ("z", T1);
val w = Variable ("w", T2);
val G = Variable ("G", Arrow (T1,T2));
```

Now we define some abbreviations and types that come up in the proof.

```
val AAP(a,b) = Application (Application (P,a),b);
val Gz = Application (G,z);
```

```
val A = All (x, Some (y, AAP(x,y)));                { Types. }
val S = Some (G, All (z, AAP(z,Gz)));
```

Finally the proof begins.

```
          { Assume proof of All x. Some y. P(x)(y) }
val hyp = Assume ("hyp", A);
val prex1 = AllElim (hyp, z);   { Some y . P(z)(y)  }
val F = Abstract (z, Wit prex1);
  { Application (F,z)=Wit prex1 }
val prex2 = BetaReduction (Application (F,z));
```

The next part of the proof proves from the preceding equality

```
Application (F,z)=Wit prex1
```

that

```
P(z)(Application (F,z))=P(z)(Wit prex1)
```

holds. This is readily believed since it is justified by substitution; here we prove formally using reflexivity, symmetry, transitivity and congruence.

```
val con = Congruence (Reflexivity (Abstract (w, AAP (z,w))), prex2);
val Tc = TypeOf con;           { way of getting types rhs and lhs. }
val rhs = RHSEqual (Tc);  val lhs = LHSEqual (Tc);
val sym = Symmetry (BetaReduction lhs);
val trn = Transitivity (con, (BetaReduction rhs));
val prex3 = Transitivity (sym, trn);
```

From the fact that P(z)(Wit prex1) holds, the proof concludes easily.

```
{
   P(z)(Application (F,z)) = P(z)(Wit prex1),  P(z)(Wit prex1)
   -----------------------------------------------------------
                   P(x, Application (F,z))
}
val prex4 = EqualityElim (prex3, PrWit prex1);
val prex5 = SomeIntro (S, AllIntro (z, prex4), F);
```

```
val axchoice = AllIntro (P, ImpIntro (hyp, prex5));
```

This ends the proof of the axiom of choice. This proof has important consequences for a theory that makes no distinction between formulas and types. (We shall see such a theory in the next section.) Every "for-all there-exists" proof has a function term in the theory

```
Abstract (z, Wit (AllElim (proof, z)))
```

which is the function implicitly defined in the proof. This can be proved using the term **PrWit**.

4.2 Type theory

We have suggested an entirely different theory in which there is no distinction between terms and proof expressions, and formulas and types. This theory could get by with fewer constructs than AHA. This section describes the programming logic TT (Type Theory) that results.

Propositions as types. Equating formulas and data types is known as the "proposition-as-types" notion. It is attributed to William Howard, although as early as 1958, Haskell Curry and R. Feys observed the similarity between the axioms of propositional calculus and the basic combinators, and W. Tait remarked in 1965 on the correspondence between the cut elimination theorem and the reduction of lambda terms. Independently in the 1960's H. Laüchli and Nicolaas de Bruijn have observed the much same thing. Martin-Lõf is the first to take it to the extremes that we intend to take it here, or that Nu PRL does [Constable and Bates 1984].

Everything is a type. We have seen that quantifying over propositions is occasionally useful. This leads to a small problem. Propositions or formulas are now types. Yet quantification is over terms; that is, in the formula

$$All\, P : Prop\, .\, All\, x : Nat\, .\, P(x)$$

P and x act like terms of type **Prop** and **Nat**, yet $P(x)$, or **Application** (P, x) is a type. (We saw this problem once before in AHA.) The best solution is to make terms and types the same kind of object—call them both expressions. Still every expression must have a type. There must be a function from expressions to expressions called **TypeOf**. Those expressions in the range of **TypeOf** are called types in certain contexts. Those expressions in the domain of **TypeOf** that not are in the range are called elements. For example, **Tr** of type **Bool** is an element. There is no expression exp such that **TypeOf** (exp) =**Tr**.

Since all expressions have a type, what is the type of **Prop**? We make an open-ended hierarchy by assigning the type of **Prop** to be **Prop2**, the type of **Prop2** to be **Prop3**, and so on.

Presentation of TT. The rules of TT are similar to the rules of AHA. We have given them different names so as to cause less confusion.

The basic data types provided in TT are **Void** with no elements, **Unit** with one element **It**, and **Bool** with two elements: **Tr** and **Fl**.

The **VoidElim** rule looks like the false elimination rule of AHA.

VoidElim (exp: **Void**, A): A

This rule means that if there is an element of **Void**, then **VoidElim** is an element of any type desired A.

Pair, **First**, and **Second** resemble the rules for **And** in AHA.

Pair ($exp_1 : A$, $exp_2 : B$): **Cross** (A,B)

First (exp: **Cross**(A,B)): A

Second (exp: **Cross**(A,B)): B

The rules for **Union** resemble those for **Or**.

InR ($exp : A$, B): **Union** (A, B)

InL (A, $exp : B$): **Union** (A, B)

Out (exp_1 : **Or**(A,B), exp_2 : **Arrow**(A,C), exp_3): C

The dependent arrow type corresponds to **All**.

PiIntro ($x : A$, $exp : B(x)$): **Pi** ($x : A$, $B(x)$)

PiElim (exp: **Pi** ($x : A$, $B(x)$), $t : A$): **B(t)**

The dependent arrow is sometimes written $\Pi\, x : A\,.\,B(x)$. The arrow construct is a special case and can be rendered by an ML abbreviation.

```
val Arrow (A,B) =
  Pi (x, B) where val x = NewVariable A end;
```

The dependent sum type corresponds to **Some**.

SigmaIntro (**Sigma**($x : A,B$), $exp : B(x_0)$, $x_0 : A$): **Sigma**($x : A,B$)

SigmaElim (exp_1 : **Sigma**($x : A,B$), $x_0 : A$, $shyp : B(x_0)$, $exp_2 : Q$): Q

The dependent sum is sometimes written $\Sigma\, x : A\, .\, B(x)$. If there is a conditional operator in the language, then the **Union** can be eliminated.

A complete set of rules for TT can be found in the appendix.

4.3 Brouwer's operator for ordinal recursion

In AHA the domain of application is number theory. The successor function and induction are simple, elegant constructs from which the edifice of number theory in constructed. The system TT is more amorphous, yet exceedingly rich. It has been suggested that such a theory with a few key constructs added promises to express everything of interest in computer science. But the absence of such a simple data type as the natural numbers causes a feeling of disorientation. In this section we wish to examine a new operator which generalizes the notion of natural number and induction that we had in AHA. We do this as the sole exercise in this thesis of the TT system, and as an aid in comparing TT with AHA.

In this section we introduce a constructor for new expressions corresponding to ordinals. It has been proposed for type theory by Martin-Löf under the parsimonious designation of **W** [Martin-Löf 1982, Constable and Zlatin 1984, Nordström 1981], although its origins go back to Brouwer [Stenlund 1972]. We choose to call the new constructor **Ord**. This is its form:

$$\texttt{Ord}(x : T,\ S(x))$$

This is exactly the same form that the dependent arrow operator takes. The variable x of type T (the first argument) is bound in the expression $S(x)$ (the second argument) which may have an occurrence of x.

The **Ord** constructor has an introduction rule and elimination rule. The introduction rule is relatively straightforward.

$$\texttt{Sup}\ (t : T,\ f : \texttt{Arrow}\ (S(t),\ \texttt{Ord}(x : T,\ S(x))))$$

This construct has type **Ord**($x : T, S(x)$). Thus **Sup**(t,f) is an ordinal. The type T acts as a collection of labels. The ordinal labeled t has as a predecessor ordinal $f(t)$. It appears that this definition is circular, since to introduce an ordinal using **Sup** one must already have a function from labels to ordinals. This apparent circularity is overcome by using an **If** construct, as will be seen in the example of natural numbers given later.

The elimination rule for ordinals is more complicated. Here is its natural-deduction style formulation:

$$\dfrac{exp_1 : Ord(x:T, S(x)), \qquad \begin{array}{c} u : T \\ v : S(u) \to Ord(x:T, S(x)) \\ w : \Pi y : S(u).C(v(y)) \\ \vdots \\ exp_2 : C(Sup(u,v)) \end{array}}{C(exp_1)}$$

The subproof computes the result C for the next ordinal $Sup(u, v)$ given the result for any possible predecessor. This is induction on the ordinals or transfinite induction. The form of the transfinite constructor is (we ignore the problem of determining the type C):

$$\texttt{Transfinite}\ (exp_1 : Ord(x:T, S(x)), u, v, w, exp_2 : C(Sup(u,v))) : \ C(exp_1)$$

where u is a variable of type T, v is a variable of type $S \to Ord(x : T, S(x))$, and w is a variable of type $\Pi y : S.C(v(y))$. The variables u, v and w can all occur free in the expression exp_2. The expression exp_2 must have the type $C(Sup(u,v))$.

We must add a contraction to the evaluator to complete the introduction of the ordinals:

$$\texttt{Transfinite (Sup}\ (t,\ f),\ u,\ v,\ w,\ exp_2) \quad \longmapsto \quad exp_3$$

where expression exp_3 is expression exp_2 with t substituted for u, and with f substituted for v, and with

$$\texttt{PiIntro}\ (z : S(t),\ \texttt{Transfinite (Sup}(t,f),\ u,\ v,\ w,\ f(z)))$$

substituted for w.

For more on recursion in type theory see [Constable and Mendler 1985].

Defining natural numbers. We now give an example of the use of ordinal terms and transfinite induction we have just introduced. The type of natural numbers will not need to be a primitive in the TT system as they were in AHA. To define the natural numbers all we need as primitive types are **Void**, **Unit** and **Bool**. The type **Void** has no elements, the type **Unit** has the one element **It**, and the type **Bool** has the two elements **Tr** and **Fl**. To complete the construction of the data type **Bool**, we need an "if" statement. Just as recursion acts as the "elimination" rule for the type of natural numbers in AHA, the "if" statement is the "elimination" rule for the boolean type. The **If** construct **If**(b, exp_1, exp_2) evaluates as one would expect.

$$\texttt{If (Tr, } exp_1\texttt{, } exp_2\texttt{)} \longmapsto exp_1$$

$$\texttt{If (Fl, } exp_1\texttt{, } exp_2\texttt{)} \longmapsto exp_2$$

Corresponding to these two contractions are two computation rules: **TrueIfCompRule** and **FalseIfCompRule**. Neither of these rules hold any surprises. They look like this:

$$\texttt{TrueIfCompRule (If(Tr,}exp_1\texttt{,}exp_2\texttt{)): If(Tr,}exp_1\texttt{,}exp_2\texttt{)=}exp_1$$

$$\texttt{FalseIfCompRule (If(Fl,}exp_1\texttt{,}exp_2\texttt{)): If(Fl,}exp_1\texttt{,}exp_2\texttt{)=}exp_2$$

Now we can give the definition of the natural numbers, which is quite simple.

```
val x = Variable ("x", Bool);
val S(x) = If (x, Unit, Void);
val Nat = Ord (x, S(x));            { Def'n of Nat }
```

The first few natural numbers can now be defined. **Zero** is particularly tricky. Out of thin air we obtain an element in **Nat** for the definition of **Zero** using **VoidElim**.

```
val z = Variable ("z", S(Fl));
val exp1 = FalseIfCompRule (S(Fl));     { S(Fl)=Void    }
val exp2 = EqualElim (z, exp1);         { Void          }
val exp3 = VoidElim (exp2, Nat);        { Nat           }
val exp4 = ArrowIntro (z, exp3);        { S(Fl) => Nat  }
val Zero = Sup (Fl, exp4);
```

Next we define **One**.

```
val z = Variable ("z", S(Tr));          { S(Tr)=Unit }
val Pred0 = ArrowIntro (z, Zero);
val One = Sup (Tr, Pred0);
```

The other natural numbers are defined similarly.

```
val Pred1 = ArrowIntro (z, One);     { Var z declared above. }
val Two = Sup (Tr, Pred1);
```

Using transfinite induction we can define the successor function. We restate the rule as it is used in this particular case for reference.

$$\frac{n : Nat, \quad \begin{array}{c} u : Bool \\ v : S(u) \rightarrow Nat \\ w : \Pi y : S(u).Nat \\ \vdots \\ Sup(Tr, Next) \end{array}}{Nat}$$

In TT the successor function is defined as follows.

```
val n = Variable ("n", Nat);
val u = Variable ("u", Bool);
val v = Variable ("v", Arrow (S(u), Nat));
val y = Variable ("y", S(u));
val w = Variable ("w", Pi (y,Nat));
val z = Variable ("z", S(Tr));
val Next = ArrowIntro (z, Sup (u,v));          { S(Tr) => Nat  }
val exp = Sup (Tr, Next);                      { Nat }
val s = Transfinite (n, u,v,w, exp);           { Nat  }
val SuccFunc = ArrowIntro (n, s);              { Nat => Nat  }
```

It is worthwhile to consider an evaluation of successor function.

```
ArrowElim (SuccFunc, One)
ArrowElim (ArrowIntro (n, s), One)
[n/One]s
Transfinite (One, u,v,w, exp)
Transfinite (Sup (Tr, Pred0), u,v,w, exp)
[w/...]([v/Pred0]([u/Tr]exp))
Sup (Tr, ArrowIntro (z, Sup (Tr, Pred0)))
```

It is not apparent, but this is the desired answer: two. To see this we note that

`ArrowIntro (z, Sup (Tr, Pred0))`

has already occurred above, where we called it **Pred1**. The resulting expression **Sup(Tr,Pred1)** is exactly the definition we gave for two.

The monus function is more subtle.

```
val u = Variable ("u", Bool);
val v = Variable ("v", Arrow (S(u), Nat));
val w = Variable ("w", Pi (y, ArrowElim (v, y)))
        where val y = Variable ("y", S(u)) end;
val exp = If(u, ArrowElim (v, It), Sup (u,v));
val tr = Transfinite (n, u,v,w, exp);
```

Notice the difficulty that arises in attempting to type check the expression **exp**. The type of v is **Arrow (Unit,Nat)** only under the assumption that **u** is **Tr**. Notice, furthermore, that it is necessary to check that the types are *computationally* equal.

V3. We have motivated TT by showing how it results from collapsing some constructs in AHA. There are yet further simplifications that can be tried. One is the elimination of variables in favor of a complete set of primitive combinators like **K** and **S** [Stenlund 1972]. The big savings that occurs when variables are eliminated is that substitution is no longer needed. It is effectively replaced by the computation rules

$$\mathbf{K}xy = x, \quad \mathbf{S}fgx = g(x)(f(x)).$$

In this setting **Lambda** is eliminated, but not **Application**. Application is the only primitive operator that acts on combinators, and is needed (tacitly) to express the computation rules above.

$$\texttt{Application (Application (K, } exp_1\texttt{), } exp_2\texttt{)} \quad \longmapsto \quad exp_1$$

V3 uses no variables. It even goes further: all the constructs in the theory are combinators. To see what this means, we must reflect on the two ways we have used to extend programming logics. The first way was simple. New constants, propositions and primitive types could be added at will using **FunctionSymbol**. New constructs could be added to the language by writing the necessary ML functions to simulate them just as all the native constructs in AHA or TT were done. The theory TT suggests that these two ways may be similar. Consider the **First** construct in TT:

$$\texttt{First (}exp\texttt{: Cross (}A\texttt{, }B\texttt{)): }A$$

This construct is implemented directly in ML. Alternatively, we can consider this construct a function from elements of type **Cross**(A,B) to type A. The **First** construct could be defined as follows:

```
val FirstFunc = FunctionSymbol (Arrow (Cross (A, B), A));
```

```
val First (exp) = Application (FirstFunc, exp);
```

(The semantics of the rule would still have to be conveyed somehow to the interpreter.) Can all the constructs of the theory be defined this way?

The type theory V3 [Constable and Zlatin 1984] explores a theory of this nature. It is not immediately obvious how to implement such a theory. What is the type of the Π combinator? Whatever the type is, it involves another Π combinator, a more powerful one. Care must be taken to insure that type-checking ends.

4.4 Implementation of the propositional calculus

We have already seen how the propositional calculus can be implemented in the manner of PPLAMBDA using the abstract type mechanism in ML to insure that elements of the type are proofs in the logic. The preceding discussion equating types and formulas suggests that a language with a sufficiently rich type structure can be used to represent a logic. We shall see in this section that the type structure of ML is sufficiently rich to represent the propositional calculus, but not the predicate calculus, and we shall see that other languages have a type structure rich enough to represent the predicate calculus.

Another representation of the propositional calculus. We wish to define a map from formulas of the propositional calculus to types in ML and proofs to expressions. This map must be such that ML types have elements if, and only if, the corresponding formulas are true in the propositional logic. We start with the formula *True* which has the simple one line proof *Truth.* We assign the ML data type `unit` to *True* and the expression () to *Truth.*

```
- ();              { ()=Truth : True }
  () : unit
```

We can make this assignment explicit by defining a new type in ML.

```
- abstype True = MakeTrue of unit
  with val Truth = MakeTrue () end;
> abstype True
| val Truth : True
```

The functions **MakeTrue** maps the ML representation of the formula *True* to an ML data type we can associate with *True*. This extra step is not needed, and we can get the same effect more economically as follows:

```
- type True = Truth;
> type True = Truth
| con Truth : True
```

ML does not have a built-in type that can be identified with the formula *False* in first-order logic. This type must be such that it is impossible to build an element of it. One can construct such a type in ML, but it is obtuse.

```
- export abstype False
  from
        abstype False = ignore with val thistoo = () end
  end;
> abstype False
```

Here we have built an abstract type whose concrete representation is the enumerated type with one element **ignore**. The body of the abstract type (which cannot be null) is the simplest function which came to mind: **val thistoo = ()**. Neither this function nor the representation of the abstract type matter, because the only object exported from the abstract type definition is the type itself, thus assuring that there is no way to build an element of **False**.

True propositions can be identified with types like **True**, (i.e., single element enumerated types), and false propositions can be identified with types like **False**. For the sake of the following discussion we introduce some (true) propositions.

```
- type P = p and Q = q and R = r;
> type P = p
| con p : P
| type Q = q
| con q : Q
| type R = r
| con r : R
```

We have defined three enumerated types; each with a single element.

The "and" connective is identified with the built-in type operator for the cartesian product.

```
- (p,q);
  (p,q) : P * Q
- (Truth,r);
  (Truth,r) : True * Q
```

The built-in pairing mechanism of ML is used to construct elements of the "and" or "cross product" type. Notice that there is no way we can construct an element of type `False * Q`, because we have no element of type `False`.

ML does not have a built-in "union" type, which we need to identify with the "or" formulas. But one can easily be constructed.

```
- type ('a,'b) Union = OrIntroL of 'a | OrIntroR of 'b;
> type ('a,'b) Union = OrIntroL of 'a | OrIntroR of 'b
| con OrIntroL : 'a -> (('a,'b) Union)
| con OrIntroR : 'b -> (('a,'b) Union)

- val OrElim (x,f,g) =
    case x of
        OrIntroL y. f(y) |
        OrIntroR y. g(y);
  val OrElim : ((('a,'b) Union) * ('a -> 'c) * ('b -> 'c)) -> 'c
```

Implication introduction and elimination are modeled by function abstraction and function application.

```
- val f (a: P) = Truth;      { Implication introduction }
  f : P -> True
- f (p);                     { Implication elimination  }
  Truth : True
```

What we have seen so far is a correspondence between ML types and formulas in propositional logic. The correspondence looks like this:

`unit`	*True*
`False`	*False*
`'a * 'b`	$A \ \& \ B$
`'a -> 'b`	$A \Rightarrow B$
`('a,'b)Union`	$A \mid B$

Building elements of the ML types **unit**, **False**, **'a*'a**, **'a->'b** and **('a,'b)Union** corresponds to using the introduction rules and elimination rules.

Here are some general examples.

```
- val g (a: Q) = (r,a);
  g : Q -> (R * Q)

- g (p);                        { ML catches a type error. }
Type Clash
Looking  for :  Q
I have found :  P

- g (q);
  (r,q)  : R * Q

- val c(x:True):True*True = (x,Truth);
  val c : True -> (True * True)

- val d(x:P->P):P->P = x;
  val d : (P -> P) -> (P -> P)
```

The identity function for elements of type `P` is a proof of the proposition that $P \Rightarrow P$.

```
- val e(x:P):P = x;
  val e : P -> P
```

```
- d(e);
  fun : P -> P
```

Often we have to specify the type (as in the case of FalseElim in AHA) to prevent ML from choosing a polymorphic type, as we did in building all the elements above. If we do not specify a type, it is like quantifying over all propositions, or all types [Reynolds 1984].

```
- val c (x) = (x,Truth);
  val c : 'a -> ('a * True)
```

As the examples above indicate, ML has a deductive mechanism more powerful than that of a Hilbert-style axiom system with K and S axioms and the single rule of inference, *modus ponens*. The expression **e** above is a proof that $P \Rightarrow P$. In the example of the propositional calculus given at the beginning of this chapter we needed a five-step proof of that fact. It is perhaps instructive to mimic that proof given earlier. The proof is true for all propositions, whether or not the proposition has an element, but to be concrete we give the proof for P.

```
- val exp1 (a:P) (b:P) = a;
  val exp1 : P -> (P -> P)

- val exp2 (f:'a->(P->P)) (g:'a->('b->P)) (a:P) = (g(a) o f)(a);
  val exp2 : (P -> (P->P)) -> ((P -> ((P->P) -> P)) -> (P->P))

- val exp3 = exp2 (exp1);
  val exp3 : (P -> ((P -> P) -> P)) -> (P -> P)

- val exp4 (a:P) (b:P->P) = a;
  val exp4 : P -> ((P -> P) -> P)

- val exp5 = exp3 (exp4);
  val exp5 : P -> P
```

The expression **exp5** is a proof of $P \Rightarrow P$.

In this above proof we have limited the proof techniques to two. Those are the rules of the axiom game. We can code these techniques in ML.

```
- val K a b = a;
  val K : 'a -> ('b -> 'a)

- val S f g a = (g(a) o f)(a);
  val S : ('a -> 'b) -> (('a -> ('b -> 'c)) -> ('a -> 'c))
```

Now we can repeat the proof with the functions K and S, and perhaps the correspondence with the proof at the beginning of the chapter will be even clearer.

```
- val exp1 = K: P->(P->P);
  val exp1 : P -> (P -> P)

- val exp2 = S;
  val exp2 : ('a -> 'b) -> (('a -> ('b -> 'c)) -> ('a -> 'c))

- val exp3 = exp2 (exp1);
  val exp3 : (P -> ((P -> P) -> 'a)) -> (P -> 'a)

- val exp4 = K: P->((P->P)->P);
  val exp4 : P -> ((P -> P) -> P)

- val exp5 = exp3 (exp4);
  val exp5 : P -> P
```

The amount of real work being done is obscured by the explicit mention of the types. The whole proof can be reduced to one line with three expressions (three uses of the axioms K and S) and two implicit applications (two uses of *modus ponens*).

```
- val exp5 = (S(K:P->(P->P)))(K);
  val exp5 : P -> P
```

Notice that this corresponds to the equivalence

$$\mathbf{I} \equiv \mathbf{SKK}$$

in the theory of combinators [Stenlund 1972]

This representation of propositional logic in ML does not extend to predicate logic. There is no way to represent a predicate in ML at all. What is needed is a type definition as follows:

```
type P(n:int) =
 if n=0 then int else bool;
```

Then we still need a dependent arrow type to represent universal quantification.

Programming languages with a type structure rich enough to embed the predicate logic are not in general use. But we mention two languages that have rich enough type structures. In RUSSELL [Demers and Donahue 1980] a dependent arrow type would have signature `func [x: A] B`. B is a signature (type) expression with x free. In PEBBLE [Burstall and Lampson 1984] there is a primitive type operator for the dependent arrow. The correspondence between PEBBLE and predicate logic looks like:

`void`	*True*
	False
`'a x 'b`	$A \ \& \ B$
`'a -> 'b`	$A \ \Rightarrow \ B$
	$A \ \mid \ B$
`x:'a ->> 'b`	$All\, x : A .\ B$
`x:'a xx 'b`	$Some\, x : A .\ B$

4.5 Intuitionistic set theory (IZF)

We have seen two programming logics thus far: AHA and TT. In this section we briefly present a third one in order to emphasize the diversity of theories that can be simulated on the logic engine.

This third programming logic is based on the axiomatization of set theory begun by Zermelo and modified by Fraenkel. Classical Zermelo-Fraenkel set theory would not prove a wholly satisfactory formalization. Even if nonconstructive reasoning were eliminated (that is, if the axiom of excluded middle were banned) from the logical apparatus, the possibility remains that the axioms pertaining to sets may somehow let nonconstructive reasoning creep back in. In fact the axiom of choice does imply the law of excluded middle [Goodman and Myhill 1975, McCarty 1984]. So, care is required in the axiomatizing of set theory. Fortunately, this has been the subject of some study under the classification of intuitionistic set theory. Hence, we name the third and final programming logic IZF.

The terms in IZF will all have the type **Set**. This means we have **Set** as the only base or primitive type. In IZF there are two set constants: **Empty** for the empty set (usually denoted by $\emptyset$ in mathematics), and **Omega** for the one infinite set assumed to exist in the world of sets (usually denoted by ω in mathematics). Roughly analogous to the primitive term constructors **Abstract** and **Application** in AHA, there are several set constructors in IZF. These are **Union**, **Power**, **Pair**, **Sep**, and **Coll**.

For formulas we have the ones we expect. Atomic formulas are either equalities or constructed using the set membership relation **Epsilon**.

```
val S, T = SetVariable ("S"), SetVariable ("T");
val eq = Equal (S,T);        { Atomic formula:  S=T    }
val in = Epsilon (S,T);      { Atomic formula:  S in T }
```

Atomic formulas are combined using the usual connectives.

To reason about sets we add all the introduction and elimination rules for the logical connectives and for the quantifiers as were introduced for AHA. Bear in mind that quantification is now restricted to sets and sets alone, in particular, there are no functions.

Before we can start presenting the rules particular to IZF, we must introduce some abbreviations.

```
val AllIn (x,S,P)  = All (x, Implies (Epsilon (x,S), P));
val SomeIn (x,S,P) = Some (x, And (Epsilon (x, S), P));
val Subset (S,T) =
  AllIn (x, S, Epsilon (x,T)))
    where val x = NewSetVariable ();
```

The first axiom in the formulation of Zermelo-Fraenkel set theory is the axiom of extensionality. We formulate this axiom as a rule of inference as follows:

$$\frac{S \subset T, \quad T \subset S}{S = T}$$

As usual this rule of inference is translated into a constructor of proof expressions. This constructor, called **Extensionality** takes the following form:

Extensionality $(prex_1 :$ **Subset**(S,T), $prex_2 :$ **Subset** $(T,S))$: **Equal**(S,T)

The next axiom in ZF concerns the existence of unordered pairs of sets. Typically this axiom is expressed in the following form:

$$All\ x,y\ .\ Some\ p\ .\ All\ z\ .\ (z \in p \iff z = x \mid z = y)$$

Skolemizing this formation of the axiom replaces p with the set $\{x,y\}$ or, in IZF, **Pair(x,y)** and allows the existential quantifier to be dropped. We permit the axiom to be instantiated with any sets S, T and U, so therefore the universal quantifiers can be ignored. Then we break up the axiom into two parts. One views the axiom as describing when it can be proved that a set is in the pair. The other views the axiom as describing what is known about a set if it is in the pair. This leads to the following formulation:

$$\frac{S = T \mid S = U}{S \in \{T,U\}} \qquad \frac{S \in \{T,U\}}{S = T \mid S = U}$$

We call the rules **PairIntro** and **PairElim** in IZF.

$$\texttt{PairIntro}\ (\mathit{prex}:\ \texttt{Or}\ (S = T,\ S = U)):\ \texttt{Epsilon}(S,\texttt{Pair}(T,U))$$

$$\texttt{PairElim}\ (\mathit{prex}:\ \texttt{Epsilon}(S,\texttt{Pair}(T,U))):\ \texttt{Or}\ (S = T,\ S = U)$$

These rules cancel each other out and the interpreter reflects this fact:

$$\texttt{PairElim}\ (\texttt{PairIntro}\ (\mathit{prex})) \longmapsto \mathit{prex}$$

The complete set of reduction rules can be found in the appendix.

The rules for the formation of the infinite union and the power set are derived similarly. Here is the natural deduction style formulation of union axiom:

$$\frac{Some\ x \in S\ .\ T \in x}{T \in \bigcup(S)} \qquad \frac{T \in \bigcup(S)}{Some\ x \in S\ .\ T \in x}$$

For the power set we have the following:

$$\frac{S \subset T}{S \in \mathcal{P}(T)} \qquad \frac{S \in \mathcal{P}(T)}{S \subset T}$$

The formation of the IZF constructors **UnionIntro**, **UnionElim**, **PowerIntro** and **PowerElim** is obvious.

An important axiom in Zermelo-Fraenkel set theory is the axiom of separation. Like the pair, union and power set axioms, this axiom can be brought into a convenient form. This is what it looks like:

$$\frac{S \in T,\quad P(S)}{S \in \{x \in T \mid P(x)\}} \qquad \frac{S \in \{x \in T \mid P(x)\}}{S \in T\ \&\ P(S)}$$

There are, however, some minor complications. First, there is a problem in determining the form of the "separating" proposition P given an instance $P(S)$. This is solved by requiring it to be given explicitly as an argument (like in the some-introduction rule) in the **SepIntro** rule. Here is the rule:

$$\texttt{SepIntro}\ (x,\ P(x),\ prex_1 : S \in T,\ prex_2 : P(S)) :\ S \in \{x \in T \mid P(x)\}$$

The first two arguments describe the separating proposition. The variable x is bound in $P(x)$. This is the form of the elimination rule:

$$\texttt{SepElim}\ (prex : S \in \{x \in T \mid P(x)\}) :\ \texttt{And}\ (S \in T,\ P(S))$$

Another problem is the form of the term constructor for separation. It is different in this case than for the power set or union, because separation takes place with respect to some predicate P and not just other sets. One solution would be to take advantage of ML and define an ML function from terms to formulas. This function would be an argument to the separation set constructor. This seems to coincide well with the intuitive meaning of $P(x)$. The problem with this approach is that equality of ML functions is impossible to determine. The solution is to require the predicate to be passed as an argument to the separation set constructor in form of a predicate with one free variable. Thus where we have written $\{x \in T \mid P(x)\}$ in the rules above, we have meant the use of the IZF **Sep** set constructor which takes on the following form:

$$\texttt{Sep}\ (T\texttt{: Set},\ x,\ P(x))\texttt{: Set}$$

If the free variable in the predicate is known, a function is effectively obtained with the help of substitution. Thus if we need to know $P(S)$, it can be obtained by $[x/S]P$. Alpha-equality insures that

$$\texttt{Sep}\ (T\texttt{: Set},\ x,\ P(x))\texttt{: Set}$$

and

$$\texttt{Sep}\ (T\texttt{: Set},\ y,\ P(y))\texttt{: Set}$$

are the same set.

The rule for induction is as follows:

$$\frac{All\ x\,.\ \left((All\ y \in x\,.\ P(y))\ \Rightarrow\ P(x)\right)}{All\ x\,.\ P(x)}$$

The noteworthy thing about this rule is the reduction.

`AllElim (Induction (`$prex$`),` S`: Set)`

$\longmapsto$ `ImpElim (AllElim (`$prex, S$`), AllIntro (`y

`ImpIntro (`$a : y \in S$`, AllElim (Induction (`$prex$`),`y`))))`

This reduction can only apply when the set S is *not* a variable. Otherwise this reduction will not end.

There are three axioms concerning infinity and the empty set. They are:

$$\emptyset \in \omega \qquad All\ x\ .\ x \notin \emptyset \qquad All\ x \in \omega\ .\ x' \in \omega$$

Here in this section on set theory, the successor operator (denoted x') is the usual function: $x' = x \cup \{x\}$. The preceding definition requires the definition of the union of two sets ($S \cup T = \bigcup\{S, T\}$) and of the singleton set ($\{S\} = \{S, S\}$). By now the translation from the more easily comprehended natural-deduction style proof rules to the corresponding constructors in the programming logic, should have become routine. But we take advantage of the simple nature of these last axioms to belabor yet another point about the formulation of constructors in a programming logic. The proof that a set is not in the empty set is a constructor of the following form:

`EmptySet (`S`: Set): Not (`$S \in \emptyset$`)`

This constructor takes one argument, a set, and the axiom is instantiated for that set. One might have expected an alternate formulation with no arguments:

`EmptySet': All (`x`, Not (`$x \in \emptyset$`))`

What is the difference? Consider `AllElim (EmptySet', `S`)` and

`AllElim (AllIntro (`x`, EmptySet (`x`)), `S`)`

This first one is not reducible. Even if there were a special clause in `AllElim` there is no proof expression to reduce to. The second one reduces to `EmptySet (`S`)`.

The last rule making up IZF is the rule of collection. Without an additional method for building new sets it is not possible to define the set $\{\omega, \omega + 1, \omega + 2, \ldots\} = \omega 2$. Here is one formulation of it [McCarty 1984]:

$$All\ x\ .\ \big(All\ y \in x\ .\ Some\ z\ .\ P(y, z)\ \Rightarrow\ Some\ C\ .\ All\ y \in x\ .\ Some\ z \in C\ .\ P(y, z)\big)$$

Modified so that the set C becomes a set constructor, it looks like:

$$\frac{y \in x\ \Rightarrow\ Some\ z\ .\ P(y, z)}{All\ y \in x\ .\ Some\ z \in C(x)\ .\ P(y, z)}$$

Further transformed the rule becomes:

$$\frac{S \in T, \quad \begin{array}{c} y \in x \\ \vdots \\ Some\ z\ .\ P(y,z) \end{array}}{Some\ z \in C(T)\ .\ P(S,z)}$$

In the above rule, S and T stand for any sets; that is, any terms of type **Set** including variables. The rule of collection can be instantiated with any terms of type **Set** in the place of S and T. On the other hand, the free variables x and y in the subproof are to be IZF variables; arbitrary terms cannot be used in their place.

On the whole, this formulation is not satisfactory. Although we have been writing $C(T)$, this set constructor depends not only on the proposition $P(y,z)$, but also on the proof. Furthermore, the rule is complicated and does not fit the pattern of introduction and elimination rules that the other set constructors have.

A more satisfactory solution is inspired by the language SETL [Kennedy and Schwartz 1975]. In SETL there is a set constructor which has the general form

$$\{E(x), x \in S \mid P(x)\}$$

where $E(x)$ is an expression possibly containing the variable x free. (Separation is a special case: $\{x, x \in S \mid P(x)\}$.) In IZF we write this as follows:

$$\texttt{Coll}\ (S\texttt{: Set, } x\texttt{, } E(x)\texttt{: Set, } P(x)\texttt{)}$$

The set $\omega 2 = \{\omega + x, x \in \omega \mid True\}$ is represented:

$$\texttt{Coll (Omega, } x\texttt{, Omega+}x\texttt{, True)}$$

(Of course, we have not defined what "+" on the ordinals is.)

To prove a set is in a collection we have the following rule:

$$\frac{S \in T, \quad P(S)}{E(S) \in \{E(x), x \in T \mid P(x)\}}$$

This introduction rule requires that $E(x)$ and $P(x)$ be given. The elimination rule is:

$$\frac{U \in \{E(x), x \in T \mid P(x)\}}{Some\ z \in T\ .\ P(z)\ \&\ U = E(z)}$$

In the formulation of the rule above the **SomeElim** rule must be modified to account for this new way to introduce a "some" formula. On the other hand, the **SomeElim** rule can be used as a model so that a completely independent construct for the collection elimination rule can be formulated.

$$\frac{U \in \{E(x), x \in T \mid P(x)\}, \qquad \begin{matrix} z \in T \\ P(z) \\ U = E(x) \\ \vdots \\ Q \end{matrix}}{Q}$$

These rules are derivable from McCarty's formulation of the collection axiom.

Throughout this presentation of rules for IZF we have conformed to the usual mathematical practice of writing

$$All\ x \in S\ .\ P(x) \text{ for } All\ x\ .\ (x \in S \Rightarrow P(x)$$

and

$$Some\ x \in S\ .\ P(x) \text{ for } Some\ x\ .\ (x \in S \ \&\ P(x)).$$

Not only is it possible to use ML to define such generalized quantifiers as we have done and define generalized rules of inference to use the formulas without decomposing them, but the logic engine also has the facility to permit print abbreviations, so that the user does not have to look at expressions with all the definitions unwound. A print abbreviation is a pair of functions. One function recognizes the abbreviation, and the other creates the print representation of it.

```
- val IsSomeIn (phi: Formula): bool =
  ...
  val IsSomeIn = fun : Formula -> bool

- val PrintSomeIn (phi:  Formula): string =
  ...
  val PrintSomeIn = fun : Formula -> string
```

A pair of functions is entered to the system using the **AddPrintAbbrev** command.

Chapter V

Applied Metamathematics

We have seen ML used as the meta-language of programming logics in several guises. The most simple use has been in forming abbreviations. This was a convenience in writing down long proof expressions. We have hinted at other uses for ML. For the propositional logic we suggested a theorem prover. In the proof of Cantor's theorem we suggested an intensional analysis of proof expressions that automatically yields counter-examples.

More useful metamathematics can be applied to the task of proof development. The key factor in taking advantage of this metamathematics is ML. Metamathematics of programming logics is possible because proof expressions are objects in ML.

In this chapter we will use ML to implement top-down development of proofs. We will then be able to dispense with the tedious bottom-up proof building strategy. In the second section we will see how to integrate the fruits of research toward automatic theorem proving with building proofs in programming logics. Finally we describe an ML program that transforms some proofs using classical methods to constructive proofs. This permits classical methods to be used to develop some proof expressions in AHA.

5.1 Tactics

ML functions can be written to assist in the development of proofs. Already some key ideas about proof development have emerged. We see some of the necessary ingredients first in the block structured proof in the PL/CV [Constable et al. 1982, Johnson 1980] and AVID [Krafft 1981] verifiers. The AVID system demonstrated the ease and naturalness of developing proofs top-down. Refinement logics [Bates 1979] are designed so that proofs are developed top-down. Instead of proof trees with the "branches" on top and the "root" or goal on the bottom, refinement logics develop proofs starting with the goal and then the "branches" are filled out. In this section we show how ML can be use to simulate a system with top-down development, as a significant aid in proof development. We will define a refinement rule and a proof tree.

Although it helps to develop proofs top-down, this does not diminish the tedium of completing proofs in every detail to the satisfaction of a proof verifier. For this reason we demonstrate how tactics [Gordon 1979, Constable et al. 1984] can be used to develop proofs in programming logics.

Block structure. We have written proof expressions in a non-structured, bottom-up style up to now, so that we could discuss constructing them from their component parts. Now it is convenient to adopt another notation which we will call block-structured. We introduce this notation for three reasons. One, the block-structured notation is better for viewing proof expressions as a whole. It even resembles (remotely) proofs done by hand. Two, the translation from proof expressions in block structure to the proofs of the PL/CV and AVID verifiers [Constable et al. 1982, Johnson 1980, Krafft 1981] is made obvious. Three, block-structured proofs suggest a top-down development of these proofs which more closely resembles how human mathematicians approach these proofs.

The basic idea behind block structuring is to linearize the proof expression tree by indenting the arguments of each construct. We occasionally add a few words of "syntactic sugar" to make the result more legible. We illustrate the new notation with the **AndIntro** construct.

AndIntro
  $prex_1$: A
  $prex_2$: B

Alternatively, we may want to write:

A & B by **AndIntro**
  $prex_1$: A
  $prex_2$: B

This serves to emphasize or recall the formula which is being proved.

We list now some of the other constructs in AHA in block-structured form to make the syntax more familiar.

AllIntro
  Let $x : T$ be arbitrary

$prex : P(x)$

Next is the implication-introduction construct.

```
ImpIntro
  Assume hyp : A
  prex : B
```

The arguments to the induction construct are named to recall their purpose.

```
Induction
  To show : All .m : Nat P(n)
  Base case
    prex1 : P(0)
  Induction variable n
  Induction hypothesis
    ihyp : P(n)
  Induction step
    prex2 : P(n')
```

Finally, we give the some-introduction and some-elimination constructs.

```
SomeElim
  prex1 : Some .x P(x)
  Choose x0 where shyp : P(x0)
  prex2 : Q

SomeIntro
  To show : Some .x P(x)
  With t for x
  prex : P(t)
```

Finally we have the or-elimination construct, which we have renamed "cases" following the example of PL/CV, since a proof "by Cases" reads better than "by OrElim."

C **by Cases**
  $prex_1 : A \;\; | \;\; B$
  $prex_2 : A \;\; \Rightarrow \;\; C$
  $prex_3 : B \;\; \Rightarrow \;\; C$

These examples, among which are all the complicated constructs, should be enough to demonstrate the basic idea. The other constructs when written in block structure are easily recognized.

We consider an example from PL/CV [Constable et al. 1982]. In the following proof the predicate G is read "Mr. Jones is Governor of New York," ITH "Mr. Jones is from Ithaca," and NYC "Mr. Jones is from New York City."

```
*PLCV
*THEOREM
  /*/  ALL (G, ITH, NYC) BIT WHERE (G => (ITH|NYC) & ~NYC .
                              (~ITH => ~G) BY INTRO,
       PROOF;
         ARBITRARY (G, ITH, NYC) BIT WHERE (G => (ITH|NYC)) & ~NYC;
         ~ITH => ~G BY INTRO
         PROOF;
           ASSUME ~ITH;
           ~G BY INTRO,
           PROOF;
           ASSUME G;
           '0' BY CASES, ITH|NYC
             PROOF
             CASE ITH;
                ~ITH;
             CASE NYC;
                ~NYC
             QED;
           QED;
         QED;
       QED;                    */
```

In PL/CV the blocks delimited by the PROOF—QED pairs indicate the scoping rules for assumptions and variables. All assumptions in the current and surrounding blocks are "visible." This suggests a mechanism for building proof expressions without discharged assumptions: keep track of the "visible" assumptions and only those visible ones can be proved by assumption. This can only be done in a top-down approach where the "declaration" (ASSUME) of a hypothesis comes before its use in the development of a proof. The declaration of a variable or an assumption in a proof block corresponds to what we have called a binding in a programming logic.

The assertion ~ITH in the sub-proof by cases is allowed because it is in the pool of visible assertions. The assertion ~NYC is deducible since it is a conjunct of a visible assertion. We see that certain trivial deductions can be performed automatically by the PL/CV system without explicit justification. This is also the case for the assertion ITH|NYC which can be deduced from visible assertions by an application of implication elimination. The mechanism that supports this reasoning is the assertion table [Johnson 1980].

Here is the proof in AHA. The proof is longer since we must fill in all the details.

```
AllIntro
  Let G, ITH, NYC be arbitrary
  G => (ITH|NYC) & ~NYC  =>  (~ITH => ~G) by ImpIntro
    Assume "hyp1" : G => (ITH|NYC) & ~NYC
    Assume "hyp2" : ~ITH
    ~G by NotIntro
      Assume "nhyp" : G
      False by FalseIntro
        Cases
          ITH|NYC by ImpElim
            G => (ITH|NYC) by AndElimR
                Assumption "hyp1"
            G by Assumption "nhyp"
          ITH => False by ImpIntro
            Assume "case1" : ITH
              False by FalseIntro
                ITH & ~ITH by AndIntro
                  Assumption "case1"
                  Assumption  "hyp2"
          NYC => False by ImpIntro
            Assume "case2" : NYC
              False by FalseIntro
                NYC & ~NYC by AndIntro
                  Assumption "case2"
                  ~NYC by AndElimL
```

```
                    Assumption "hyp1"
QED
```

The AVID system took the PL/CV logic and provided a structured editor for building proofs from the goal and filling in the incompleted arguments interactively. At the user's command the system provides a template to be filled in. This saves remembering the syntax (which for PL/C is fairly baroque), and eliminates many key strokes. We would like such convenience for programming logics. In addition we would like the convenience of having trivial deductions made automatically. We can get both by introducing refinement rules.

Refinement. The idea is to refine a goal into subgoals. A logic built this way is a "refinement logic," a phrase coined by Bates [Bates 1979]. In a natural-deduction style calculus a rule of inference would have premises and conclusions. One would build proofs bottom-up, from premises to conclusions. In a refinement logic, however, proof building is turned the other way around. The goal corresponds to the conclusion and the generated subgoals to the premises. Proof building is a top-down process in a refinement logic.

Toward this end we define an ML type of functions that take a goal and produce a list of subgoals and a function (called a validation [Gordon et al. 1979, Constable et al. 1984]) to reach the goal from the subgoals.

```
abstype RefinementRule =
  RR of Formula -> (Formula list * ((ProofExp list) -> ProofExp))
with
  { Constructor for a refinement rule. }
    val RefinementRule f = RR f;

  { Given refinement rule and formula, get list of subgoals. }
    val Subgoals (RR f, phi) = sgs where val (sgs,_)=f(phi) end;

  {  Given a refinment rule, a formula and a list of sub proofs
     (of the subgoals), form a proof of the formula.          }
    val Proof (RR f, phi, sps) = g (sps)
                               where val (_,g) = f(phi) end
end;
```

After each application of a refinement rule we must know how to build a proof of the goal from the proofs of the subgoals. This is the responsibility of the function (ProofExp list)->ProofExp given with every refinement rule. The list of proof expressions can be assumed to prove the subgoals. Some way of constructing the goal must be known.

This is best clarified by an example of a refinement rule.

```
val AndRefinement = RefinementRule f
  where val f exp = (l,g)
    where val l = [FirstConj exp; SecondConj exp];
          val g sps = AndIntro (Subgoal (1,sps),Subgoal (2,sps))
    end
  end;
```

The function g produces a proof of the goal (a conjunction) using AndIntro and the two proof expressions proving the subgoals.

For every rule in AHA there is a corresponding refinement rule. For axioms there are refinement rules that make no subgoals, since axioms have no premises.

To facilitate top-down development of proofs we introduce another ML abstract data type. This ML type is the proof tree. Some structure is necessary to carry the incompleted proof while it is being developed. A proof tree is not a proof until all the subgoals have been refined and none are left. (This abstract data type, and RefinementRule as well, are simplifications of the types actually used, in that all the error checking and handling has been removed.)

```
abstype ProofTree =
  PT of (Formula list) * ((ProofExp list) -> ProofExp)
with
  { Given a formula to proof and a refinement rule,
    get the top of the proof tree.                      }
    val ProofBy (phi, rr) = PT (l,f)
        where val l = Subgoals (rr, phi);
              val f (sgs) = ProofExp (rr, phi, sgs)
        end;

  { Refine subgoal i of a proof tree. }
    val RefineBy (PT (sgs,f), rr, i) = ...
```

```
    { Given a proof tree with no subgoals, form the proof
      expression proving the top goal.                    }
      val FormProofExp (PT (sgs,f)) = f([])
  end
```

A proof of A starts with ProofBy(A,rr), where rr is the first refinement rule. Any subgoal of the resulting proof tree can be refined using RefineBy.

One must not confuse the user-friendly proofs—the block-structured proofs and the proof trees—with the underlying proof expressions they help represent. The representation we have chosen for proof expressions has allowed the clear expression of many ideas and will continue to do so.

These refinement rules do not enforce the laws of proof of the logic. Nothing prevents a user from refining $A \mid B$ using AndRefinement. Chaos may result as selectors fail to find the constructs they expect, but more importantly there is no possibility of creating an invalid proof, because the function FormProofExp has the job of creating a proof expression (an element of type theorem) and this is not possible unless all the laws of the logic are followed.

We do not pursue refinement logics here, but we wish to consider a generalized notion of refinement.

Tactics. Refinements rules are the smallest increment of progress possible in proof development. We wish to combine them into to more sophisticated proof strategies, called tactics. One way to accomplish this is to define other functions of type

```
Formula -> (Formula list * ((ProofExp list) -> ProofExp))
```

that encompass more than one inference rule at a time. There is no limit to the complexity of these functions—any theorem prover imaginable can be converted to a function of this type. However, a disciplined way is needed to build these tactics up to avoid rigid, one shot tactics. This is possible if we insist that all tactics signal when they are not applicable. One means of doing this is to use the escape mechanism of ML. (In fact, this mechanism is part of the ML programming language for this very application in defining tactics.)

We start with the refinement rules as the first primitive tactics. For example, here is the tactic for conjunctions.

```
  val AndTactic exp =
    (if IsAnd exp then (l,g) else escape "No")
```

```
where val l = [FirstConj exp; SecondConj exp];
      val g sps = AndIntro (Subgoal (1,sps),Subgoal (2,sps))
end;
```

If this tactic does not apply, it signals failure.

The failure mechanism can be used to good advantage in trying one tactic then another.

```
val OR (T1, T2) =
  where val f exp = T1 (exp) ? T2 (exp)
```

The function **OR** is a useful means of building new tactics. Such a function is called a *tactical* (in analogy to functional) in LCF. LCF has several tacticals predefined, such as **THEN** and **REPEAT**.

For an extensive account of tactics see [Constable et al. 1984].

5.2 The connection with PROLOG

It is worthwhile to see the relation between a programming logic and another approach which uses logic as a programming language. This other approach uses a restricted set of formulas of first-order logic to codify the "facts" pertaining to a program. The system automatically deduces certain existential formulas and can provide the user with witnesses. This is the idea behind PROLOG [Clocksin and Mellish 1981].

We will briefly consider the example of the append function for lists. We know that this function is characterized by the following two equations:

$$[]@x = x, \qquad (x :: y)@z = x :: (y@z)$$

where "@" denotes the append function, "[]" the empty list, and "::" the "cons" or list construction function. In these equations x, y and z stand for any lists whatever. Here is a program for **Append** in pidgin PROLOG:

```
Append ([],x,x) .
Append (x::y,z,x::w) if Append (y,z,w) .
```

The program has two clauses and is written in relational form with the result of the append function appearing as the third argument. The first line of the program is an assertion that appending the empty list does not change the second argument. The second line is a conditional clause that requires a recursive call to **Append**. To invoke the append function, we will consider a query primitive **ONE** for PROLOG that finds the first (if any) solution for a variable consistent with the given clauses. Here **ONE** is used to append two lists together.

```
- ONE {x : Append ([a;b;c], [d;e], x)} ?
> [a;b;c;d;e]
```

Other types of queries can be made.

```
- ONE {x : Append ([a;b], x, [a;b;c])} ?
> [c]
- ONE {x : Append (x, [c;d], [a;b;c])} ?
> No solutions
```

The clauses in question are Horn clauses; that is, clauses in the form A **if** B_1, ..., B_n or $B_1 \& \ldots \& B_n \Rightarrow A$ [Clocksin and Mellish 1981, Bundy 1984]. PROLOG computes the solution using linear input resolution, an algorithm that unifies terms and backtracks when an alternative fails. This same approach can be used by an ML program to prove formulas of the form $C_1 \& \ldots \& C_n \Rightarrow$ $Some\,x.P(x)$. With additional effort the program could be made into a tactic; if the program cannot find a solution that tactic would fail, otherwise the proof found could be inserted. The remainder of this section shows how to simulate PROLOG in AHA.

First we consider a simple PROLOG program and a query.

```
R(a,b) .
R(f(x),g(x)) .
Q(x,y) if R(x,g(y)) .
ONE {z : Q(f(z),a)} ?
```

The answer to the query is the constant a, since $R(f(a), g(a))$ implies $Q(f(a), a)$.

We shall translate this example program to AHA. First we need to establish the vocabulary. We have constants $\{a, b\}$, function symbols $\{f, g\}$, relation symbols $\{R, Q\}$ and variables $\{x, y, z\}$. Since AHA is typed we will introduce a new primitive data type **Pan** for the PROLOG universe.

```
val Pan = PrimType ("Pan");
val a, b = FuncSymbol ("a", Pan), FuncSymbol ("b", Pan);
val f = FuncSymbol ("f", Arrow (Pan, Pan));
val g = FuncSymbol ("g", Arrow (Pan, Pan));
val R = RelationSymbol ("R", Cross (Pan, Pan));
val Q = RelationSymbol ("Q", Cross (Pan, Pan));

val x = Variable ("x", Pan);
val y = Variable ("y", Pan);
val z = Variable ("z", Pan);
```

Next we define some ML functions to build the necessary terms.

```
val f (x) = Application (fFunc, x);
val g (x) = Application (gFunc, x);
val R (x,y) = Application (RRel, Pair (x,y));
val Q (x,y) = Application (QRel, Pair (x,y));
```

The three clauses of the PROLOG program translate to three formulas in first-order logic (the universal closures one would expect), and, hence, directly into three AHA formulas.

```
val cl1 = R (a,b);
val cl2 = All (x, R (f(x), g(x)));
val cl3 = All (x, All (y, Imply (R (x,g(y)), Q (x,y))));
```

The PROLOG query translates to the request to prove the formula `So` from the hypotheses `Hyp` as defined below.

```
val Hyp = nAnd [cl1; cl2; cl3];
val So = Some (z, Application (Q, Pair (Application (f,z), a)));
```

It is fairly clear how one can derive `So` from `Hyp`.

```
val hyp = Assume ("hyp", Hyp);
val hyp1 = nAndElim1 (hyp);          { Clause 1.  (Not needed.)     }
```

```
val hyp2 = nAndElim2 (hyp);          { Clause 2.  R(f(x),g(x)       }
val hyp3 = nAndElim3 (hyp);          { Clause 3.  R(x,g(y))=>Q(x,y) }

val prex1 = AllElim (hyp2, a);               { R (f(a),g(a))        }
val prex2 = AllElim (AllElim (hyp3,fa), a); {R(f(a),g(a))=>Q(f(a),a)}
val prex3 = ImpIntro (prex2, prex1);
val prex4 = SomeIntro (So, prex3, a);
```

A tactic that would have found this proof might work as follows. First, it assembles the givens. From the first clause we know that if we need to prove $R(a,b)$, we can do so by assumption. From the second clause we know that if we need to prove $R(f(t),g(t))$ for some term t, we can prove it by all-elimination. From the third clause we know that if we need to prove $Q(t,s)$ for any terms t and s, it is possible if we can prove $R(t,g(s))$. The first goal is to prove $Q(f(z),a)$. So we resolve $Q(f(z),a)$ against $R(a,b)$, $R(f(x),g(y))$ and $Q(x,y)$. Only $Q(x,y)$ succeeds with substitutions $z \rightarrow z$ and $x \rightarrow f(z), y \rightarrow a$ and since $Q(x,y)$ is part of a conditional clause, the next goal is $R(f(z),g(a))$. Resolving again against the three clauses succeeds only with $R(f(x),g(x))$ of clause two, with substitutions $z \rightarrow a$ and $x \rightarrow a$. Since this is not a conditional clause no further subgoals are generated, and a proof has been found.

The proof expression can now be built up knowing the substitutions required and the clauses used in the successful resolutions. The proof starts with `AllElim (hyp2, a)` since the last resolution succeeded against clause two with substitution $x \rightarrow a$. We also know that a is the term to be eventually used as the witness in the some-introduction for z. (If the substitution had been $z \rightarrow z$, then we could pick any term for z.)

```
val prex1 = AllElim (hyp2, a);
val prex2 = AllElim (AllElim (hyp3, fa), a);
```

The resolution against clause three required $x \rightarrow f(z)$ or, since a has been chosen for z, $x \rightarrow f(a)$ and $y \rightarrow a$. Because clause three is a conditional clause, the proof expression is built with implication-elimination.

```
val prex3 = ImpElim (prex2, prex1);
```

There are no more resolutions, so all that is left is the some-introduction.

```
val prex4 = SomeIntro (So, prex3, a);
```

This is the proof expression desired. It is clear that the method is general enough to handle all such clauses (implicitly universally quantified) and can easily be generalized to more complicated conditional clauses. Unification algorithms exist for typed terms as well [Huet 1975], so this method can be generalized in this direction.

It is interesting to note that one can still compute the solution to the query (by using the axiom of choice) without discharging the hypothesis, as with the proof expression **prex4** above. This suggests the possibility that one can program in a programming logic without proving all obligations, if the computational content in no way depends on these missing pieces. Of course, there is no guarantee that the program is correct, but this gives rise to the possibility of a compromise with the practical expediencies of the real world.

There are some pitfalls with writing tactics. For instance, tactics do not necessarily terminate. On the following input, the PROLOG program sketched above does not terminate.

```
R(b,a) .
R(x,y) if R(y,x) .
ONE { z : R(a,z) } ?
```

This query will terminate depending on the order in which the clauses are chosen to do the resolution. The conditional clause could always be chosen successfully leading to permuting the arguments endlessly.

Resolution is not the only theorem proving technique that one might want to employ as a strategy for developing proofs. Decision procedures like ARITH [Constable et al. 1982] are useful for dispensing with the details of integer arithmetic. The congruence closure algorithm [Nelson and Oppen 1980] is useful for reasoning about equality with uninterpreted function symbols. Notice that in the case of certifying that an equality holds, the proof of the equality from the axioms—reflexivity, symmetry and transitivity—has no computational content and could be eliminated once the equality is established. [Bates 1979]. Practical proof development systems like PRL make use of decision procedures to eliminate details automatically [Harper].

5.3 Relating classical logic to constructive logic

We have already seen how Markov's principle (MR) can be implemented as a search operator in a programming logic. Now we look at an interesting metamathematical result concerning Markov's principle and AHA, and show how to carry it out in ML. Here is the formulation of Markov's rule again:

$$\text{MR} \qquad \frac{\neg\neg\, \mathit{Some}\ x\,.\,A(x), \quad \mathit{All}\ x\,.\,A(x) \mid \neg A(x)}{\mathit{Some}\ x\,.\,A(x)}$$

AHA is closed under the MR rule; that is, any proof in AHA using MR can be replaced with a proof that does not. Using this result, it has been demonstrated that AHA proves the same Π_2^0 sentences (in the Kleene or arithmetical hierarchy) as classical Peano Arithmetic. This theorem was proved by Harvey Friedman for Heyting Arithmetic [Friedman 1978]. A constructive proof of this fact is an algorithm that transforms a proof in Peano Arithmetic to a proof in AHA. This is the program that will be carried out in this section by describing the ML function that realizes the transformation.

The importance of the transformation to automatic deduction rests in the convenience it provides in establishing some proofs. In certain circumstances proofs may be developed using nonconstructive proof techniques like the law of excluded middle or classical proof by contradiction. These proofs can then be transformed to constructive proofs and used as programs just like those proofs developed using only constructive proof techniques.

In order to best indicate the meaning of Friedman's theorem we recast it in the language of Turing machines.

Theorem. Every Turing machine that, provably in Peano Arithmetic, converges at all arguments does so provably in AHA.

What does this mean? The importance of the theorem rests in the fact that proofs in Peano Arithmetic (PA) can use the axiom of excluded middle, but proofs in AHA do not, because of the constructivity requirement. In fact, we will understand our definition of PA to be AHA plus the one additional axiom

$$\text{ExMid} \qquad \phi \mid \neg \phi,$$

the law of excluded middle, which was introduced earlier in the discussion of constructive mathematics. In general, we do not expect a proof in PA to be convertible to one in AHA. For example the proof of

$$\mathit{All}\ P\,.\,P \mid \neg P$$

cannot be converted. In the case of proving Turing machines to be total, it is possible to eliminate the use of the excluded middle.

To help clarify the theorem further, we consider an example where one might incorrectly think that the theorem applies and another example of where the theorem might actually be used. Suppose we define a Turing machine to compute the following function. The function will be based on some unsettled conjecture of mathematics, for instance, Fermat's last theorem: for all $n > 2$ there are no integer solutions to the equation $x^n + y^n = z^n$. The function f is then defined by

$$f(x) = \begin{cases} 1, & \text{if conjecture true;} \\ 0, & \text{if conjecture false.} \end{cases}$$

This function is obviously total. Can we use a proof of this fact in PA and the theorem to find a proof in AHA that the function is total? The first step in answering this question is to formalize the notion of a total function. The obvious method is to use the Kleene T-Predicate: $T(i, x, y, t)$ holds if, and only if, Turing machine i converges in t steps on input x to the value y. Now then, can one prove $\mathit{Some}\, y, t \,.\, T(i_0, x, y, t)$ where i_0 is the index of a Turing machine which computes $f(x)$? Such a proof is easy. The conjecture is either true or false; in either case $f(x)$ is a constant function which is easily shown to be total. Can this proof be converted to a proof in AHA as the theorem seems to indicate? The answer is no. The reason is that the result does not apply. Notice that we cannot find the index of the Turing machine which computes the function f without first solving the conjecture. Thus we do not have a proof that i_0 is total, but rather a proof that for any x there is some i. This is not quite what we had in mind.

For an example of where the theorem does apply, consider a set of natural numbers characterized by some predicate P. If we have a Turing machine i_0 which recognizes every n such that $P(n)$ holds and if we have a Turing machine j_0 which recognizes every n such that $\neg P(n)$ holds, then we know that the set characterized by P is recursive. This means there is a total Turing machine, say with index d, such that it answers "yes" or "no" for every input n. The typical argument calls for dovetailing Turing machines i_0 and j_0. This means that Turing machine d first simulates machine i_0 for awhile, then simulates machine j_0, and goes back and forth like that until one machine or the other halts. Since by the axiom of excluded middle $P(x)$ or $\neg P(x)$ holds, either i_0 will halt or j_0 will halt, and hence the Turing machine d is total. The theorem states that this proof can be transformed into a proof which does not use the axiom of excluded middle. We will return to this example later.

We now go into the details of Friedman's proof of the theorem. This requires describing two translations of formulas: the double-negation translation and the A-translation. Corresponding to each translation there is a metatheorem that shows how to transform a proof of a formula to a proof of the translated formula.

Double-negation translation. The double-negation translation of a formula ϕ is denoted by ϕ^-. The translation is given in the following table.

$$
\begin{array}{rcll}
True^- & \longmapsto & True & \\
False^- & \longmapsto & False & \\
\phi^- & \longmapsto & \neg\neg\phi & \text{for atomic } \phi \\
(\neg\phi)^- & \longmapsto & \neg(\phi^-) & \\
(\phi \,\&\, \psi)^- & \longmapsto & \phi^- \,\&\, \psi^- & \\
(\phi \mid \psi)^- & \longmapsto & \neg\neg(\phi^- \mid \psi^-) & \\
(\phi \Rightarrow \psi)^- & \longmapsto & \phi^- \Rightarrow \psi^- & \\
(All\ x\,.\,\phi)^- & \longmapsto & All\ x\,.\,(\phi^-) & \\
(Some\ x\,.\,\phi)^- & \longmapsto & \neg\neg\, Some\ x\,.\,(\phi^-) &
\end{array}
$$

The case for the negation of a formula is redundant, because we have taken $\neg\phi$ to be an abbreviation for $\phi \Rightarrow$ *False*. It is, of course, easy to perform the translation in AHA using an ML function. We call the function `NegNeg` and give it in full as it is the first example of a function that takes advantage of the structure of formulas in AHA.

```
val NotNot (phi) = Not (Not (phi));      { An abbreviation. }

val rec NegNeg (phi) =
  if IsExpTrue (phi) then True               { reduced! }
  else if IsFalse (phi) then False           { reduced! }
  else if IsEqual (phi) then NotNot (phi)    { An equality }
  else if IsAnd (phi) then
    And (NegNeg (FirstConj phi), NegNeg (SecondConj phi))
  else if IsOr (phi) then
    NotNot (Or (NegNeg (FirstDisj phi), NegNeg (SecondConj phi)))
  else if IsImply (phi) then
    Imply (NegNeg (Hypothesis phi), NegNeg (Conclusion phi))
  else if IsAll (phi) then All (AllVar phi, NegNeg (AllBody phi))
  else if IsSome (phi) then
    NotNot (Some (SomeVar phi, NegNeg (SomeBody phi)))
  else NotNot (phi);        { O/w assume it is a predicate. }
```

We now note the following metamathematical theorem. If PA $\vdash \phi$, then HA $\vdash \phi^-$. Kleene [Kleene 1964] and Myhill [Myhill 1973] as well as Friedman have proved versions of this theorem for different translations. In all cases a syntactic proof suffices. We give the ML function to transform a proof of some formula ϕ in PA (AHA plus the law of excluded middle) to a proof in AHA of the formula ϕ^-.

The recursive function to transform a proof is a large case statement. There are too many cases to consider all of them in detail, so we will examine some representative ones. The transformation function, called `Transform`, begins as follows.

```
val rec Transform (prex) =
    { The axioms }
        if IsExpTruth (prex) then prex      { Reduced! }
        else if IsPP8 (prex) then NotNotTransform (prex)
        else if IsBetaReduction (prex) then NotNotTransform (prex)
        else if IsRefl (prex) then NotNotTransform (prex)
        ...
```

The very first case is the most simple. The axiom proving `True` (remember it is the proof expression `Truth`) is transformed to itself. We have defined double-negation translation of `True` to be `True` instead of $\neg\neg True$ to reduce the clutter of "not" signs somewhat. So `Truth`, the proof of `True`, is also the proof of the double-negation translation of `True`. The transform affected is thus the identity transform. The remainder of the axioms (ignoring the law of excluded middle for the moment) are all perfectly good proofs in AHA; we need only see how to convert the proof using one of these axioms to a proof in AHA of the double-negation translation. The same proof transformation, implemented by the ML function `NotNotTransform`, works for all of them. Before we can describe this function we must introduce a common proof technique that is often needed in a sea of negated formulas: proof by contradiction or *reductio ad absurdum*. In words, this sort of argument goes like this: assume some hypothesis A, then if B and $\neg B$ are derivable, conclude $\neg A$. This is implemented in ML as follows.

```
val Contra (hyp, prex1, prex2) =
  NotIntro (hyp, FalseIntro (AndIntro (prex1, prex2)));
```

Now the `NotNotTransform` is easily defined using this technique.

```
    { From AHA |- A get AHA |- ~~A }

val NotNotTransform (prex) =
  let val A = ProofOf prex;
      val raa = Assume ("raa", Not A) in
        Contra (raa, prex, raa)
  end;
```

Now we examine the one axiom in PA that is not a part of AHA. The axiom of excluded middle is not a rule of inference in AHA, so a proof of $(\phi \mid \neg\phi)^-$ must be produced out of thin air. Since $(\phi \mid \neg\phi)^-$ is equivalent to $\neg\neg(\phi^- \mid \phi^-)$ we need a proof of a formula in that form. In fact it is the case for any ψ that $\neg\neg(\psi \mid \neg\psi)$. The proof runs like this: assume $\neg(\psi \mid \neg\psi)$ for purposes of contradiction; likewise assume ψ; hence $\psi \mid \neg\psi$, a contradiction with the first assumption; similarly assuming $\neg\psi$ yields a contradiction; thus from the first assumption we have proved $\neg\psi$ and ψ; this is a contradiction, so we must have $\neg\neg(\psi \mid \neg\psi)$. Here is the proof in AHA.

```
      { AHA |- ~~(A | ~A) }

val NotNotExMid (A) =
  let val raa1 = Assume ("raa1", A);
      val raa2 = Assume ("raa2", Not A);
      val raa3 = Assume ("raa3", Not (Or (A, Not A)));
      val prex1 = Contra (raa1, OrIntroR (raa1, Not A), raa3); { ~A}
      val prex2 = Contra (raa2, OrIntroL (A, raa2), raa3) in   {~~A}
            Contra (raa3, prex1, prex2)
  end;
```

Now we can extend the definition of `Transform` to cover the law of excluded middle. We must find a proof of $(\phi \mid \neg\phi)^-$ which is equivalent to $(\phi^- \mid \neg\phi^-)$. Such a proof is provided by applying `NotNotExMid` to ϕ^-.

```
...
else if IsExMid (prex) then NotNotExMid (NegNeg (ExForm prex))
...
```

In the above ML expression, the function ExForm extracts the formula A out of which the law of excluded middle $(A \mid \neg A)$ is formed and NegNeg is the function which computes the double-negation translation of a formula.

One case that must not be overlooked is that of a proof expression that consists of only an assumption.

```
...
else if IsAssume (prex)
       then Assume (IndividualName prex, NegNeg (ProofOf prex))
...
```

The assumption is just changed to an assumption of the same name, but it now assumes the double-negation translation of the original formula.

The next class of proof expressions we will examine are those with conclusions which are atomic formulas. One might be tempted to use NotNotTransform on all those proof expressions using these rules (symmetry, transitivity, congruence, etc.), since ϕ^- is $\neg\neg\phi$ for atomic ϕ. This approach fails, since it overlooks the case where the law of excluded middle has been used in proving one of the premises. Hence the proof of the premise is not an AHA proof and must not be used in constructing an AHA proof of the conclusion. This is not the only reason the transform must be applied recursively to all premises. If any premise contains an assumption, it too must be transformed to an assumption of the double negation, for otherwise it may not be properly discharged.

Each case must be examined individually. The case of symmetry is representative of the general method. Next we define a function to transform a proof by symmetry. Ultimately the goal is to take a proof PA $\vdash m = n$ and obtain AHA $\vdash \neg\neg m = n$. Since the premise of the symmetry rule (PA $\vdash n = m$) is available we can call Transform recursively on it to get AHA $\vdash \neg\neg n = m$. Thus the difficult part is to convert this to AHA $\vdash \neg\neg m = n$.

```
{       AHA |- ~~n=m  yields  AHA |- ~~m=n    }

val TransformSym (prex) =
  let val phi = NegatedFormula (NegatedFormula (ProofOf prex));
      val psi = Equal (RHSEqual phi, LHSEqual phi);
```

```
    val raa1 = Assume ("raa1", phi);
    val raa2 = Assume ("raa2", Not (psi))
in
     Contra (raa2, Contra (raa1, Symmetry raa1, raa2), prex)
end;
```

A recursive call transforms the premise of a proof by symmetry into the proof of its double-negation translation. This is suitable input for the function **TransformSym** which we have just described.

```
...
else if IsSymmetry (prex) then TransformSym (Transform (Premis prex))
...
```

Most of the remaining rules are straightforward.

```
...
else if IsAndIntro (prex) then
  AndIntro (Transform (Premis1 prex), Transform (Premis2 prex))
else if IsImpElim (prex) then
  ImpElim (Transform (Premis1 prex), Transform (Premis2 prex))
else if IsFalseElim(prex) then
  FalseElim (Transform (Premis1 prex), NegNeg (Premis2 prex))
...
```

Only two rules pose a special problem, the or-elimination rule and the some-elimination rule. Fortunately the same method of attack works for both rules, so we examine or-elimination here. We recapitulate the or-elimination rule:

$$\frac{A \mid B, \quad A \Rightarrow C, \quad B \Rightarrow C}{C}$$

We would then like the or-elimination transform to take the three transformed premises

$$\mathrm{AHA} \vdash (A \mid B)^- = \neg\neg(A^- \mid B^-)$$

$$\mathrm{AHA} \vdash (A \Rightarrow C)^- = A^- \Rightarrow C^-$$

$$\mathrm{AHA} \vdash (B \Rightarrow C)^- = B^- \Rightarrow C^-$$

and return a proof $\mathrm{AHA} \vdash C^-$. However, it is not the case that for arbitrary A, B, and C

$$\mathrm{AHA} \vdash \neg\neg(A \mid B), \quad \mathrm{AHA} \vdash A \Rightarrow C, \quad \mathrm{AHA} \vdash B \Rightarrow C$$

imply AHA $\vdash C$.

This relationship holds only for A, B, and C which are images of the double-negation translation. On the other hand, it is the case that

$$\text{AHA} \vdash \neg\neg(A \mid B), \quad \text{AHA} \vdash A \Rightarrow C, \quad \text{AHA} \vdash B \Rightarrow C$$

imply AHA $\vdash \neg\neg C$. This fact, together with the Double Negation theorem: AHA $\vdash \neg\neg\phi^-$ implies AHA $\vdash \phi^-$, leads to the correct transformation of the or-elimination rule.

```
{ From AHA |- ~~(A|B), AHA |- A=>C, and AHA |- B=>C  get  AHA |- C
  (A,B,C must be transforms of other formulas.)                   }

val TransformOrElim (prex1, prex2, prex3) =
  let val OrFormula=NegatedFormula (NegatedFormula (ProofOf prex1));
      val raa1=Assume ("raa1", OrFormula);
      val raa2=Assume ("raa2", Not (Conclusion (ProofOf prex2)));
      val prex4 = OrElim (raa1, prex2, prex3);
      val prex5 = Contra (raa1, prex4, raa2)               {~A|B}
  in
        UnNotNot (Contra (raa2, prex5, prex1))
  end;
```

The ML function `UnNotNot` implements the transformation which the Double Negation theorem provides. The or-elimination clause in `Transform` is as follows.

```
...
else if IsOrElim (prex)
       then TransformOrElim (Transform (Premis1 prex),
                             Transform (Premis2 prex),
                             Transform (Premis3 prex))
...
```

An important fact about the double-negation translation is that AHA $\vdash \neg\neg\phi^-$ implies AHA $\vdash \phi^-$. We call this the Double Negation lemma. The ML function `UnNotNot` provides a proof of this lemma.

```
  { From AHA |- ~~A  get  AHA |- A
      (A must be a transform of some other formula.)  }

val rec UnNotNot (prex) =
  let val phi = NegatedFormula(NegatedFormula(ProofOf prex)) in
    if IsNot (phi) then UnNotNotNot (prex)
    else if IsAnd (phi) then UnNotNotAnd (prex)
    else if IsImply (phi) then UnNotNotImply (prex)
    else if IsAll (phi) then UnNotNotAll (prex)
    else Error
  end
  ...
```

The mutually recursive definition of UnNotNot breaks into four cases depending on the main connective. (The other connectives are not possible for translated formulas.) Here is the case for conjunction.

```
  ...
   { From  AHA |- ~~(A&B)  get  AHA |- (A&B)
       (A and B must be transforms of other formulas.) }

and UnNotNotAnd (prex) =
  let val phi = NegatedFormula (NegatedFormula (ProofOf prex));
      val A,B = FirstConj (phi), SecondConj (phi);
      val raaA, raaB = Assume ("raaA",Not A), Assume ("raaB",Not B);
      val px1a, px1b = OrIntroR (raaA,Not B), OrIntroL (Not A,raaB);
      val px2a, px2b = DeM (px1a), DeM (px1b);       { ~(A&B) }
      val px3a = Contra (raaA, px2a, prex);          { ~~A    }
      val px3b = Contra (raaB, px2b, prex)           { ~~B    }
  in
        AndIntro (UnNotNot px3a, UnNotNot px3b)
  end
  ...
```

The ML function DeM implements one of DeMorgan's laws: $\neg A \mid \neg B \Rightarrow \neg(A \,\&\, B)$.

This concludes the description of the function to transform a proof in PA to one in AHA of the double-negation translation. The function Transform has the important property that for all proof expressions *prex* it is the case that

NegNeg (ProofOf *prex*) = **ProofOf (Transform** *prex*).

The theorem for all *proofs* (proof expressions with no undischarged assumptions) follows as a special case from this property and the fact that **Transform** does not add any undischarged assumption where there were none before.

We have in mind applying **Transform** to proofs of the form *All x . Some y . T(x,y)*. The double-negation translation of this formula is

$$All\ x\,.\,\neg\neg\,Some\ y\,.\,\neg\neg\,T(x,y).$$

For a decidable predicate $T(x)$ this can be reduced to $All\ x\,.\,\neg\neg\,Some\,y\,.\,T(x,y)$. We give the proof here as we will need it later in the example.

```
Theorem: DecidablePred
Declaration of variables
    DT: Data type variable
    M: Relation symbol with signature DT
    t, t0: DT
To prove: All DT. All M. (All t. M(t) | ~M(t)) & Some t. ~~M(t) =>
                         Some t. M(t)
AllIntro
  Let DT and M be arbitrary
  ImpIntro
    Assume "hyp" : All t. M(t)|~M(t) &  Some t. ~~M(t)
    SomeElim
      Some t. ~~M(t) by AndElimL
        Assumption "hyp"
      Choose t0 where "shyp" : ~~M(t0)
      Some t. M(t) by Cases
        M(t0) | M(t0) by AndElimR
          AllElim
            Assumption "hyp"
            t0
        M(t0) => Some t. M(t) by ImpIntro
          Assume "case1" : M(t0)
          SomeIntro
            To show : Some t. M(t)
            With t0 for t
            Assumption "case1"
        ~M(t0) => Some t. M(t) by ImpIntro
```

```
            Assume "case2" : ~M(t0)
            Some t. M(t) by FalseElim
                FalseIntro
                  ~M(t0) & ~~ M(t0) by AndIntro
                    Assumption "case2"
                    Assumption "shyp"
QED
```

Having removed one double-negation, it is only natural to remove the other double-negation. This provides the motivation for the following theorem proved by Friedman. If HA $\vdash \neg\neg Some x.T(x)$ for atomic formula $T(x)$, then HA $\vdash$ $Some\, x\,.\,T(x)$. This theorem is remarkable because it produces a constructive proof of an existential statement where there was none before. We wish to examine this proof more carefully and will find that the proof holds in AHA with the caveat that the domain over which x is quantified is non-empty.

Friedman's proof divides into two parts. Lemma 1. If HA $\vdash \phi$, then HA $\vdash \phi_A$. Here ϕ_A represents the A-translation of the formula ϕ which will be introduced shortly. Lemma 2. For E equal to $Some\, x\,.\,T(x)$, if HA $\vdash (\neg\neg\, Some\, x\,.\,T(x))_E$, then HA $\vdash$ $Some\, x\,.\,T(x)$, for atomic formula $T(x)$. The two lemmas together imply the theorem.

A-translation. We now introduce the A-translation of a formula. The A-translation of a formula "ors" the formula A to all the atomic subformulas.

$$
\begin{array}{rcl}
True_A & \longmapsto & True \\
False_A & \longmapsto & A \\
\phi_A & \longmapsto & \phi \mid A \quad \text{for atomic } \phi \\
(\neg\phi)_A & \longmapsto & \phi_A \Rightarrow A \\
(\phi \,\&\, \psi)_A & \longmapsto & \phi_A \,\&\, \psi_A \\
(\phi \mid \psi)_A & \longmapsto & \phi_A \mid \psi_A \\
(\phi \Rightarrow \psi)_A & \longmapsto & \phi_A \Rightarrow \psi_A \\
(All\, x\,.\,\phi)_A & \longmapsto & All\, x\,.\,(\phi)_A \\
(Some\, x\,.\,\phi)_A & \longmapsto & Some\, x\,.\,(\phi)_A
\end{array}
$$

The ML function which effects the translation is called **A_Translate**.

Lemma 1 is proved for AHA by writing a proof transformer with the property that every proof of a formula ϕ is transformed to a proof of the A-translation of ϕ. Once again we examine only a few representative cases of this transformer which is implemented by the ML function **A_Transform**.

```
val rec A_Transform (prex, A) =
    {  The axioms. }
        if IsExpTruth (prex) then prex         { Reduced! }
        else if PP8 (prex) then TransformPP8 (prex, A)
        else if IsBetaReduction (prex) then OrIntroR (prex, A)
        else if IsRefl (prex) then OrIntroR (prex, A)
    {  An assumption. }
        else if IsAssume (prex)
               then Assume (Name prex, A_Translate (ProofOf prex, A)
    {  Rules of inference.  (Conclusions are atomic)  }
        else if IsPP7 (prex)
          then A_TransformPP7 (A_Transform prex, A)
        else if IsSymmetry (prex)
          then A_TransformSym (A_Transform prex, A)
        ...
    { Rules of inference.  (continued) }
        else if IsAndIntro (prex)
               then AndIntro (Transform (Premis1 prex),
                              Transform (Premis2 prex))
        else if IsImpElim (prex)
               then ImpElim (Transform (Premis1 prex),
                             Transform (Premis2 prex))
        ...
```

Some of the rules need comment. One is the Peano postulate 8, an axiom with no premises. It is not a proof of an atomic formula, so it is not possible to just "or" on the formula A, as was done for the those axioms with atomic conclusions.

```
    {  From  AHA |-  Phi => False  get  AHA |- Phi|A => A  }

val TransformPP8 (prex, A) =
  let val imp = ProofOf prex;       {  imp in form phi => false, }
      val phi = Hypothesis imp;     {  where phi is atomic.      }
      val as1 = Assume ("as1", phi);
      val as2 = Assume ("as2", A);
      val as3 = Assume ("as3", Or (phi, A));
      val case1 = ImpIntro (as1, FalseElim (ImpElim (prex,as1), A));
      val case2 = ImpIntro (as2, as2)
```

```
in
      ImpIntro (as3, OrElim (as3, case1, case2))
end;
```

All the premises of all the rules must be transformed in order that all assumptions are transformed and thereby correctly discharged. Even if the conclusion of the rule is atomic the premises must be transformed. Representative of this is Peano Postulate 7.

```
{   AHA |- n'=m' | A    yields    AHA |- n=m | A }

val A_TransformPP7 (prex, A) =
  let val Disj1 = FirstDisj (ProofOf prex);      { n'=m' }
      val c1 = Assume ("c1", Disj1);
                                            { n'=m' => (n=m | A) }
      val prex1 = ImpIntro (c1, OrIntroR (PeanoPostulate7 c1, A))
      val a = Assume ("a", A);
      val eq = Equal (Argument (LHSEqual Disj1),
                      Argument (RHSEqual Disj1));
      val prex2 = ImpIntro (a, OrIntroL (eq, a));  { A=>(n=m|A) }
  in
      OrElim (prex, prex1, prex2)
  end;
```

Another rule that causes problems is the false-elimination rule. It requires the following lemma about the A-translation:

$$\text{AHA} \vdash A, \qquad \text{implies} \qquad \text{AHA} \vdash \phi_A,$$

for all formulas ϕ whose existential quantifiers range over non-empty domains. The ML function **A_Proof** implements this lemma. Now we can tackle the false-elimination rule. If *prex* is a proof by false-elimination, then its one premise is a proof of false. (We hope this is only because of contradictory assumptions!) Calling **A_Transform** recursively on this proof of false yields a proof of $false_A$ which is just A. Hence we have a proof of A and we want a proof of ϕ_A, where ϕ is the formula introduced by the false-elimination rule. This is exactly what the previous lemma was designed to produce.

...

```
if IsFalseElim (prex)
    then A_Proof (A_Transform (Premis1 prex, A), FEForm prex)
...
```

The function **FEForm** extracts that formula ϕ which the false-elimination rule introduced.

The following is the ML function **A_Proof** which serves as a proof of the preceding lemma.

```
{        AHA |- A  implies  AHA |- Phi_A  }

val rec A_Proof (prex, phi) =
  let val A = TypeOf prex in
    if IsExpTrue (phi) then Truth  { Reduced from:  Or(True, A) }
    else if IsFalse (phi) then prex { Reduced from: Or(False,A) }
    else if IsEqual (phi) then OrIntroL (phi, prex)
    else if IsAnd (phi)
             then AndIntro (A_Proof (prex, FirstConj phi),
                            A_Proof (prex, SecondConj phi))
    else if IsOr (phi)
             then OrIntroR (A_Proof (prex, FirstDisj phi),
                            A_Translate (SecondConj phi, A))
    else if IsImply (phi) then ImpIntro (
               Assume ("a", A_Translate (Hypothesis phi, A)),
               A_Proof (prex, Conclusion phi))
    else if IsAll (phi)
           then AllIntro (AllVar phi, A_Proof (prex,AllBody phi))
    else if IsSome (phi) then SomeIntro (
                       A_Translate (phi, A),
                       A_Proof (prex, SubsScope (phi, t0)),
                       t0)
        where val t0 = AnElementOf (TypeOf (SomeVar phi)) end
    else OrIntroL (phi, prex)    { O/w assume predicate symbol }
  end;
```

The assumption that existential quantifiers range over non-empty domains is used to find a witness (any will do) for the construction of a **SomeIntro** proof expression. Normally in logic non-empty domains are taken for granted, but we do not wish to preclude interest in a data type **Void**, for instance.

This completes the discussion of **A_Transform** which serves as a proof of Lemma 1.

For Lemma 2 we prove the slightly stronger result that when E is set to be $Some\, x\,.\,T(x)$, the formula $(\neg\neg\, Some\, x\,.\,T(x))_E \Rightarrow Some\, x\,.\,T(x)$ is provable in AHA.

```
Theorem:  Simplify
Declaration of variables:
    DT:  Data type variable
    M:  Relation variable with signature DT
    z, z0:  DT
Abbreviations:
    E = Some z. M(z)

To prove:  All DT. All M. ((Some z. (M(z)|E) => E) => E) => E
ImpIntro
  Assume "hyp1" : (Some z . (M(z)|E) => E) => E
  Some z. M(z) by ImpElim
    Assumption "hyp1"
    Some z. (M(z)|E) => E by ImpIntro
      Assume "hyp2" : Some z. M(z)|E
      Some z. M(z) by SomeElim
        Some z. (M(z)|E) by Assumption "hyp2"
        Choose z0 where "shyp" : M(z0) | Some z. M(z)

        Some z. M(z) by Cases
          M(z0)| Some z. M(z) by Assumption "shyp"
          M(z0) => Some z. M(z) by ImpIntro
            Assume "case1" : M(z0)
            SomeIntro
              To show : Some z. M(z)
              With z0 for z
              M(z0) by Assumption "case1"
          Some z. M(z) => Some z. M(z) by ImpIntro
            Assume "case2" : Some z. M(z)
            Assumption "case2"
QED
```

Summary. We now summarize the steps involved in changing a "for-all there-exists" proof in PA to one in AHA for decidable, atomic predicates. We start with a proof in PA.

$$PA \vdash All\, x\,.\,Some\, y\,.\,T(x,y)$$

There is an AHA proof of the double-negation translation.

$$\text{AHA} \vdash \textit{All } x \,.\, \neg\neg\, \textit{Some } y \,.\, \neg\neg\, T(x,y)$$

Since $T(x,y)$ is decidable this proof can be simplified.

$$\text{AHA} \vdash \textit{All } x \,.\, \neg\neg\, \textit{Some } y \,.\, T(x,y)$$

The A-translation of this is provable.

$$\text{AHA} \vdash (\textit{All } x \,.\, \neg\neg\, \textit{Some } y \,.\, T(x,y))_E$$

This last proof can be further simplified.

$$\text{AHA} \vdash \textit{All } x \,.\, \textit{Some } y \,.\, T(x,y)$$

The dovetailing example. We now consider the dovetailing example again. The appropriate theorem is proved in AHA plus the axiom of excluded middle, and the transformation outlined in this section is carried out.

We introduce the Kleene T-predicate as an uninterpreted relation symbol and axiomatize it. In axiomatizing it, we go no further than magically assuming that it is decidable. It is an interesting exercise to implement the T-predicate as an interpreted relation or function symbol, and to implement the appropriate axioms. All this is not needed for the main point of the dovetailing example, so we ignore these details.

```
val TPred = RelationSymbol ("T",
  Cross (Nat, Cross (Nat, Cross (Nat,Nat))));
val T (i,x,y,t) =
  Application (TPred, Pair (i, Pair (x, Pair (y,t))));

val i = Variable ("i", Nat);
val x = Variable ("x", Nat);
val y = Variable ("y", Nat);
val t = Variable ("t", Nat);

val Ax = Or (T(i,x,y,t), Not (T(i,x,y,t)));
val Axiom = nAll ([i;x;y;t], Magic Ax);
```

The predicate P characterizes a set of integers. We are going to show that the dovetailing Turing machine (TM) can decide if its input is in the set or not.

```
val Psym = RelationSymbol ("P", Nat);
val P (term) = Application (Psym, term);
```

For convenience we make mnemonic abbreviations for the two outputs we are interested in.

```
val No = Zero;
val Yes = Succ (Zero);
```

The constant d is the index of the dovetailing TM.

```
val d = FuncSymbol ("d", Nat);
```

If $P(x)$ holds, then TM d is assumed to halt with an affirmative answer.

```
val Ax1 = Imply (P(x), Some (t, T(d,x,Yes,t)));
val Axiom1 = All (x, Magic Ax1);
```

If $\neg P(x)$ holds, then TM d is assume to halt with a negative answer.

```
val Ax2 = Imply (Not (P(x)), Some (t, T(d,x,No,t)));
val Axiom2 = All (x, Magic Ax2);
```

Here is a formal proof in PA of the dovetailing theorem.

```
Theorem: DoveThm
Declaration of variables:
    x: Nat
    p: Cross (Nat, Nat)
Abbreviations:
    p1 = First p
```

```
    p2 = Second p
To prove:  All x. Some p. T(d,x,p1,p2)
AllIntro
  Let x be arbitrary
  Cases
    P(x) | ~P(x) by ExMid

    P(x) => Some p. T(d,x,p1,p2) by ImpIntro
      Assume "case1" : P(x)
      T(d,x,p1,p2) by SomeELim
        Some t. T(d,x,Yes,t) by ImpElim
          Lemma "Axiom1"
          Assumption "case1"
        Choose t0 such that "shyp": T(d,x,Yes,t0)
        Some p. T(d,x,p1,p2) by SomeIntro
          To show : Some p. T(d,x,p1,p2)
          With (Yes,t0) for p
          Assumption "shyp"

    ~P(x) => Some p. T(d,x,p1,p2) by ImpIntro
        [ The proof is analogous to the previous case.]
QED
```

Now take this proof in PA and transform it to a proof in AHA of the double-negation translation.

```
val Step1 = Reduce (Transform (DoveTheorem));
```

The proof expression **Step1** is a proof of $All x.\neg\neg Some p.\neg\neg T(d,x,p_1,p_2)$. Now we want to eliminate the double negation. The innermost double negation can be eliminated since the T-predicate is decidable. The appropriate proof expression is obtained using the theorem **DecidablePred** proved earlier.

```
val prex1 = AllElim (DecidablePred, Cross (Nat,Nat));
val z = Variable ("z", Cross (Nat, Nat));
val prex2 = AllElim (prex1, Lambda (z, T(d,x,First z,Second z)));
val prex3 = Reduce (prex2);
val prex4 = AllElim (prex3, x);
val prex5 = AndIntro (Axiom, AllElim (Step1, x))
```

```
val Step2 = Reduce (AllIntro (x, ImpElim (prex4, prex5)));
```

The proof expression **Step2** is a proof of $All\ x\,.\,\neg\neg\, Some\, p\,.\,T(d,x,p_1,p_2)$.

```
val E = Some (p, T(d,x,p1,p2)));
val Step3 = Reduce (A_Transform (Step2, E));
```

The result is a proof of

$$All\ x\,.\,((Some\ y\,.\,Some\ t\,.\,(T(d,x,p1,p2)\ |\ E) \Rightarrow E) \Rightarrow E)$$

where E is $Some\, p\,.\,T(d,x,p1,p2)$.

The last step cleans up proof expression **Step3** using the simplification theorem.

```
val prex1 = AllElim (Simplify, Cross (Nat,Nat));
val z = Variable ("z", Cross (Nat, Nat));
val prex2 = AllElim (prex1, Lambda (z, T(d,x,First z,Second z)));
val prex3 = Reduce (prex2);
val prex4 = ImpElim (prex3, Step3);
val Step4 = Reduce (prex4);
```

What sort of proof expression is **Step4**? When applied to an actual value n of type **Nat** it will reduce to a some-introduction proof expression, thus exhibiting a pair (y,t) such that the Turing machine d will halt in t steps on input n with output y. (Of course, we must provide the proofs about the Turing machines i_0 and j_0 which we assumed magically existed.)

The computational meaning of the proof expression is given below.

```
if T(d,x,y1,t1)
  then (y1,t1)
  else if T(d,x,y2,t2)
    then (y2,t2)
    else (y0,t0)
```

The values y_0 and t_0 are default values assumed to exist since the range of existential quantifiers is non-empty. This second else-case is not expected to ever be taken. The only reason the proof expression **Step4** can offer (y_0, t_0) as a solution is because assuming not $T(d, x, y_1, t_1)$ and not $T(d, x, y_2, t_2)$ leads to a contradiction and thus any pair offered can be proved to be a solution.

In this section we have carried out a transformation of AHA proofs of Π_2^0 sentences using the law of excluded middle to AHA proofs without using that dubious axiom. For a more advanced study of what classically provable assertions are provable constructively in various other theories see [Beeson 1978].

IZF. A similar transformation exists for Π_2^0 sentences in IZF.

All the logical rules in IZF are the same as those in AHA. So the only question is whether the rules unique to IZF present a problem. They do not. We examine one of the rules to make plausible the claim that the transformation of Π_2^0 sentences goes through in IZF. We look at one rule, the union elimination rule:

$$\frac{T \in \bigcup(S)}{Some\; x \in S \,.\, T \in x}$$

We have PA $\vdash T \in \bigcup(S)$, and so by induction hypothesis we have AHA $\vdash \neg\neg T \in \bigcup(S)$. It remains to show that AHA $\vdash \neg\neg\, Some\; x \,.\, \neg\neg(x \in S \,\&\, T \in x)$. For purposes of contradiction, assume $\neg\, Some\; x \,.\, (\neg\neg x \in S \,\&\, T \in x)$. Now assume $T \in \bigcup(S)$. Hence, by the union elimination rule, $Some\; x \in S \,.\, T \in x$. So $Some\; x \,.\, (\neg\neg x \in S \,\&\, T \in x)$, a contraction. We are forced to conclude $\neg T \in Union(S)$, but this contradicts the hypothesis. Therefore, $\neg\neg\, Some\; x \,.\, (\neg\neg x \in S \,\&\, T \in x)$.

Consider the same union elimination rule under the A-translation. The A-translation of the hypothesis is:

$$(T \in \bigcup(S))_A = T \in \bigcup(S) \mid A$$

The A-translation of the conclusion is:

$$(Some\; x \in S \,.\, T \in x)_A = Some\; x \,.\, \big((x \in S \mid A) \,\&\, (T \in x \mid A)\big)$$

By induction hypothesis we have AHA $\vdash T \in \bigcup(S) \mid A$. The proof proceeds by cases. If $T \in \bigcup(S)$, then by union elimination $Some\; x \in S \,.\, T \in x$. Hence by some-elimination and or-introduction, $(x \in S \mid A) \,\&\, (T \in x \mid A)$. Therefore $Some\; x \,.\, \big((x \in S \mid A) \,\&\, (T \in x \mid A)\big)$. On the other hand, if A, then $x \in S \mid A$ and $T \in x \mid A$. So by some-introduction using any set whatsoever, we have again, $Some\; x \,.\, \big((x \in S \mid A) \,\&\, (T \in x \mid A)\big)$.

Chapter VI

Conclusion

In this final chapter we briefly review what has been accomplished in this thesis and suggest possible areas that deserve further investigation.

6.1 Review

Representing theories in ML calls for defining an abstract type corresponding to theorems of the theory. We started this thesis by describing ML and describing how LCF represented the nonconstructive logic PPLAMBDA in ML. The type security of ML insures that elements of the ML type theorem represent valid theorems in the logic. These elements of type theorem are called proof expressions. The LCF implementation had no need for the proof trees of theorems, but in this thesis we are interested in constructive theories. In these theories, proofs can be executed like programs and so we need the proof trees as part of the implementation. Since this interpreter evaluates proof expressions, it is necessary to choose the constructs of the theory carefully. It is this interpreter that turns a constructive theory into a programming language. We call these programming languages programming logics, because they are languages that express both reasoning and computation.

Programming logics are strongly typed, functional programming languages. In this thesis we described the simulation of three programming logics with an ML implementation. The three programming logics are fully implemented on a common logic engine written in ML. To simulate one of them, the appropriate ML type definitions and functions are loaded. Expressions can be built up, and the functions `Reduce` and `TypeOf` are available to evaluate and find the type of the expressions.

The first programming logic described in this thesis has application in number theory and is called AHA. This is a powerful language with a finite type hierarchy and quantification over propositions and propositional functions. In keeping with the traditional syntactic classes of logic we organize the objects in the language into terms, types, formulas, and proof expressions. Terms and proof expressions are the "expressions" in the strongly typed, functional programming language

AHA. These expressions must all have types. The type of a member of the syntactic class term is a member of the syntactic class type. The type of a proof expression is a formula.

Adding the axiom of choice to AHA makes it possible to introduce function symbols for the functions that realize "for-all there-exists" proofs. The interpreter can compute with these functions. This is a programming paradigm that programming logics have in common with PRL. Computing in this paradigm has three steps. First, prove a theorem of the form:

$$All\ x : T_1\ .\ Some\ y : T_2\ .\ P(x,y)$$

Second, introduce a function symbol, say f, for the function from T_1 to T_2 that for every x returns the y such that $P(x,y)$. Third, evaluate $f(x_0)$ for some value x_0 of type T_1 with the interpreter.

Some of the constructs in AHA are redundant. Two constructs are used where one will do (for example, `Variable` and `Assumption`, `Abstract` and `ImpIntro`, `Recursion` and `Induction`), because we are not accustomed to thinking that term constructors are the same sort of constructors as proof expression constructors. The redundant constructs are collapsed, yielding a new programming logic. This is the programming logic TT, and it resembles a type theory of Martin-Löf. In this theory the axiom of choice is provable. As an exercise in using the programming logic TT we introduced a construct for transfinite induction and defined the natural numbers and the successor function.

In Chapter Five we saw that metamathematics could be applied to the problem of developing proofs. For programming logics implemented in ML, it was natural to use ML as the meta-language. This was possible because proof expressions are objects in ML. We showed how refinement rules and tactics could be built for programming logics.

Other metamathematics can be used to develop proofs in programming logics. Another example is the algorithm for linear input resolution which forms the basis for PROLOG. Programming logics and PROLOG are completely different: programming logics depend on constructive mathematics, PROLOG on classical mathematics; one programs by proving in programming logics, but one programs by axiomatizing in PROLOG; and, programming logic uses the methods of verification, while PROLOG uses the methods of theorem-proving. Because proof expressions are objects in ML, a resolution theorem prover can be programmed to build proof expressions. Hence PROLOG is subsumed in a programming logic. Naturally, other theorem-proving techniques can be integrated as well.

Using metamathematics to develop proofs automatically enables programming logics to be a foundation for automatic deduction which conforms more closely to the high-level reasoning humans use to prove theorems. A wide variety of techniques can be applicable. Constructs that at first do not appear to fit in the framework of a programming logic, like the law of excluded middle, can, in certain circumstances, be used and then transformed. This is evidence of the value of applying metamathematical results to automatic deduction. We gave the ML program that transforms some proofs using the law of excluded middle to proofs that do not. Thus, in some cases classical logic can be added to the arsenal of proof development techniques without jeopardizing the ability to execute proofs as programs.

6.2 Further areas to investigate

One of the obvious goals that we would like to see realized is the construction of a practical system out the abstract systems described here. An easy way of realizing this quickly is to use the abstract syntax of a programming logic as input to the Cornell synthesizer generator [Teitelbaum and Reps 1984]. Already this has been done for a natural-deduction style predicate calculus [Reps and Alpern 1984], so the principle obstacle to overcome would be implementation of the interpreter.

As mentioned earlier, the great economy of constructs brought about by the "propositions-as-types" idea stimulates the investigation of even more economical underlying principles. The type theory V3 is an example. It would be interesting to build V3 on the logic engine described in this thesis. Some problems exist that were brought up earlier, but it appears that they can be overcome. Perhaps some modifications to the logic engine would be necessary. Certainly a simpler logic engine could be used for any combinatory theory without variables—substitution would not need to be supported. Computation would be accomplished wholly by pattern matching and replacement.

The PRL project at Cornell is examining a theory related to TT as the basis for a new paradigm in programming. It would be interesting to examine IZF as this basis. For example, the program for topological sort would be a proof of the theorem:

$$\mathit{All}\ x : \mathit{Set}\,.\,\mathit{PO}(x) \Rightarrow \mathit{Some}\ y : \mathit{Set}\,.\,\mathit{TO}(y)\ \&\ \mathit{Domain}(x) = \mathit{Domain}(y)$$

where $PO(x)$ means that the set x is a partial order and $TO(y)$ means that the set y is a total order. The resulting programs might be conveniently compiled into the programming language SETL.

In this thesis a method of implementing programming logics has been described. The logic engine has a certain number of capabilities which are used in implementing each programming logic. Every construct defined in the programming logic has basically the same structure. Certain constructs are related to each other. For example, the typed lambda-calculus has three constructs:

abstraction, application and beta-reduction. These relationships of introduction, elimination and computation can be seen among many groups of constructs. Such overall patterns suggest that one can automate the process of generating a programming logic, just as YACC generates a compiler and the Cornell synthesizer generator generates a structure editor. The key here would be a specification language. Had such a specification language existed, the description of the programming logics presented in this thesis would have been much easier. As it was, the specification language we used was natural-deduction style logic, and the mental jump to the meaning in ML was substantial. Had the actual ML implementation been used as the specification language instead, the meaning would be precisely given yet obscured by all the details.

Another project worth pursuing is compiling the computational content of programs (proof expressions) in programming logics. Interpreting proof expressions is an inefficient means of program execution. Already the PRL project is gaining some experience in compiling efficient LISP code. A practical question is whether one can compile efficient code in some widely used language, say ADA. If so, one could build a programming logic front-end that generates formally verified ADA programs. Programmers familiar with verification could program in a suitable programming logic using appropriate program development tactics, and produce libraries of reliable software.

Appendix A

Implementation

In this appendix we describe in greater detail the actual ML implementation of the logic engine. The logic engine is the core of the three programming logics AHA, TT and IZF. The rules for these programming logics are found in the subsequent appendices. Each programming logic is implemented in about 1000 lines of ML code written in about 10 modules each. This code depends on the abstract type definitions and functions defined in the logic engine. These definitions and functions are in two ML modules which are the subject of this appendix.

```
module Exp
includes AHALib              { import a couple of general functions.}
body
export                       { this module defines these types and }
                             { functions; all else is auxiliary.   }
   abstype Exp Binding
   val
        SynEq AlphaEq
        Repr RuleName IndividualName UnivNum
        Universe Variable Definition ExpOp
        IsUniverse IsVariable IsDefinition IsExpOp
        TypeOf Reduce Print
        FreeVars IsVarFreeIn Substitute
        BindingDestr
from
...
```

Now we come to the definition of the ML abstract types used in the implementation.

```
...
abstype rec Exp =
    UnivCon of int                      |
    VarCon of string * Exp              |
    DefCon of string * Exp * Exp * Exp  |
    ExpCon of Rule * (Binding list)
...
```

The representation of an expression is essentially a list of expressions (its arguments). The variable and definition constructors, **VarCon** and **DefCon**, are useful to keep variables and definitions apart from the other terms. The **UnivCon** is not, strictly speaking, a necessary component of the **Exp** type. It is basically an expression constructor with integer arguments in place of expression arguments. These integer arguments could be coded in the expression arguments or a union type of expressions and integers could be used. Both options cause far more problems than they solve. The only use made of the **UnivCon** constructor was to represent the type of types, or, more generally, the levels in a type hierarchy.

```
...
and Rule =

    RuleCon of
        string                                  { name }
      * (Exp -> Exp)                            { type }
      * ((Exp -> bool) * (Exp -> Exp)) list     { contr }
      * (Exp -> string)                         { print }

   { Binding is a (possibly empty) list of variables bound in exp. }

and Binding = BindingCon of (Exp list) * Exp
with
...
```

The rest of the module defines all the functions we need to manipulate the types **Exp**, **Rule**, and **Binding**.

```
...
val rule (t: Exp): Rule =
    case t of
        ExpCon (rule,repr)  . rule   |
        _                   . Error ["rule:  not expression op"];
val Repr (t: Exp): Binding list =
    case t of
        ExpCon (rule,repr)  . repr   |
        _                   . Error ["Repr:  not expression op"];
val RuleName (t: Exp): string =
    let val RuleCon (n,typeof,contr,print)=(rule t) in
        n
    end;
val typeof (t: Exp): Exp -> Exp =
    let val RuleCon (n,typeof,contr,print)=(rule t) in
        typeof
    end;
val contr (t: Exp): ((Exp -> bool) * (Exp -> Exp)) list =
    let val RuleCon (n,typeof,contr,print)=(rule t) in
        contr
    end;
val print (t: Exp): Exp -> string =
    let val RuleCon (n,typeof,contr,print)=(rule t) in
        print
    end;
...
```

The next functions are recognizers for the various possible elements of the type Exp.

```
val IsUniverse (exp: Exp): bool =
  case exp of
    UnivCon n      . true    |
    _              . false;
val IsVariable (exp: Exp): bool =
  case exp of
    VarCon (n,e)  . true    |
    _             . false;
val IsDefinition (exp: Exp): bool =
  case exp of
```

```
    DefCon _     . true    |
    _            . false;
  val IsExpOp (exp: Exp, n: string): bool =
    case exp of
     ExpCon _    . (RuleName exp)=n |
     _           . false;
```

The next functions manipulate bindings.

```
  val Binding (l: Exp list,e: Exp): Binding =
    if Every IsVariable l
      then BindingCon (l,e)
      else Error ["Binding: not var"];

  val BindingDestr (BindingCon b): (Exp list) * Exp = b;
```

The next functions implement the boolean-valued function which tests for syntactic equality. It depends heavily on the assumption that expressions defined using ExpOp have unique names.

```
  val EqRule (r: Rule, s: Rule): bool =
    let val (RuleCon (nr,_,_,_), RuleCon (ns,_,_,_)) = (r,s) in
      nr=ns
    end;
  val EqLists f (l1, l2) =
    (if length l1 = length l2 then AuxEqLists f (l1, l2) else false)
      where val rec AuxEqLists f (l1,l2) =
        if null l1 then true
        else if f (hd l1, hd l2) then AuxEqLists f (tl l1, tl l2)
        else false
       end;
  val rec EqBinding (b1: Binding, b2: Binding): bool =
    let val BindingCon (l1,e1) = b1;
        val BindingCon (l2,e2) = b2 in
      if EqLists SynEq (l1,l2) then SynEq (e1,e2) else false
    end
```

```
and SynEq (t: Exp, s: Exp): bool =
  case (t,s) of
    (UnivCon n, UnivCon m) .        n=m     |

    (VarCon (str1,t1), VarCon (str2,t2)) .
               if str1=str2 then SynEq(t1,t2) else false     |

    (DefCon (s1,x1,t1,e1), DefCon (s2,x2,t2,e2)). s1=s2      |

    (ExpCon (r1,b1), ExpCon (r2,b2)) .
        if EqRule (r1,r2)
          then EqLists EqBinding (b1,b2)
          else false                                        |

    (_,_)                                        . false;



val IndividualName (exp: Exp): string =
  case exp of
    UnivCon n          . stringofint n |
    VarCon (s,e)       . s             |
    DefCon (s,x,t,e)   . s             |
    ExpCon (r,b)       . RuleName exp;
val UnivNum (exp: Exp): int =
  case exp of
    UnivCon n. n |
    _         . Error ["UnivNum: not universe"];



val TypeOf (exp: Exp): Exp =
  case exp of
    UnivCon n          . UnivCon (n+1) |
    VarCon (s,e)       . e             |
    DefCon (s,x,t,e)   . t             |
    ExpCon _           . (typeof exp) (exp) ;
```

Now we give a simplified print function. The actual print function uses some non-functional programming to accomplish eliding and user-defined abbreviations.

```
val Print (exp: Exp): string =
  case exp of
    UnivCon n         .  "U" ^ (stringofint n) |
    VarCon (s,e)      .  s                     |
    DefCon (s,x,t,e)  .  "Def'n of " ^ s       |
    ExpCon _          .  (print exp) (exp);
```

```
val ExpOp (n,repr,typ,eva,pr) =
    ExpCon (RuleCon (n,typ,eva,pr), map Binding repr);
val Transform (exp: Exp, f: Binding -> Binding): Exp =
  ExpCon (rule exp, map f (Repr exp));
```

Next the functions to accomplish evaluation are presented.

```
val Applicable (rr,exp): bool =
  let val (p,t) = rr in p(exp) end;
val Contract (rr,exp): Exp =
  let val (p,t) = rr in t(exp) end;
val rec Reduce (exp: Exp): Exp =
  case exp of
    UnivCon n         . exp                  |
    VarCon (n,e)      . VarCon(n,Reduce e)   |
    DefCon (s,x,t,e)  . e                    |
    ExpCon _          .
      UseContractions (contr exp, Transform(exp,ReduceBinding))

and ReduceBinding (b: Binding): Binding =
  let val BindingCon (l,e) = b in
     Binding (map Reduce l, Reduce e)
  end

and UseContractions (rl, exp) =
  if null rl
    then exp
     else if Applicable (hd rl, exp)
     then Reduce (Contract (hd rl, exp))
     else UseContractions (tl rl, exp);
```

Substitution takes up the largest part of this module.

```
val rec IsExpInList (v,l) =
  if null l then false
  else if SynEq (v, hd l) then true
  else IsExpInList (v, tl l);
val rec UnionExp (l1,l2) =
  if null l1 then l2
  else if IsExpInList (hd l1, l2) then UnionExp (tl l1, l2)
  else (hd l1)::UnionExp (tl l1, l2);
val rec UnionExpList (l: ((Exp list) list)): Exp list =
  if null l then [] else UnionExp (UnionExpList (tl l), hd l);
val rec RemoveExp (v,l) =
  if null l then []
  else if SynEq (v, hd l) then tl l
  else (hd l)::RemoveExp (v, tl l);
val rec RemoveExpList (l1,l2) =
  if null l1
    then l2
    else RemoveExpList (tl l1, RemoveExp (hd l1, l2));
val rec FreeVarsInBinding (b: Binding) =
  let val BindingCon (l,e) = b in
    RemoveExpList (l, UnionExpList ((FreeVars e)::(map FreeVars l)))
  end
and
FreeVarsInBList l =
  if null l then []
  else if null (tl l) then FreeVarsInBinding (hd l)
  else UnionExp (FreeVarsInBList (tl l), FreeVarsInBinding (hd l))
and
FreeVars (exp: Exp) : Exp list =
  case exp of
    UnivCon n    . []    |
    VarCon (s,e). UnionExp (FreeVars e, [exp])  |
    DefCon _     . []    |        { Better not be any in a def'n }
    ExpCon _     . UnionExpList (map FreeVarsInBinding (Repr exp));
val FreeIn (vs, lt) = filter (fun x. IsExpInList (x, lt)) (vs);
```

That ends the functions to find the free variables in an expression.

Now we are ready to define the actual substitution operator.

```
val SubError (fv,v,t) =
  let val vs = FreeIn (fv, FreeVars t) in

Error ["Substitution failed.\L "; Print t;
  "\L has variable captured while substituting for the variable  ";
  Print v; ".\L The list of vars captured:\L\T";
  (SeparatedList (fun x. (Print x)^" : "^Print(TypeOf x))
               "\L\T" vs)]

  end;
val rec AuxSubBinding (fv,v,t,b): Binding =
  let val BindingCon (l,e) = b in
    if IsExpInList (v,l)
      then b
      else Binding (map (fun x. AuxSub (fv,v,t,x)) l,
                    AuxSub (l@fv,v,t,e))
  end
and
AuxSub (fv,v,t,e) =
  case e of
    UnivCon n     . e      |
    VarCon (n,ty) . if SynEq (e,v)
                      then if not (null (FreeIn (fv, FreeVars t)))
                        then SubError (fv,v,t)
                        else t
                      else VarCon (n, AuxSub (fv,v,t,ty))  |
    DefCon _ . e   |
    ExpCon _ . Transform (e, (fun b. AuxSubBinding (fv,v,t,b)));
val Substitute (v,t,e) = AuxSub ([],v,t,e) ?
  Error ["\LSubstitution failed.  Substituting\L "; Print t;
         "\L for the variable  "; Print v;
         "\L in the expression\L "; Print e; "\L"];
```

Some rules require restrictions on free variables. The next function is a useful check used by some rule constructors: does a variable v occur free in t?

```
val IsVarFreeIn (v,t) = IsExpInList (v, FreeVars t);
```

With substitution defined it is now possible to introduce alpha equality. Two terms are said to be alpha variants if a renaming of bound variables can make them syntactically equal. Alpha equality is crucially important when, in the construction of a formal theory, the system can introduce its own variable names.

```
{ Equality of expressions considering alphabetic variants equal and
    definitions equal to what they have been defined to. }
    { If the rules are equal then the shape of the bindings
        can be assumed to be equal.                              }
val rec AlphaBinding (b1: Binding, b2: Binding): bool =
  let val BindingCon (l1,e1) = b1;
      val BindingCon (l2,e2) = b2 in
    if (length l1) = (length l2)
      then AlphaVariants (l1,e1,l2,e2) else false
  end
and AlphaVariants (l1:Exp list, e1:Exp, l2:Exp list, e2:Exp):bool=
  if null l1
    then AlphaEq (e1,e2)
    else
    let val v1 = hd l1;
        val v2 = hd l2 in
      if AlphaEq (TypeOf v1, TypeOf v2)
        then AlphaVariants (map Subs1 (tl l1), Subs1 e1,
                            map Subs2 (tl l2), Subs2 e2)
            where val new = VarCon ("$ae" ^ NextNumber(),TypeOf v1);
                  val Subs1 t = Substitute (v1, new, t);
                  val Subs2 t = Substitute (v2, new, t)
            end
      else false
    end
and AlphaEq (t: Exp, s: Exp): bool =
  case (t,s) of
    (UnivCon n, UnivCon m) .       n=m    |
    (VarCon (str1,t1), VarCon (str2,t2)) .
        if str1=str2 then AlphaEq(t1,t2) else false    |
    (DefCon (str,x,ty,e), s') . AlphaEq (x,s')    |
{
    (t', DefCon (str,x,ty,e)) . AlphaEq (t',x)    |
}
    (ExpCon (r1,b1), ExpCon (r2,b2)) .
```

```
      if EqRule (r1,r2)
        then EqLists AlphaBinding (b1,b2) else false |
   (_,_)                                          . false;
```

Now for some simple constructors of the ML type Exp.

```
val Universe n = if n>0 then UnivCon n else Error["Univ: n<=0"];
val Variable (s,e) = VarCon (s,e);
val Definition (s,x) = DefCon (s, x, TypeOf x, Reduce x)
```

That is all that fits in this module.

```
end     {end of with}
end     {end of export}
end;    {end of module Exp}
```

The next module completes the definition of those functions used as the building blocks.

Computational equality of expressions is determined by the following function.

```
val CompEq (exp1, exp2) = AlphaEq (Reduce exp1, Reduce exp2);
```

```
val EmptyContractions = [];
val SelectBinding (n, exp) = nth (n-1, Repr exp)
                             ? Error ["SelectBinding"];
```

```
val Exp1 exp =
  let val (v,e) = BindingDestr (SelectBinding (1,exp)) in e end;
val Exp2 exp =
  let val (v,e) = BindingDestr (SelectBinding (2,exp)) in e end;
val Exp3 exp =
  let val (v,e) = BindingDestr (SelectBinding (3,exp)) in e end;
```

The list of variables bound in an expression is returned by the following functions.

```
val Var1 (exp: Exp): Exp list =
  let val (l,e) = BindingDestr (SelectBinding (1,exp)) in l end;
val Var2 (exp: Exp): Exp list =
  let val (l,e) = BindingDestr (SelectBinding (2,exp)) in l end;
val Var3 (exp: Exp): Exp list =
  let val (l,e) = BindingDestr (SelectBinding (3,exp)) in l end;
```

Functions for manipulating universes.

```
val rec Level (exp: Exp): int =
  if IsUniverse (exp) then UnivNum (exp) else Level (TypeOf exp);
val PWT (exp: Exp): string =
  implode [Print exp; ":"; Print (TypeOf exp)];
val PrintBinding b =
  let val (l,e) = BindingDestr b in Print e end;
val rec ListMax l =
  if null l then 1
  else
    let val m = ListMax (tl l) in
    if hd l<m then m else hd l
    end;
val LevelBinding b =
  let val (l,e) = BindingDestr b in
    Max (ListMax (map (Level o TypeOf) l), (Level o TypeOf) e)
  end;
val MaxUniv (e:Exp) = Universe (ListMax (map LevelBinding (Repr e)));
val ConstantType (t: Exp) = (fun e:Exp. t)
val DefaultPrint (exp: Exp): string =
  implode [ RuleName exp; "[";
            (SeparatedList PrintBinding "," (Repr exp)); "]"];
```

Appendix B

Summary of the AHA rules

In this appendix we list all the rules in the AHA programming logic.

The rules for the connectives.

FalseElim (*prex*: **False**, A): A

AndIntro ($prex_1 : A$, $prex_2 : B$): **And** (A,B)

AndElimR (*prex*: **And**(A,B)): A

AndElimL (*prex*: **And**(A,B)): B

OrIntroR ($prex : A$, B): **Or** (A, B)

OrIntroL (A, $prex : B$): **Or** (A, B)

OrElim ($prex_1$: **Or**(A,B), $prex_2$: **Imply**(A,C), $prex_3$: **Imply**(B,C)): C

ImpIntro ($hyp : A$, $prex : B$): **Imply** (A, B)

ImpElim ($prex_1$: **Imply**(A,B), $prex_2 : A$): B

Truth: **True**

PeanoPostulate7 (*prex*: **Equal** (n', m')): **Equal** (n,m)

We use x' as an abbreviation for **Application** (**SuccFunc**, x).

PeanoPostulate8 (n : **Nat**): **Not** (**Equal** (n', **Zero**))

Rules for the quantifiers.

AllIntro (v, $prex : P(v)$): **All** $(v, P(v))$

AllElim ($prex$: All (v, $P(v)$), t): P(t)

SomeIntro (Some(x,P), $prex : P(x_0)$, x_0): Some(x,P)

SomeElim ($prex_1$: Some(x,P), x_0, $shyp : P(x_0)$, $prex_2 : Q$): Q

Induction (All(m,P), $prex_1 : P(0)$, n, $ihyp : P(n)$, $prex_2 : P(n')$): All(m,P)

Recursion ($prex_1 : T$, n, $ihyp : T$, $prex_2 : T$): Arrow(Nat,T)

The equality rules.

Reflexivity (n): Equal (n, n)

Symmetry ($prex$: Equal (n, m)): Equal (m, n)

Transitivity ($prex_1$: Equal(n,m), $prex_2$: Equal(m,p)): Equal (n, p)

Congruence ($prex_1$: Equal(f,g), $prex_2$: Equal(x,y)): Equal($f(x)$,$g(y)$)

Substitution (exp_1 : Equal (x,y), $exp_2 : P(x)$): $P(y)$

EqualityElim ($prex_1$: Equal (P, Q), $prex_2 : Q$): P

Abbreviations.

FalseIntro ($prex$: And(A,Not(A))): False

NotIntro ($hyp : A$, $prex$: False): Not (A)

Now come the reductions. For each reduction there is a corresponding computation rule.

AndElimR (AndIntro ($prex_1$, $prex_2$)) $\longmapsto$ $prex_1$

AndElimL (AndIntro ($prex_1$, $prex_2$)) $\longmapsto$ $prex_2$

Application (Abstract (x, b), t) $\longmapsto$ $[t/x]b$

AllElim (AllIntro (x, $prex$),t) $\longmapsto$ $[t/x]prex$

ImpElim (ImpIntro (hyp, $prex_1$), $prex_2$) $\longmapsto$ $[hyp/prex_2]prex_1$

OrElim (OrIntroR (A, $prex_1$), $prex_2$, $prex_3$) $\longmapsto$ ImpElim ($prex_2$, $prex_1$)

$$\texttt{OrElim (OrIntroL (}A\texttt{, }prex_1\texttt{), }prex_2\texttt{, }prex_3\texttt{)} \quad \longmapsto \quad \texttt{ImpElim (}prex_3\texttt{, }prex_1\texttt{)}$$

$$\texttt{SomeElim (SomeIntro (}S\texttt{, }prex_1\texttt{, }t\texttt{), }y_0\texttt{, }shyp\texttt{, }prex_2\texttt{)}$$
$$\longmapsto \quad [y_0/t]([shyp/prex_1]prex_2)$$

$$\texttt{AllElim (Induction (}A\texttt{, }prex_1\texttt{, }n\texttt{, }ihyp\texttt{, }prex_2\texttt{), Zero)} \quad \longmapsto \quad prex_1$$

$$\texttt{AllElim (Induction (}A\texttt{, }prex_1\texttt{, }n\texttt{, }ihyp\texttt{, }prex_2\texttt{), Succ }x\texttt{)}$$
$$\longmapsto \quad [n/x]([ihyp/\texttt{AllElim (Induction (...))}]prex_2)$$

Appendix C

Summary of the TT rules

In this appendix we list all the rules in the TT programming logic.

VoidElim (exp: **Void**, A): A

Pair ($exp_1 : A$, $exp_2 : B$): **Cross** (A,B)

First (exp: **Cross**(A,B)): A

Second (exp: **Cross**(A,B)): B

InR ($exp : A$, B): **Union** (A, B)

InL (A, $exp : B$): **Union** (A, B)

Out (exp_1 : **Or**(A,B), exp_2 : **Arrow**(A,C), exp_3): C

It: **Unit**

Rules for the dependent arrow and sum types.

PiIntro ($x : A$, $exp : B(x)$): **Pi** ($x : A$, $B(x)$)

PiElim (exp: **Pi** ($x : A$, $B(x)$), $t : A$): **B(t)**

SigmaIntro (**Sigma**($x : A$,B), $exp : B(x_0)$, $x_0 : A$): **Sigma**($x : A$,B)

SigmaElim (exp_1 : **Sigma**($x : A$,B), $x_0 : A$, $y : B(x_0)$, $exp_2 : Q$): Q

Reflexivity (n): **Equal** (n, n)

Symmetry (exp: **Equal** (n, m)): **Equal** (m, n)

Transitivity (exp_1 : **Equal**(n,m), exp_2 : **Equal**(m,p)): **Equal** (n, p)

Congruence ($exp_1 : f = g$, $exp_2 : x = y$): $f(x) = g(y)$

Substitution (exp_1 : **Equal** (x,y), $exp_2 : P(x)$): $P(y)$

EqualityElim (exp_1 : **Equal** (P, Q), $exp_2 : Q$): P

The computation rules.

BetaReduction (App (Lambda (v, b), x)): **Equal** $(f(x), [v/x]b)$

BaseCaseReduction (App (Rec $(b, \ldots)$, **Zero**)): **Equal** $(\ldots, b)$

Abbreviations.

```
val Arrow (A,B) =
  Pi (x, B) where val x = NewVariable A end;
```

Now come the reduction rules.

First (Pair $(prex_1, prex_2)$) $\longmapsto$ $prex_1$

Second (Pair $(prex_1, prex_2)$) $\longmapsto$ $prex_2$

Application (Abstract (x, b), t) $\longmapsto$ $[t/x]b$

Out (InR $(A, prex_1)$, $prex_2$, $prex_3$) $\longmapsto$ **Application** $(prex_2, prex_1)$

Out (InL $(A, prex_1)$, $prex_2$, $prex_3$) $\longmapsto$ **Application** $(prex_3, prex_1)$

SigmaElim (SigmaIntro $(S, prex_1, t)$, x_0, y, $prex_2$)

$\longmapsto$ $[x_0/t]([y/prex_1]prex_2)$

Appendix D

Summary of the IZF rules

In this appendix we list all the rules of the IZF programming logic.

FalseElim ($prex$: **False**, A): A

AndIntro ($prex_1 : A$, $prex_2 : B$): **And** (A,B)

AndElimR ($prex$: **And**(A,B)): A

AndElimL ($prex$: **And**(A,B)): B

OrIntroR ($prex : A$, B): **Or** (A, B)

OrIntroL (A, $prex : B$): **Or** (A, B)

OrElim ($prex_1$: **Or**(A,B), $prex_2$: **Imply**(A,C), $prex_3$:**Imply**(B,C)): C

ImpIntro ($hyp : A$, $prex : B$): **Imply** (A, B)

ImpElim ($prex_1$: **Imply**(A,B), $prex_2 : A$): B

Truth: **True**

Rules for the quantifiers.

AllIntro (v, $prex : P(v)$): **All** (v, $P(v)$)

AllElim ($prex$: **All** (v, $P(v)$), t): P(t)

SomeIntro (**Some**(x,P), $prex : P(x_0)$, x_0): **Some**(x,P)

SomeElim ($prex_1$: **Some**(x,P), x_0, $shyp : P(x_0)$, $prex_2 : Q$): Q

Extensionality.

Extensionality $(prex_1 : \texttt{Subset}(S,T),\ prex_2 : \texttt{Subset}\ (T,S)) : \texttt{Equal}(S,T)$

The rules for the formation of the unordered pair.

PairIntro $(prex : \texttt{Or}\ (S = T,\ S = U)) :\ S \in \{T, U\}$

PairElim $(prex : S \in \{T, U\}) :\ \texttt{Or}\ (S = T,\ S = U)$

The rules for the formation of the infinite union.

UnionIntro $(prex : Some\ x \in S\ .\ T \in x) :\ T \in \bigcup(S)$

UnionElim $(prex : T \in \bigcup(S)) :\ Some\ x \in S\ .\ T \in x$

The rules for the formation of the power set.

PowerIntro $(prex : S \subset T) :\ S \in \mathcal{P}(T)$

PowerElim $(prex : S \in \mathcal{P}(T)) :\ S \subset T$

The rules for use of separation.

SepIntro $(x,\ P(x),\ prex_1 : S \in T,\ prex_2 : P(S)) :\ S \in \{x \in T \mid P(x)\}$

SepElim $(prex : S \in \{x \in T \mid P(x)\}) :\ \texttt{And}\ (S \in T,\ P(S))$

The rule for induction.

Induction $(prex : All\ x\ .\ ((All\ y \in x\ .\ P(y)) \Rightarrow P(x))) :\ All\ x\ .\ P(x)$

The rules for infinity and the empty set.

EmptyIn: $\emptyset \in \omega$

EmptySet $(S : \texttt{Set}) :\ \texttt{Not}\ (S \in \emptyset)$

Infinity $(prex : S \in \omega) :\ S' \in \omega$

Here the successor operator (denoted x') is the usual function: $x' = x \cup \{x\}$. The preceding definition requires the definition of the union of two sets ($S \cup T = \bigcup\{S, T\}$) and of the singleton set ($\{S\} = \{S, S\}$).

The last rule is the rule of collection.

Coll $(S : \texttt{Set},\ x,\ E(x) : \texttt{Set},\ P(x))$

CollIntro $(x,\ E(x)\text{: Set},\ P(x),\ prex_1 : S \in T,\ prex_2 : P(S))$

The **CollIntro** construct is a proof of:

$$E(S) \in \texttt{Coll}\ (T,\ x,\ E(x),\ P(x))$$

CollElim $(prex_1 : U \in \texttt{Coll}\ (T,\ x,\ E(x),\ P(x)),$

$z,\ hyp_1 : z \in T,\ hyp_2 : P(z),\ hyp_3 : U = E(z) \text{: } prex_2 : Q) \text{: } Q$

The equality rules.

Reflexivity (n): **Equal** $(n,\ n)$

Symmetry $(prex$: **Equal** $(n,\ m))$: **Equal** $(m,\ n)$

Transitivity $(prex_1$: **Equal**(n,m), $prex_2$: **Equal**$(m,p))$: **Equal** $(n,\ p)$

Substitution $(exp_1$: **Equal** (x,y), $exp_2 : P(x))$: $P(y)$

EqualityElim $(prex_1$: **Equal** $(P,\ Q)$, $prex_2 : Q)$: P

Abbreviations.

FalseIntro $(prex$: **And**$(A$,**Not**$(A)))$: **False**

NotIntro $(hyp : A,\ prex$: **False**$)$: **Not** (A)

Now we present the reduction rules.

AndElimR (**AndIntro** $(prex_1,\ prex_2)) \longmapsto prex_1$

AndElimL (**AndIntro** $(prex_1,\ prex_2)) \longmapsto prex_2$

AllElim (**AllIntro** $(x,\ prex)$,t) $\longmapsto [x/t]prex$

ImpElim (**ImpIntro** $(hyp,\ prex_1),\ prex_2) \longmapsto [hyp/prex_2]prex_1$

OrElim (**OrIntroR** $(A,\ prex_1),\ prex_2,\ prex_3) \longmapsto$ **ImpElim** $(prex_2,\ prex_1)$

OrElim (**OrIntroL** $(A,\ prex_1),\ prex_2,\ prex_3) \longmapsto$ **ImpElim** $(prex_3,\ prex_1)$

SomeElim (**SomeIntro** $(S,\ prex_1,\ t),\ y_0,\ shyp,\ prex_2)$

$\longmapsto [y_0/t]([shyp/prex_1]prex_2)$

Now the reduction rules for sets.

PairElim (**PairIntro** (*prex*)) $\longmapsto$ *prex*

UnionElim (**UnionIntro** (*prex*)) $\longmapsto$ *prex*

PowerElim (**PowerIntro** (*prex*)) $\longmapsto$ *prex*

SepElim (**SepIntro** (s, $P(x)$, $prex_1$, $prex_2$)) $\longmapsto$ **AndIntro** ($prex_1$, $prex_2$)

AllElim (**Induction** (*prex*), S: **Set**)

$\longmapsto$ **ImpElim** (**AllElim** (*prex*, S), **AllIntro** (y

ImpIntro ($a : y \in S$, **AllElim** (**Induction** (*prex*), y))))

We note that S must not be a variable or this reduction would never terminate.

Bibliography

[Ackermann 1928]
Ackermann, Wilhelm. "Zum Hilbertschen Aufbau der reellen Zahlen." *Mathematische Annalen*, volume 99, 1928, pages 118–133.

[Allen 1978]
Allen, John. *Anatomy of LISP.* McGraw-Hill, New York, 1978.

[Anderson and Johnstone 1962]
Anderson, John M. and Henry W. Johnstone, Jr. *Natural Deduction: The Logical Basis of Axiom Systems.* Wadsworth, Belmont, California, 1962.

[Bates 1979]
Bates, Joseph L. A Logic for Correct Program Development. Ph.D. dissertation, Department of Computer Science, Cornell University, 1979.

[Bates and Constable 1983]
Bates, Joseph L. and Robert L. Constable. Proofs as programs. TR 82-530, Department of Computer Science, Cornell University, March 1983.

[Beeson 1978]
Beeson, Michael J. "Some relations between classical and constructive mathematics." *Journal of Symbolic Logic*, volume 43, number 2, June 1978, pages 228–246.

[Bell 1937]
Bell, Eric Temple. *Men of Mathematics.* Simon and Schuster, New York, 1937.

[Bentley 1984]
Bentley, Jon Louis. "Algorithm design techniques." *Comm. of the ACM*, volume 27, number 9, September 1984, pages 865–871.

[Bishop 1970]
Bishop, Errett. "Mathematics as a numerical language." In *Intuitionism and Proof Theory*, edited by Akiko Kino, John R. Myhill and Richard Vesley, North-Holland, Amsterdam, 1970, pages 53–71.

[Bundy 1983]
Bundy, Alan. *The Computer Modelling of Mathematical Reasoning.* Academic Press, London, 1983.

[Burstall et al. 1971]
Burstall, Rodney Martineau, J. S. Collins and Robin J. Popplestone. *Programming in POP-2.* Edinburgh University Press, Edinburgh, 1971.

[Burstall and Lampson 1984]
Burstall, Rodney Martineau and B. Lampson. "A kernel language for abstract data types and modules." In *Semantics of Data Types*, edited by Gilles Kahn, David B. MacQueen and Gordon

Plotkin, Lecture Notes in Computer Science, volume 173, Springer-Verlag, Berlin, 1984, pages 1–50.

[Burstall et al. 1980]
Burstall, Rodney Martineau, David B. MacQueen and Don T. Sannella. "HOPE: An experimental applicative language." In *Conference Record of the 1980 LISP Conference*, Stanford, 1980, pages 136–143.

[Cantor 1874]
Cantor, Georg. "Über eine Eigenschaft des Inbegriffs aller reellen algebraischen Zahlen." *Journal für die reine und angewandte Mathematik*, volume 77, 1874, pages 258–262.

[Cardelli 1983]
Cardelli, Luca. "ML under UNIX." *Polymorphism: The ML/LCF/HOPE Newsletter*, volume 1, number 3, December 1983.

[Clocksin and Mellish 1981]
Clocksin, William F. and Christopher S. Mellish. *Programming in PROLOG*. Springer-Verlag, Berlin, 1981.

[Colmerauer et al. 1972]
Colmerauer, Alain, H. Kanuoi, R. Pasero and P. Roussel. Un système de communication homme-machine en français. Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Université d'Aix-Marseilles, 1972.

[Constable 1971]
Constable, Robert L. "Constructive mathematics and automatic program writers." In *Proceedings IFIP Congress*, North-Holland, Amsterdam, 1971, pages 229–233.

[Constable 1983]
Constable, Robert L. Constructive mathematics as a programming logic I: Some principles of theory. TR 83-554, Cornell University, May 1983.

[Constable 1984]
Constable, Robert L. "Mathematics as programming." In *Logics of Programs*, edited by Edmund Melson Clarke, Jr. and Derek C. Oppen, Lecture Notes in Computer Science, volume 164, Springer-Verlag, Berlin, 1984, pages 116–128.

[Constable and Bates 1984]
Constable, Robert L. and Joseph L. Bates. The nearly ultimate PEARL. TR 83-551, Cornell University, January 1984.

[Constable et al. 1982]
Constable, Robert L., Scott David Johnson and Carl D. Eichenlaub. *An Introduction to the PL/CV2 Programming Logic*. Lecture Notes in Computer Science, volume 135. Springer-Verlag, Berlin, 1982.

[Constable et al. 1984]
Constable, Robert L., Todd B. Knoblock and Joseph L. Bates. Writing programs that construct proofs. TR 84-645, Department of Computer Science, Cornell University, October 1984.

[Constable and Mendler 1985]
Constable, Robert L. and Nax Paul Mendler. Recursive definitions in type theory. TR 85-659, Cornell University, January 1985.

[Constable and O'Donnell 1978]
Constable, Robert L. and Michael J. O'Donnell. *A Programming Logic*. Winthrop, Cambridge, Massachusetts, 1978.

[Constable and Zlatin 1984]
Constable, Robert L. and Daniel R. Zlatin. "The Type Theory of PL/CV3." *ACM Transactions on Programming Languages and Systems*, volume 6, number 1, January 1984, pages 94–117.

[Demers and Donahue 1980]
Demers, Allen J. and James E. Donahue. "Data types, parameters and type checking." In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, 1980, pages 12–23.

[Dummett 1977]
Dummett, Michael A. E. *Elements of Intuitionism.* Oxford Logic Series. Oxford University Press, London, 1977.

[Friedman 1978]
Friedman, Harvey. "Classically and intuitionistically provable recursive functions." In *Higher Set Theory*, Lecture Notes in Mathematics, volume 669, Springer-Verlag, Berlin, 1978, pages 21–27.

[Gelernter 1963]
Gelernter, Herbert. "Realization of a geometry-theorem proving machine." In *Computers and Thought*, edited by Edward A. Feigenbaum and Julian Feldman, McGraw-Hill, New York, 1963, pages 134–152.

[Gödel 1958]
Gödel, Kurt. "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes." *Dialectica*, volume 12, number 3, 1958, pages 280–287.

[Goodman and Myhill 1975]
Goodman, N. and John R. Myhill. "Choice implies excluded middle." *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, volume 24, 1975, page 461.

[Gordon 1982]
Gordon, Michael J. C. "Representing a logic in the LCF metalanguage." In *Tools and Notions for Program Construction*, edited by D. Néel, Cambridge University Press, Cambridge, England, 1982, pages 163–185.

[Gordon et al. 1979]
Gordon, Michael J. C., R. Milner and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Lecture Notes in Computer Science, volume 78. Springer-Verlag, Berlin, 1979.

[Gries 1982]
Gries, David. "A note on a standard strategy for developing loop invariants and loops." *Science of Computer Programming*, volume 2, 1982, pages 207–214.

[Harper]
Harper, Robert William, Jr. Forthcoming Ph.D. dissertation, Cornell University.

[Heyting 1930a]
Heyting, Arend. "Die formalen Regeln der intuitionistischen Logik." *Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse*, number II, 1930, pages 42–56.

[Heyting 1930b]
Heyting, Arend. "Die formalen Regeln der intuitionistischen Mathematik II." *Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse*, number II, 1930, pages 57–71.

[Heyting 1930c]
Heyting, Arend. "Die formalen Regeln der intuitionistischen Mathematik III." *Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse*, number X-XII, 1930, pages 158–169.

[Horn 1951]
Horn, Alfred. "On sentences which are true of direct unions of algebras." *Journal of Symbolic Logic*, volume 16, number 1, March 1951, pages 14–21.

[Howard 1970]
Howard, William A. "Assignment of ordinals to terms for primitive recursive functionals of finite type." In *Intuitionsim and Proof Theory*, edited by Akiko Kino, John R. Myhill and Richard Vesley, North-Holland, Amsterdam, 1970, pages 443–458.

[Howard 1980]
Howard, William A. "The formulas-as-type notion of construction." In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, edited by Jonathan Paul Seldin and James Roger Hindley, Academic Press, London, 1980, pages 479–490.

[Huet 1975]
Huet, Gérard P. "A unification algorithm for typed λ-calculus." *Theoretical Computer Science*, volume 1, number 1, June 1975, pages 27–58.

[Huet and Oppen 1980]
Huet, Gérard P. and Derek C. Oppen. "Equations and Rewrite Rules." In *Formal Languages: Perspectives and Open Problems*, edited by Ronald V. Book, Academic Press, New York, 1980.

[Johnson 1980]
Johnson, Scott David. A Computer System for Checking Proofs. Ph.D. dissertation, Department of Computer Science, Cornell University, November 1980.

[Kennedy and Schwartz 1975]
Kennedy, K. and Jacob T. Schwartz. "An introduction to the set theoretical language SETL." *Comput. Math. Appl.*, volume 1, 1975, pages 97–119.

[Kleene 1964]
Kleene, Stephen Cole. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1964.

[Krafft 1981]
Krafft, Dean Blackmar. AVID: A System for the Interactive Development of Verifiably Correct Programs. Ph.D. dissertation, Department of Computer Science, Cornell University, August 1981.

[Lakatos 1976]
Lakatos, Imre. *Proofs and Refutations: The Logic of Mathematical Discovery*. Edited by John Worrall and Elie Zahar. Cambridge University Press, Cambridge, England, 1976.

[Landin 1966]
Landin, Peter J. "The next 700 programming language." *Comm. of the ACM*, volume 9, number 3, March 1966, pages 157–164.

[Lenat 1982]
Lenat, Douglas B. "AM: Discovery in mathematics as heuristic search." In *Knowledge-based Systems in Artificial Intelligence*, McGraw-Hill, New York, 1982, pages 1–225.

[Luckham and Suzuki 1979]
Luckham, David C. and Norihisa Suzuki. "Verification of array, record, and pointer operations

in PASCAL." *ACM Transactions on Programming Languages and Systems*, volume 1, number 2, October 1979, pages 226–244.

[McCarthy 1962]
McCarthy, John. "Towards a mathematical science of computation." In *Information Processing 1962*, edited by Cicely M. Popplewell, North-Holland, Amsterdam, 1963, pages 21–28.

[McCarty 1984]
McCarty, David Charles. Realizability semantics for intuitionistic formal systems. Ph.D. dissertation, Oxford University, 1984.

[McNaughton 1982]
McNaughton, Robert. *Elementary Computability, Formal Languages, and Automata*. Prentice-Hall, Englewood Cliffs, 1982.

[Manna 1974]
Manna, Zohar. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.

[Manna and Waldinger 1980]
Manna, Zohar and Richard Waldinger. "A deductive approach to program synthesis." *ACM Transactions on Programming Languages and Systems*, volume 2, number 1, January 1980, pages 90–121.

[Martin-Löf 1982]
Martin-Löf, Per. *Constructive Mathematics and Computer Programming*. Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science, Hannover, 22-29 August 1979. Edited by Cohen. North-Holland, Amsterdam, 1982.

[Milner 1972]
Milner, Robin. "Implementation and applications of Scott's logic for computable functions." In *Proc. ACM Conf. on Proving Assertions About Programs*, January 1972, pages 1–6.

[Milner 1979]
Milner, Robin. "LCF: A way of doing proofs with a machine." In *Mathematical Foundations of Computer Science 1979*, edited by Jiří Bečvář, Lecture Notes in Computer Science, volume 74, Springer-Verlag, Berlin, 1979, pages 146–159.

[Milner]
Milner, Robin. The standard ML core language. Forthcoming.

[Myhill 1973]
Myhill, John R. "Embedding classical logic in intuitionistic logic." *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, volume 19, 1973, pages 93–96.

[Nelson and Oppen 1980]
Nelson, Greg and Derek C. Oppen. "Fast decision procedures based on congruence closure." *J. of the ACM*, volume 27, number 2, April 1980, pages 356–364.

[Nordström 1981]
Nordström, Bengt. "Programming in constructive set theory: Some examples." In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, ACM, New York, 1981, pages 141–153.

[Paulson 1983]
Paulson, Lawrence. "The revised logic PPLAMBDA: A reference manual." *Polymorphism: The ML/LCF/HOPE Newsletter*, volume 1, number 2, April 1983.

[Polya 1954]
Polya, George. *Induction and Analogy in Mathematics: A Guide to the Art of Plausible Reasoning.* Mathematics and Plausible Reasoning, volume 1. Princeton University Press, Princeton, New Jersey, 1954.

[Prawitz 1965]
Prawitz, Dag. *Natural Deduction: A Proof Theoretical Study.* Almqvist and Wiksell, Stokholm, 1965.

[PRL Staff 1984]
PRL Staff. *PRL: Proof Refinement Logic Programmer's Manual (Lambda PRL VAX Version).* Computer Science Department, Cornell University, 1984.

[Putnam 1967]
Putnam, Hilary. "The thesis that mathematics is logic." In *Bertrand Russell: Philosopher of the Century*, edited by Ralph Schoenman, George Allen & Unwin, London, 1967, pages 273–303.

[Reps and Alpern 1984]
Reps, Thomas and Bowen Alpern. "Interactive proof checking." In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1984, pages 36–45.

[Reynolds 1984]
Reynolds, John C. "Types, abstraction and parametric polymorphism." In *Information Processing 83*, edited by R. E. A. Mason, North-Holland, 1984, pages 513–523.

[Sasaki]
Sasaki, James T. Forthcoming Ph.D. dissertation, Cornell University.

[Scott 1970]
Scott, Dana S. "Outline of a mathematical theory of computation." In *Proc. 4th Annual Princeton Conf. on Information Sciences & Systems*, Princeton, 1970, pages 169–176.

[Stenlund 1972]
Stenlund, Sōren. *Combinators, λ-terms, and Proof Theory.* D. Reidel Publishing Company, Dordrecht, Holland, 1972.

[Stoy 1977]
Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.

[Suppes 1981]
Suppes, Patrick, editor. *University-Level Computer-Assisted Instruction at Stanford: 1968–1980.* Institute for Mathematical Studies in the Social Sciences, Stanford, 1981.

[Teitelbaum and Reps 1984]
Teitelbaum, Tim and Thomas Reps. The synthesizer generator reference manual. TR 84-619, Cornell University, 1984.

[Tennent 1976]
Tennent, R. D. "The denotational semantics of programming languages." *Comm. of the ACM*, volume 19, number 8, August 1976, pages 437–453.

[Troelstra 1973]
Troelstra, Anne Sjerp, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Lecture Notes in Mathematics, volume 344. Springer-Verlag, Berlin, 1973.

[van Emden and Kowalski 1976]
van Emden, M. H. and Robert A. Kowalski. "The semantics of predicate logic as a programming language." *J. of the ACM*, volume 23, number 4, October 1976, pages 733–742.

[Wos 1982]
Wos, L. "Solving open questions with an automatic theorem-proving program." In *6th Conference on Automated Deduction*, edited by Donald W. Loveland, Springer-Verlag, 1982, pages 1–31.