Writing Programs That Construct Proofs*

R.L. Constable
T.B. Knoblock
J.L. Bates

TR 84-645
October 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

* This research supported in part by the National Science Foundation under grant MCS-81-04018.

WRITING PROGRAMS THAT CONSTRUCT PROOFS

R.L. Constable
T.B. Knoblock
J.L. Bates

TR 84-645
Draft of 15 October 1984.

Department of Computer Science
Cornell University
Ithaca, New York 14850

This research supported in part by the National Science Foundation under grant MCS-81-04018.

ABSTRACT

When we learn mathematics, we learn more than definitions and theorems. We learn techniques of proof. In this paper, we describe a particular way to express these techniques and incorporate them intro formal theories and into computer systems used to build such theories. We illustrate the methods as they were applied in the λ-PRL system, essentialy using the ML programming language from Edinburgh LCF [23] as the formalised metalanguage. We report our experience with such an approach emphasizing the ideas that go beyond the LCF work, such as transformation tactics, refinement tactics, and special purpose reasoners. We also show how the validity of tactics can be guaranteed. The introduction places the work in historical context and the conclusion briefly describes plans to carry the methods further. The majority of the paper presents the λ-PRL approach in detail.

Contents

1 Introduction.

Statement of The Problem.

In the time of the Greeks, geometers were already building machines to help them with derivations. Continuous and sustained interest in providing mechanical aids to reasoning can be traced to the seventeenth century. Gottfried Leibniz is popularly believed to have contributed to symbolic logic in striving to mechanize reasoning, and while his technical contributions in this subject were minor, his vision and the authority which his stature accorded it are with us today. His words still kindle an interest little diminished by the naivete of their details [30].

> A term is the subject or predicate of a categorical proposition. ...
>
> Let there be assigned to any term its symbolic number, to be used in calculation as the term itself is used in reasoning. I chose numbers whilst writing; in due course I will adapt other signs ... For the moment, however, numbers are of the greatest use ... because everything is certain and determinate in the case of concepts, as it is in the case of numbers. The one rule for discovering suitable symbolic numbers is this; that when the concept of a given term is composed directly of the concept of two or more other terms, then the symbolic number of the given term should be produced by multiplying together the symbolic numbers of the terms which compose the concept of the given term. In this way we shall be able to discover and prove by our calculus at any rate all the propositions which can be proved without the analysis of what has temporarily been assumed to be prime by means of numbers we can judge immediately whether propositions presented to us are proved, and that which others could hardly do with the greatest mental labor and good fortune, we can produce with the guidance of symbols alone ... As a result of this, we shall be able to show within a century what many thousands of years would hardly have granted to mortals otherwise.

The possibility of actually carrying out such a program of analysis for a substantial body of knowledge, such as mathematics, did not exist until the appearance of the predicate calculus in G. Frege's **Begriffsschrift** [18]. Frege created a new language for writing precise thought. His reasons for doing so are exactly those which motivate this work. We defer to Frege's wording:

> In apprehending a scientific truth we pass, as a rule, through various degrees of certitude. Perhaps first conjectured on the basis of an insufficient number of particular cases, a general proposition comes to be more and more securely established by being connected with other truths through chains of inferences. ... Hence we can inquire,

on the one hand, how we have gradually arrived at a given proposition and, on the other, how we can finally provide it with the most secure foundation. The first question may have to be answered differently for different persons; the second is more definite, and the answer to it is connected with the inner nature of the proposition considered. The most reliable way of carrying out a proof, obviously, is to follow pure logic, a way that, disregarding the particular characteristics of objects, depends solely on those laws upon which all knowledge rests. ... In attempting to comply with this requirement in the strictest possible way I found the inadequacy of language to be an obstacle; no matter how unwieldy the expressions I was ready to accept, I was less and less able, as the relations became more and more complex, to attain the precision that my purpose required. This deficiency led me to the idea of the present ideography. Its first purpose, therefore, is to provide us with the most reliable test of the validity of a chain of inferences and to point out every presupposition that tries to sneak in unnoticed, so that its origin can be investigated.

A monumental effort to apply the logistic methods is the three volume 1,929 page **Principia Mathematica** by A.N. Whitehead and B. Russell [45]. Its reception in some quarters was highly favorable as we can see from this review by C.J. Keyser [28].

> Logic it is called and logic it is, the logic of propositions and functions and classes and relations, by far the greatest (not merely the biggest) logic that our planet has produced. ... Few will read it, but all will feel its effect, for behind it is the urgence and push of a magnificent past; two thousand five hundred years of record and yet longer tradition of human endeavor to think aright.

The logic of this treatise can express all of mathematics yet it is formal [1]. It is thus possible in principle to translate the proof of any mathematical theorem into a completely formal proof. However, the prospect of actually doing this is quite daunting because an informal proof of modest length will expand to a formal one of prodigious size and will require in its production extreme care and detailed knowledge of the more or less arbitrary conventions of the particular formalism. These tedious details will in sheer number dominate the interesting mathematical ideas which are the very *raisin d'être* of the proof. This state of affairs prompted some to greet Principia with far less enthusiasm than C.J. Keyser. From Henri Poincaré [35] we read "On the contrary, I find nothing in logistic for the discover but shackles. It does not help us at all in the direction

[1] **Principia Mathematica** is not completely formal in the modern sense, but could be made so.

of conciseness, far from it; and if it requires twenty seven equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?"

We see now before us the question which motivates this study. Is it in fact possible to formalize real mathematical argument in a useful way? Will formal proofs of important theorems always be so long that no one will read them and so tedious that there will be no point in trying? Will formal proofs ever be more than museum pieces and curiosities? Or will there be such good systems for writing and displaying formal proofs that they will become an accepted standard of rigor and will be treasured like diamonds for their strength? Will they open a realm of mathematics in which the computer will play a significant role — in checking proofs, giving advice about details, retrieving relevant facts from libraries of theorems and performing numerous other chores of a *mathematician's assistant* long before machines can help mankind in general with even the most menial *common sense* reasoning?

In this paper we will explore these questions. We begin with a look at relevant theoretical results. Most of these results arose in the context of studying the Hilbert program as a means of providing a foundation for mathematics. [26] This program was not in fact concerned with the issue which is central to us and was dominant in Frege, namely a plan to build and use an "ideography". Hilbert wanted only to study an ideography and he was immensely pleased that a (nearly) adequate language was at hand with *Principia Mathematica*.

We will see that because the major theoretical results deal with Hilbert's program, they are not as directly relevant to our program as one might naively expect but they form a background which must be addressed.

2 Theoretical Results.

2.1 Proofs as Expressions.

There is no doubt that we can adequately formalize the concept of a mathematical sentence; the notion of a formula in the predicate calculus does that. So our attention focuses on the concept of a proof. The simplest definition, used in the Hilbert program from which the term *Hilbert style proof* arises, is that a proof is a sequence of formulas each of which is either an axiom or follows by a rule of inference from previous formulas in the sequence. A typical axiom would be presented as $P \vee \neg P$ and a typical rule of inference would be presented as

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

meaning that if A and $A \Rightarrow B$ are previous lines in the proof, then B can be added as a new proved line.

Although the definition of proof is stated in terms of a sequence in analogy to the way proofs presented on paper, a somewhat more abstract account arises if we make the algebraic structure more explicit. To this end we can characterize a proof as an expression built from constant terms, called axioms, and from unary and binary operators. For example, if the above rule of *modus ponens* is represented by the operator MP, then a proof of B from A and $A \Rightarrow B$ might be written as $MP(a_1, a_2)$, where a_1 names A and a_2 names $A \Rightarrow B$. In this account, proofs are treated like algebraic expressions and their individual structure is tree–like. The class of proofs is still defined inductively.

This concept of proof is simple, and it captures the inductive character of the concept, but it is not an adequate representation of proofs as they actually occur in mathematics. This is true for many reasons, and to find an adequate notion of proof we must unravel the inadequacies one by one and sort out how to assemble a sufficiently adequate account. That problem will be the principal concern of the next subsection where we shall examine some new ideas. At this point we bring into the account the best representation of proofs known from the early work of logicians, especially of G. Gentzen.

We see in actual mathematics *arguments from assumptions*. For instance to prove $A \Rightarrow B \Rightarrow A$ we say "assume A is true, then we will prove $B \Rightarrow A \ldots$." Gentzen [19] analyzed such arguments and discovered the Calculus of Natural Deduction. It is interesting that proofs in this calculus can also be presented as algebraic expressions (inductively defined) if we take as primitive a slightly more general concept than that of a formula. We take instead the idea of a *sequent* as Gentzen called it, which has the form $A_1, \ldots, A_n \vdash B$ in our case, which is to be read "from the assumptions $A_1, \ldots, A_n$, the conclusion B follows". The meaning is similar to that of the formula $A_1 \& \ldots \& A_n \Rightarrow B$, but the syntax of sequents favors a class of operations (inference rules) which would be awkward to state for formulas. The use of sequents moves some of the detail of the deductive machinery out of the object language, from the concept of formula, and into the metalanguage.

Here is how the rules of proof are stated in terms of sequents and operations on them. The analogue of an axiom scheme is a certain kind of sequent, namely one of the form $A_1, \ldots, A_n \vdash A_i$ for $1 \leq i \leq n$. These are the primitive or atomic sequents. The rule of *modus ponens* can be expressed in several forms. One form is

$$\frac{L_1 \vdash A \quad L_2 \vdash A \Rightarrow B}{L_1 \cup L_2 \vdash B}$$

where L_1 and L_2 are lists of formulas and $L_1 \cup L_2$ is their "union". Another form is

$$\frac{L_1, A \Rightarrow B, L_2 \vdash G \quad L_3 \vdash A}{L_1, B, L_2, L_3 \vdash G}$$

Either of these rules can be represented by an expression of the form MP(a,b) where a and b denote sequents of the appropriate form and MP as a binary operator on proofs.

In PRL proofs are defined in terms of sequents, but in addition the inference rules are presented in a top-down style with the goal first and the subgoals under it. Thus the PRL rule for *modus ponens* is in fact written as

1. $H_1, \ldots, n.A \Rightarrow B, \ldots, m.H_m \vdash G$ **by elim** n
   1. $H_1, \ldots, H_m \vdash A$
   2. $H_1, \ldots, H_m, m+1.A, m+2.B \vdash G$

As a proof expression, we might write an application of this rule as elim(n)(h_1,h_2) where h_1 denotes the first subgoal and h_2 the second.

Here is an example of a complete PRL proof of the formula $A \& A \Rightarrow B \Rightarrow B$.

```
⊢ A & (A ⇒ B) ⇒ B  by intro
1. A & (A ⇒ B)  ⊢  B   by elim 1
   1.  A   2.  A ⇒ B  ⊢  B   by elim 2
             1.  A,  2.  A ⇒ B  ⊢  A   by hyp 1
             1.  A,  2.  A ⇒ B,  3.  B  ⊢  B   by hyp 3
```

An algebraic expression for this proof is intro(elim(1)(elim(2)(hyp 1, hyp 3))). This expression together with the goal formula completely determines the proof as we shall see later.

2.2 Formal Versus Informal Proofs.

The idea of a proof in mathematics is exceedingly subtle. Even in a very rigorous style of mathematics it is quite obvious that what passes for a proof is not the simple kind of formal algebraic structure outlined in the previous subsection. In this subsection we want to examine some of the features of informal proofs that significantly distinguish them from formal proofs as they are now known.

Certain arguments rely heavily on our intuition, perhaps on geometric or physical intuition or on a deep understanding of natural phenomena.

We do not attempt to treat such arguments here and we will not call them proofs. We will rule them out by insisting on the concept of a purely mathematical proof. This is a concept which in some form we owe to the Greeks.

In the realm of purely mathematical proofs we can recognize the step by step character of the formal proofs defined above, and we agree that the formal algebraic structure is necessary to understand informal proofs. But it is not clear that all proof concepts can be reduced to these. Anyone with experience with real mathematical proofs will recognize other kinds of justification, many of them signaled by the phrases like "it is obvious". This justification of a proof step can hide a great deal of reasoning of various kinds such as:

1. Application of trivial steps of logic, e.g., noticing from $(A \Rightarrow B) \Rightarrow C$, $\neg B \Rightarrow C$, and $B \vee \neg B$, that C will follow,
2. application of basic rules in some familiar domain such as arithmetic, e.g. noticing that $1 \leq a \leq b \leq 1$ means $a = b = 1$,
3. recall of some well-known fact and some well-known way to apply that key fact, e.g. noticing that any number can be uniquely factored into a product of primes,
4. observation that a proof technique just successfully applied to a formula will apply with *minor modifications* to the formula at hand.

We believe that we know how to formalize the methods of reasoning of the first two kinds. In the first case we use an algorithm which carries out trivial or immediate reasoning as reported in work on PL/CV [12] or as summarized in sections 4 and 5 here. In the second case we know of a variety of decision procedures which appear to capture this style of reasoning.

For categories 3 and 4 we will make proposals in later sections, but unlike for the first two categories we do not have experimental evidence that we are on the right track. However, it can be argued that methods in these later two categories do not constitute rigorous proof but rather informal proof discovery heuristics. So if we restrict the concept of proof still further to include only rigorous purely mathematical proof, then there is reason to believe that we know how to formalize it.

One way to assess the effectiveness of the techniques used to formalize proofs is to measure the length of the complete and checked formal proof as a function of the length of a carefully written down rigorous informal proof. It is widely believed that existing techniques of formalization are ineffective because the formal proofs are much longer. But what is not appreciated is that the evidence thus far accumulated is that there is (at worst) a linear relationship between the two lengths, say cn where n is the length of the informal proof and c is a constant. In early proof checking systems such as AUTOMATH [14] the constant c was rather large, around 50. In systems like PL/CV [12] the constant was smaller because more elaborate techniques such as decision procedures and automatic rules were used. We will see here that the method of tactics can lower the constant even more. Indeed, if our ideas for formalizing the methods of categories 3 and 4 succeed, then we can imagine that formal machine checked proofs will actually be shorter than their informal counter parts.

We will see in subsequent sections which ideas are essential to preserving a small linear relationship between the size of formal and informal proofs. We will also see how these empirical discoveries were made and will quote from the original reports of them to emphasize their significance.

2.3 Metamathematical Results.

Our claim that we can formalize the concept of proof appears to contradict results from metamathematics such as Gödel's incompleteness theorem[21]. How can we discuss the size relationship between formal and informal proofs in number theory when there are statements of number theory which are not provable in any fixed formal system yet are provable in the metatheory? The answer here is that we consider only informal proofs in some *fixed axiomatic theory*, such as Peano arithmetic. When we expand the informal theory, say by including new axioms, then we correspondingly produce a new enlarged formal theory.

There are other metamathematical results that seem equally discouraging to our enterprise. For example, Gödel [22] shows that in second–order number theory there are proofs of first–order statements that are much shorter than any first–order proof. Hartmanis [25] has considerably elaborated on this theme by proving that for any formal system F and any recursive function f there are theorems of length n which we can prove from outside of F in n steps but whose shortest proofs in F require $f(n)$ steps. Thus Hartmanis [25] concludes:

> These results show very clearly that we pay a price for formalizing mathematics. In every formalization, infinite sets of trivial theorems will require very long proofs. Thus we have a very dramatic and quantitative explanation of why we should not and in practice do not freeze a formation when doing or discussing mathematics.

What are we to make of these results? In the first place, none of them apply if we are willing to fix an axiomatic theory and compare formal and informal proofs over it. Nor do they apply if we are willing to add new axioms to the formal theory if the need arises and the axioms are widely known and accepted. Either of these positions is acceptable given the nature of our investigation. But moreoever it should be noted that these negative metamathematical results may apply only to an infinitesimally small collection of theorems which may not be of interest in the first place. It may also be the case that by allowing one simple informal metatheory level mechanism for extending a theory, e.g. adding axioms asserting consistency, these negative results can be avoided. Whatever the possibilities, it is important to know what the empirical results are. Do we in fact see in practice any sign of these limitations?

Another criticism of our approach is suggested by results on decision problems. For example it is known for any nondeterministic Turing machine recognizing the theorems of Presburger arithmetic that for every n there are theorems of length n for which the machine will require $2^{2^{cn}}$ steps. Similar results are known for other extremely basic theories.

Hartmanis [25] interprets these results as "Again the limitations of formal methods and mechanical proof procedures ... have been pointed out." De Millo, Lipton, and Perlis [16] say:

> Outsiders see mathematics as a cold, formal, logical, mechanical, monolithic process of sheer intellection; we argue that insofar as it is successful, mathematics is a social, informal, intuitive, organic, human process, a community project. Within the mathematical community, the view of mathematics as logical and formal was elaborated by Bertrand Russell and David Hilbert in the first years of this century. They saw mathematics as proceeding in principle from axioms or hypotheses to theorems by steps, each step easily justifiable from its predecessors by a strict rule of transformation, the rules of transformation being few and fixed. The *Principia Mathematica* was the crowning achievement of the formalists. It was also the deathblow for the formalist view. There is no contradiction here: Russell did succeed in showing that ordinary working proofs can be reduced to formal, symbolic deductions. But he failed, in three enormous, taxing volumes, to get beyond the elementary facts of arithmetic. He showed what can be done in principle and what cannot be done in practice. If the mathematical process were really one of strict, logical progression, we would still be counting on our fingers.

Let us see, in light of these discouraging remarks, what has in fact been accomplished.

3 Empirical Results.

We have seen that the theoretical results concerning the feasibility of adequately formalizing the concept of proof are not definitive. We have also said that there are empirical studies which appear to show that such formalization is feasible in the sense that computer systems have been written which help the user generate and check formal proofs to such an extent that people have been willing to undertake writing formal mathematics using them; they have tolerated expansion of the informal text by factors between 5 and 50. In this section we take a closer look at these empirical results by outlining a brief history of a topic we call *mechanical proof checking*. This topic blends at the edges with the subject of *automatic theorem proving* and forms part of the topic now called *automated reasoning* in the literature. There is a two volume series, *Automation of Reasoning* [39], which gives an adequate historical account and a survey of the empirical evidence in the subject of automatic theorem proving, so this brief account will focus on *proof checking*, which is directly related to our concern with the formalization of proofs, to the near exclusion of automatic theorem proving.

3.1 Early Work in A.I.

Already in 1962 John McCarthy [31] wrote about the possibility of using computers to check proofs and verify program correctness:

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

He went on to examine formal systems that admit brief proofs and outlined a proof checker to be written in Lisp.

> We envisage the use of computer proof-checking in mathematics as follows: The mathematician already has formalizations of his branch of mathematics and the computer system has stored in it the theorems that have previously been proved. In addition, there are a number of techniques embodied in programs for generating proofs.

> The mathematician expresses his ideas of how a proof may be found by combining these techniques into a program. The computer carries out the program which may prove the theorem, may generate information that will guide another try, may indicate an elementary misconception, or may be of no help whatsoever.

In 1963 Paul Abrahams [1] wrote a Ph.D. thesis at MIT under Marvin Minsky in which he describes a program to check theorems in *Principia Mathematica* [45] (about 63 tautologies were actually checked). Abrahams worked out the notion of *macro steps* as a way to let the computer build proofs from outlines of methods. He saw writing formal proofs as equivalent to writing assembly language programs and saw his accomplishment as steps toward higher level proof languages. It is interesting that in the same period, 1960–1963, Hao Wang [43] wrote a program based on a decision procedure for a fragment of the predicate calculus, that proved all the pure predicate calculus theorems of *Principia Mathematica*, (about 400 of them). This achievement drew attention to the more ambitious project of automatic theorem proving in general.

By 1969 there were two major proof checking projects underway at the Stanford A.I. Lab supported by John McCarthy; one was FOL (First Order Logic) [44] with Richard Weyhrauch, and the other was Stanford LCF (Logic for Computable Functions) with Robin Milner. Both projects are still active, the latter now at Edinburgh [23], Cambridge [34], INRIA and to some extent at Cornell (e.g., this report). We will confine our remarks to LCF since it is directly relevant to this work. The LCF group successfully proved the correctness of a small compiler [9] and built up the experience needed to define and implement Edinburgh LCF [23] about which we will have much to say in section 4.

Also at Stanford durring this period work was begun under P. Suppes to use proof-checkers in computer aided instruction. The EXCHECK system [41] has been used in teaching set theory to undergraduates at Stanford, and the relationship between proof theory and proof checking was explored extensively (C.f., [41] for articles by G. Kreisel for example).

3.2 The AUTOMATH Project.

In 1967–68 at Eindhoven Technical University, N.G. deBruijn began the AUTOMATH project [15]. "AUTOMATH is a language which we claim to be suitable for expressing very large parts of mathematics, in such a way that the correctness of the mathematical contents is guaranteed as long as the rules of the grammar are obeyed."

The AUTOMATH effort concentrated on building a very expressive language in which any mathematical statement could be stated. L.S. Jutting [27] transcribed an entire book by Edmund Landau, *Grundlagen der Analysis* (158 pages), with AUTOMATH (a major five year effort resulting in several volumes of computer checked mathematics). The lesson learned was that the job was possible; no exponential explosion in proof length appeared. Indeed, it seemed that there was a linear relationship. Here is how deBruijn [15] describes the result:

> A very important thing that can be concluded from all writing experiments is the *constancy of the loss* factor. The loss factor expresses what we lose in shortness when translating very meticulous "ordinary" mathematics into AUTOMATH. This factor may be very big, something like 10 or 20 (or *50*), but it is constant: it does not increase if we go further in the book. It would not be too hard to push the constant factor down by efficient abbreviations.

3.3 Programming Logics.

In 1976 at Cornell University we began the PL/CV project which was also concerned with proof checking. The first phase of the effort concentrated on using fast decision procedures and techniques from the theory of algorithms [2] and using the techniques of programming language and later synthesizer design [42] to produce readable proofs in algorithmic mathematics. We were able to write 1477 lines of formal constructive math fairly easily culminating in the Fundamental Theorem of Arithmetic. The proofs were algorithmic and could be efficiently executed. Students also wrote numerous isolated proved-programs from a first semester programming course.

The PL/CV effort confirmed a kind of *linearity hypothesis* for elementary proofs. But because of computer aid in generating these proofs, the *loss factor* was more like 10 rather than 50.

3.4 Spectrum of Methods to Automate Reasoning.

One can identify at least three distinct general approaches to automating reasoning. At one extreme is pure proof checking, as exemplified by AUTOMATH. At the other extreme is automatic theorem proving, as exemplified by various well-known provers [6,5,40]. In between are those approaches which rely to some extent on proof checking and on decision procedures. They might be characterized as nonheuristic theorem proving; a typical example is PL/CV. Some systems use all three strategies. Let us consider the characteristics of each method and then see how Edinburgh LCF offers the best of each approach.

Proof Checking:

The pure proof checking methods rely on a very expressive language in which to capture the abstractions that make rigorous mathematics possible. They require large libraries of results and use a minimum amount of algorithmic metamathematics. Such techniques are thus very safe but also very tedious and unexciting. They tend to use the computer the way that compilers do.

Theorem Proving:

The automatic theorem provers rely on Gödel's completeness [20] theorem. They usually code some complete proof search strategy based on the idea that to prove A one should look systematically for a model satisfying $\neg A$. Inherent in these methods is the possibility that the procedure will fail after an investment of considerable resources. Thus the methods can be very costly, but can discover unexpected results and can aid in the discovery of a proof. Current methods are based on an inherently nonconstructive semantics for the first order predicate calculus.

Decision Procedures:

The decision procedure technique frequently relys on a deep analysis of theorems and requires complex algorithms. Algorithms for simple theories such as equality, natural number arithmetic and rational arithmetic have been quite successful. These algorithms have good expected computing times. Although a great deal is known about the asymptotic intractability of many decision problems such as Presburger arithmetic and real closed fields, it is not known whether there are useful algorithms for the naturally occurring statements in these theories. In contrast to the proof checking methods, these decision procedure techniques rely on complex algorithms whose correctness must be a major factor in judging the reliability of a system using them.

LCF Idea:

The Edinburgh LCF project took the approach that one should build a system which allows experimentation with a mix of strategies along the spectrum from pure proof checking to full theorem proving. They state [23] that one of the main aims of the projects was "to provide an interactive metalanguage (ML) for conducting proofs, in which in principle almost any style can be programmed, but which provides the greatest possible security against faulty proofs."

Related Ideas:

Inherent in the LCF approach is a formalization of the metatheory to some extent. The programming language ML of LCF formalizes the syntax and proof rules of the object theory. Other schemes have been proposed for incorporating metamathematical reasoning. For example, Davis & Schwartz [13] proposed completely formalizing the metamathematics and proving that various extensions of the inference rules are correct. These can be added as new rules. Boyer & Moore [6] and Weyhrauch [44] propose a similar scheme and discuss the other options. The PL/CV project [12] proposed a scheme whereby the metamathematics of one level of the system can be reflected in the next level.

Of all of these methods we have found the LCF idea easiest to use and most powerful. We have been persuaded to study our technique in the context of the Edinburgh LCF approach.

The PRL proof generating environment attempts to achieve a low *loss factor* by a combination of techniques similar to those envisioned by McCarthy [31]:

- a proof editor for ease in generating formulas and entering new notations; the editor is oriented to a screen with mouse and windows.
- a top down logic to support goal oriented proving directly (we call these *refinement logics*).
- fast decision procedures for key subtheories: equality, lists, restricted arithmetic.
- representation of the logic in the Edinburgh LCF metalanguage, ML, and use of tactics and tacticals to provide safe user-defined extensions of the logic along the entire spectrum from decision procedures to heuristic search.
- use of functions (called transformation tactics) which convert one proof into another.

4 The Object Language and the Metalanguage.

4.1 Introduction.

It is fundamental in the study of logic to differentiate between *metatheory* and *object theory*. The object theory is that formal theory which is the subject of the metatheory. Typically the metatheory is not altogether formal; if it is, then it too has a metatheory and various questions arise about the relationship between the formal metatheory and the formal object theory. In our case the object theory is the PRL number and list theory, called λ-PRL. The metatheory contains a formal part called the metalanguage; this is the ML programming language from LCF.

In the formal metalanguage we can write programs which search for proofs or transform proofs. This section will explain how that is accomplished. We begin with a brief account of the object theory, PRL, then an account of the metatheory, ML, and finally an introduction to the concept of a tactic.

4.2 The Object Theory, λ-PRL.

Here we briefly describe the PRL logic to the extent necessary to understand the detailed structure of proofs. The key novelty here is that PRL is a refinement logic [4], that is a sequent calculus [19] oriented top down.

Syntax and Proof Rules.

The *atomic types* of the theory are *integer* and *integer list*, which are abbreviated *int* and *list* respectively.

The *terms* of the theory are *constants*, *variables*, *applications* of the form $f(e_1, \ldots, e_n)$ or e_1 op e_2 where e_1, e_2 are terms, and op is an operator, and *listings* $[e_1, ..., e_n]$ where the e_i are integer terms.

The *constants* include the natural numbers, the unary function -, the binary operators: +, -, *, /, and various atomic functions: *mod*, *hd*, *tl*, and $\cdot$ (the last of these represents cons in PRL). The list constants are [ ], $[n_1, ..., n_p]$ for n_i integers.

The function constants have the following types:

$$\begin{aligned} mod &: \mathit{int} \times \mathit{int} \rightarrow \mathit{int} \\ hd &: \mathit{list} \rightarrow \mathit{int} \\ tl &: \mathit{list} \rightarrow \mathit{list} \\ \cdot &: \mathit{int} \times \mathit{list} \rightarrow \mathit{list} \end{aligned}$$

The atomic formulas of the theory are equalities and integer inequalities.

Compound formulas are ¬A, A&B, A∨B, A⇒B for A and B formulas. The usual precedence holds among these connectives: ¬, &, ∨, ⇒, with ⇒ being right associative. Compound formulas include

$$\forall x_1, \ldots, x_n{:}type.A$$
$$\exists x_1, \ldots, x_n{:}type.A$$

where A is a formula and x_i are variables. Quantifiers bind more weakly than connectives, so they have a wide scope.

A *sequent* has the form

$$[e]\ 1.F_1, \ldots, \mathrm{n}.F_n \vdash C$$

where i.F_i are numbered formulas, C is a formula and e is an environment of the form variable list:int, variable list:list. We adopt a less detailed notation when possible, usually suppressing the environment and writing $S \vdash C$ for S a numbered sequence of formulas, or writing S,n.$F, S' \vdash C$ when we are interested only in the number of formula F at one occurrence. We also write $S, A, B \vdash C$ when we do not care what numbers are assigned to A and B but want to depict their relative order.

For T a type, either int or list, we use $[e \cup x{:}T]$ to indicate the new environment formed by adding x to the appropriate variable list, e.g., $[n, y{:}int, A{:}list \cup x{:}int] = [n, y, x{:}int, A{:}list]$.

A *proof* is an expression of the form

$$\begin{array}{c} \text{goal } \textit{by} \text{ rule–name} \\ p_1 \\ \cdot \\ \cdot \\ \cdot \\ p_n \end{array}$$

where p_i are proofs. The rule names are certain *constants* such as those listed below. It is convenient to think of the proof expression in the form $f(p_1, \ldots, p_n)$ where f is the rule name and "goal" is the range type of f viewed as a function.

The proof rules fall into five categories: (1) predicate calculus rules, (2) arithmetic rules (taken from PL/CV2 [9]), (3) list rules, (4) rules to reference the library and defined objects and, (5) rules to invoke tactics built in the metalanguage ML.

Here we illustrate some of these. All rules are presented in *refinement style*; that is the conclusion is listed first, thought of as a *goal*, and the hypotheses are listed under it, as *subgoals*. The rule name is listed after the goal. Environments are not shown if they do not change from goal to subgoals. The rule of inference for λ-PRL are sumarized in figures 1 and 2. In addition to the rules given, there is an induction rule for lists.

The λ-PRL proof editor is based upon a window display system. At each stage in editing a proof, one sequent along with the associated refinement rule and subgoals (if any) are displayed in a window. To refine a sequent, the user enters the refinement rule in another window. When finished, the system calculates and displays the subgoals. Figures 3 and 4 show sample proof editing.

&	$S \vdash A\&B$ by intro 1. $S \vdash A$ 2. $S \vdash B$	S,n.$A\&B, S' \vdash C$ by elim n 1. $S, S', A, B \vdash C$
$\vee$	$S \vdash A \vee B$ by intro 1 1. $S \vdash A$ $S \vdash A \vee B$ by intro 2 1. $S \vdash B$	S,n.$A \vee B, S' \vdash C$ by elim n 1. $S, S', A \vdash C$ 2. $S, S', B \vdash C$
$\Rightarrow$	$S \vdash A \Rightarrow B$ by intro 1. $S, A \vdash B$	S,n.$A \Rightarrow B, S' \vdash C$ by elim n 1. S,n.$A \Rightarrow B, S' \vdash A$ 2. S,n.$A \Rightarrow B, S', A, B \vdash C$
$\forall$	$[e]S \vdash \forall x{:}A.B$ by intro 1. $[e \cup x{:}A]S \vdash B$	S,n.$\forall x{:}A.B, S' \vdash C$ by elim n,t 1. $S, \forall x{:}A.B, S', B(t/x) \vdash C$
$\exists$	$S \vdash \exists x{:}A.B$ by intro t 1. $S \vdash B(t/x)$	$[e]S$,n.$\exists y{:}A.B, S' \vdash C$ by elim n 1. $[e \cup y{:}A]S, S', B \vdash C$

Note: $B(t/x)$ stands for B with t substituted for x.

consequence

$S \vdash C$ by seq T
1. $S \vdash T$
2. $S, T \vdash C$

hypothesis

S,n.$A, S' \vdash A$ by hyp n

false elimination

S,n.$false, S' \vdash C$ by elim n

Figure 1: The Basic Rules of Inference in λ-PRL.

The induction rule for integers has the following form when specialized to a base case of 0.

$[e]S \vdash \mathit{all}\ x{:}\mathit{int}.P$ by ind

1. $[e \cup x{:}\mathit{int}]\ \ S, x < 0, P(x + 1/x) \vdash P$
2. $[e]\ \ S \vdash P(0/x)$
3. $[e \cup x{:}\mathit{int}]\ \ S, x > 0, P(x - 1/x) \vdash P$

The variable x cannot already appear in the environment. If it does, the rule can be called "ind y" where y is a new variable which will be used in the hypotheses in place of x. Other forms of the rule allow other base cases to be specified.

Figure 2: The Induction Rule For Integers.

```
EDIT THM max
____________________________________
# top
[]
>> all x,y:int.some m:int.
     ¬m<x & ¬m<y & (m=x∨m=y)

BY <refinement rule>
```

```
EDIT rule of max
________________________
intro
```

Figure 3: Sample proof editing: Entering the refinement rule.

```
EDIT THM max
________________________________
# top
[]
>> all x,y:int.some m:int.
    ¬m<x & ¬m<y & (m=x∨m=y)

BY intro

1# [int x,y]
    >> some m:int.
        ¬m<x & ¬m<y & (m=x∨m=y)
```

Figure 4: Sample proof editing: the result of the refinement.

4.3 The Metalanguage, ML.

The language in which we write tactics in is the ML programming language [23]. ML is a functional programming language with three important characteristics which make it a good language for expressing tactics.

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

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

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

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

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

This function has type int→int→bool, i.e., it maps integers to functions from integers to boolean values. There is also an explicit abstraction operator, λ. The previous function could have been defined as

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

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

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

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

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

4.4 Tactics in ML.

The ML concept of tactic is a formalization of the idea of top–down heuristic problem solving. The method was systematized already by the Greeks, e.g., Pappas, and it is a key element in G. Polya's heuristic [36]. It also formed the basis for the Logic Theorist of Newell, Shaw and Simon [33]. Let us hear how tactics were presented in these various settings.

First from the Greeks, Polya [36] quotes Pappas (circa 300 BC) as follows:

> The so-called Heuristic is, to put it shortly, a special body of doctrine for the use of those who, after having studied the ordinary Elements, are desirous of acquiring the ability to solve mathematical problems and is useful for this alone. ... If we have a 'problem to prove' we are required to prove or disprove a clearly stated theorem A. We do not yet know whether A is true or false, but we derive from A another theorem B, from B another C and so on until we come upon a last theorem L about which we have definite knowledge.

Polya himself says [36] that

> heuristic, or *ars inveniendi* was the name of a certain branch of study, not clearly circumscribed, belonging to logic, or to philosophy, or to psychology ... The aim of heuristic is to study the methods and rules of discovery and invention.

The Logic Theorist embodied heuristics, as Minsky put it [32] :

> The LT (Logic Theory) program is centered around the idea of 'working backward' to find a proof. ... The heuristic technique of working backward yields something of a teleological process, and LT is a forerunner of more complex systems which construct hierarchies of goals and subgoals.

Indeed the concept of goal or problem used in LT fits extremely well in this context. Minsky [32] says: "abstractly a person is given a *problem* if he is given a set of possible solutions, and a test for verifying whether a given element of this set is in fact a solution to this problem."

Finally here is how the LCF designers put the matter [23]:

> To make sense of the notion of tactic, we further postulate a binary relation of achievement between events and goals. Many problem solving situations can be understood as instances of these three notions: goal, event and achievement. Further we make general type definitions:
>
> ```
> tactic = goal -> goal list # validation
>    validation = event list -> event .
> ```

The idea is that a tactic decomposes a goal G into a list of subgoals $G_1, \ldots, G_n$. An event g_i achieves a subgoal, say g_i achieves G_i. A validation v will take g_i and build an event $v(g_1, \ldots, g_n)$ which achieves G. In our setting we think of G as a theorem and g as its proof. The type–theoretic structure of ML makes it possible to define these concepts precisely. However, the type structure is not quite rich enough to do this exactly as we would wish. The full PRL type structure in fact captures exactly the concepts needed for constructive proof, but we shall not pursue this aspect of the theory in this paper.

5 Tactics in λ-PRL.

In this section, we examine how the general ML tactic mechanism has been specialized to PRL. Recall that the generic type of tactics is

$$\text{tactic} : \text{goal} \to \text{goal list} \# \text{validation},$$
$$\text{validation} : \text{event list} \to \text{event}.$$

What should *goal* and *event* correspond to in PRL? It would seem reasonable to associate *goal* with the PRL sequent (recall that a sequent consists of a variable environment, a hypothesis list, and a conclusion) and to associate *event* with the proof of a sequent. However, we want tactics to operate on partial proofs. This is a generalization since a sequent may be viewed as a degenerate partial proof. Because the tactics will be invoked in an interactive environment, it is desirable to allow them to return (*achieve*) incomplete proofs. A tactic will complete as much of a proof as possible, leaving what is left to be supplied by the user. Thus, for the purpose at hand, it is desirable to associate *event* with *partial proof.* Note that in what follows, the term *proof* should not be interpreted as implying that the proof is complete.

A further generalization of tactics is desirable for PRL. Tactics are classified into two categories: refinement tactics and transformation tactics. These are each describe below.

5.1 Refinement Tactics.

Refinement tactics are like derived rules of inference. The user invokes a refinement tactic by typing the name of the tactic where a refinement rule is requested by the proof editor. If the tactic succeeds, then the name of the tactic, as it was typed by the user will appear as the refinement rule in the proof. Any subgoals that are not completely proved by the tactic will be presented as subgoals of the refinement. As will be described in detail below, the tactic will have built a refinement proof that connects the original sequent on which the tactic was invoked, and the subgoals resulting from the tactic. This portion of the proof is hidden from the user, although it is saved for other uses (such as extraction of theorems). All that is visible to the user is the name of the tactic and the unproved subgoals.

When a refinement tactic is invoked, the following steps occur:

1. The variable **prlgoal** is associated with the current sequent viewed as a degenerate proof. Note that there may be a refinement rule and subgoals below the sequent, but that these are ignored as far as refinement tactics are concerned.

2. The given tactic is applied to **prlgoal**, resulting in a (possibly empty) list of unproved subgoals and a validation.

3. The validation is applied to the subgoals.

4. The tactic name is installed as the name of the refinement rule in the proof. The refinement tree that was produced by the validation in the previous step is stored in the proof. Any remaining unproved subgoals become subgoals of the refinement step.

The above four steps assume that the tactic terminates without producing an error or throwing a failure that propagates to the top level. If such an event does occur, then the error or failure message is reported to the user and the refinement is marked as *bad*, precisely as if a primitive refinement rule had failed.

```
EDIT THM max
------------------------------------
# top 1
[int x,y]
>> some m:int.
     ¬m<x & ¬m<y & (m=x∨m=y)

BY cases x<y ∨ ¬x<y

1# [int x,y]
     1.  x<y
     >> some m:int.
          ¬m<x & ¬m<y & (m=x∨m=y)

2# [int x,y]
     1.  ¬x<y
     >> some m:int.
          ¬m<x & ¬m<y & (m=x∨m=y)
```

Figure 5: The result of refinement using the **cases** refinement tactic.

5.2 Transformation Tactics.

Transformation tactics are used to transform one proof into another. The user invokes a transformation tactic by traversing the proof tree to a node, and supplying the name of the transformation tactic to be applied. The transformation tactic is applied to the whole proof below the designated node, including this node. If the transformation succeeds, then the result of the tactic replaces the previous subproof. In contrast to refinement tactics, the name of the transformation tactic is not included in the proof, and the result of the tactic explicitly becomes the subproof. Transformation tactics may be used, for example, to complete and expand unfinished proofs, to produce new proofs that are in some way analogous to a given proof, or to perform various analyses and optimizations to proofs. When a transformation tactic is invoked, the following occur:

1. The ML variable **prlgoal** is associated with the proof below, and including the current sequent.
2. The specified transformation tactic is applied to **prlgoal**, resulting in a (possibly empty) list of subgoals and a validation.
3. The validation is applied to the list of subgoals.
4. The proof that is the result of the previous step is grafted into the original proof below the sequent.

```
EDIT THM max
-------------------------------------
# top 1
[int x,y]
>> some m:int.
    ¬m<x & ¬m<y & (m=x∨m=y)

BY seq x<y ∨ ¬x<y

1* [int x,y]
    >> x<0 ∨ ¬ x<0

2# [int x,y]
    1.  x<0 ∨ ¬ x<0
    >> some m:int.
        ¬m<x & ¬m<y & (m=x∨m=y)
```

Figure 6: The result of applying "cases {x<0 ∨ ¬x<0}" as a transformation.

The key difference between refinement and transformation tactics is that transformation tactics are allowed to examine the subproof that is below the current node, whereas refinement tactics are not. The result of a transformation tactic will, in general, depend upon the result of the examination. Since most tactics do not depend on the subproof below the designated node, they may be used either as a transformation tactic or a refinement tactic. The main implementation difference between refinement tactics and transformation tactics is how the result of the tactic is used. In the former, the actual proof constructed by the tactic is hidden from the user, and only the remaining unproved subgoals are displayed. In the latter, the result is explicitly placed in the proof. In fact, since a refinement tactic can not examine the subproof, any refinement tactic may be used as a transformation tactic. Appendix A contains a summary of the library tactics in λ-PRL. The reader may wish to read this before proceeding.

5.3 The Tactic Library.

When λ-PRL is started, a library of predefined tactics is available to the user. Appendix A contains a summmary of the library tactics. Since this library does not, and could not, contain all the tactics a user might like, two facilities have been included in PRL that allow the user to define and experiment with his own tactics. First, ML may be used interactively within PRL. This allows the user to experiment with and to debug tactics, and to make (temporary) changes to the ML state. Second, PRL library objects may be created that can contain ML expressions._This allows the user to store tactic defintions between λ-PRL sesions.

There is a distinguished refinement tactic defined in the tactic library called **auto_tactic**. The **auto_tactic** is invoked automatically on the result of each primitive refinement step. This tactic is intended to complete any *simple* remaining subgoals without further effort on the part of the user. There is a default **auto_tactic**, but any refinement tactic may be designated by the user as the **auto_tactic**.

5.4 Implementing Tactics for PRL.

Tactics are implemented using the dialect of the ML programming language developed as part of the Cambridge LCF project (Tactics were first implemented in PRL using the original ML implementation from the University of Edinburgh and later reimplemented using Cambridge ML.) The ML language was intended to be the *meta-language* of "PPLAMBDA", the logic of LCF. One of the first tasks in implementing tactics for PRL was to change ML so that it is the meta-language of the λ-PRL logic.

In changing the object language of ML, all references to PPLAMBDA and operators on objects of the types of PPLAMBDA were removed. In their place were substituted primitive types of PRL objects and operations on these types as described below. The base types of PRL that are implemented in ML are described in figure 7. These types should not be confused with the base types of the λ-PRL logic (integers and lists of integers).

proof: Type of partial PRL proofs. Proofs consist of proofs nodes. Each node represents one refinement step. A node consists of a sequent, a refinement rule, and proofs of the children of the refinement, where the lattter two will be missing in some leaf nodes of an incomplete proof.

rule: Type of PRL refinement rules.

binding: Type of PRL bindings. A binding associates a variable (in the environment of a sequent) with a base type of integer or list of integers.

formula: Logical formulas of the λ-PRL.

term: Expressions over the PRL base types.

Figure 7: Summary of the primitive object language types for ML.

For the types of **term**, **formula**, **rule**, and **binding**, an associated collection of predicates, constructors, and destructors have been provided. The predicates on the type **formula**, for example, allow the kind of a formula to be determined. An example of a predicate on formulas is **is_universal**, which returns *true* if and only if the formula it is applied to is universally quantified. The constructors and destructors for each of the types allow new objects of the types to be synthesized and existing objects of the type to be divided into component parts. Appendix B contains a complete list of all primitive extensions to ML that have been made in implementing PRL tactics.

The rule constructors in ML do not correspond precisely to the rules in PRL. Refinement rules in PRL are usually entered as "**intro**", "**elim**", "**hyp**", etc. Strictly speaking, the notation "**intro**" refers not to a single refinement rule, but to a collection of introduction refinement rules. Normally the context of the proof is used to disambiguate the intended introduction rule at the time the rule is applied to a sequent. There is a similar ambiguity with the other *names* of the refinement rules. In addition to this ambiguity, the various sorts of the rules require different additional arguments. For example, to apply an **intro** rule to a conjunctive formula, no further information is required, but to apply **intro** to a disjunctive formula requires that one of the disjuncts be designated. Because rules in ML may exist independently of the proof context that allows the particular kind of rule to be determined, and because functions in ML are required to have a fixed number of arguments, the rule constructors have been sub-divided beyond the ambiguous classes of intro, elim, etc., that are normally visible to the PRL user. For instance, the function **intro** in ML, which takes no arguments, constructs an intro rule which will be valid when applied to any sequent that does not have a conclusion that is a disjunction or is quantified. For the latter, there are three rule constructors, **or_intro**, **all_intro**, and some_intro, each of which require additional arguments. There is a similar complication with the structure of the elimination rules. See Appendix B for the complete list of ML refinement rule constructors.

For the ML type of **proof**, a complement of destructors are available that allow the conclusion, hypotheses, environment, rule and children to be extracted from a proof. There is only one primitive function in ML that constructs new proof objects, that is **refine**. The function **refine** maps rules into tactics, and forms the basis of all tactics. When supplied with an argument rule and proof, refine performs, in effect, one refinement step upon the sequent of the proof using the given rule. The result of this is the typical tactic result structure of a list of subgoals paired with a validation. The list of subgoals is the list of children (logically sequents, but represented as degenerate proofs) resulting from the refinement of the sequent with the rule.

The function **refine** is the representation of the actual λ-PRL logic in ML. Every primitive refinement step accomplished by a tactic will be performed by applying **refine**. The subgoals are calculated by actually calling the PRL refinement routine, **deduce_children**, with the proof and the rule. Constructing the validation, an ML function, is more complicated. The purpose of the validation, given achievements of the subgoals, is to produce an achievement of the goal. The validation, hence, constructs a new proof node where the sequent is the sequent of the original goal, the refinement rule is the rule supplied as an argument to refine, and the children are the events achieving (partial proofs) of the subgoals. See figure 8 for a possible implementation of **refine**.

```
let refine rule =  λ proof .
    let children = deduce_children rule (sequent proof) in
        let validation =
            λ achievement .
                make_proof (sequent(proof), rule, achievement) in
            (children, validation);;
```

where **sequent** extracts the sequent from a proof, and **make_proof** constructs a new proof node given a sequent, rule and children.

Figure 8: An abstract implementation of the function **refine**.

If **deduce_children** is applied to a sequent that can not legally be refined with the rule, then **deduce_children** will fail (in sense of ML failures), including the text of the reason for the failure as part of the failure.

5.5 Validations.

The validations produced by **refine** may be thought of as alternative representations of proof nodes where the children have not yet been entered into the node. Compound validations produced by tacticals from these simple validations are, in this view, alternative representations of proofs. Validations are a mechanism whereby the actual construction of proof nodes is delayed until execution of the tactic is complete. The tactic works top–down, but the resulting proof is constructed by the validation bottom–up. This is particularly desirable in light of the possibility that part or all of a result may be abandoned when a tactic fails. Since no node that is not actually included in the final result is constructed until the tactic terminates, there is no need to undo the work of a failed tactic.

It is of fundamental importance that the resultant proof of a tactic is a correct proof in the λ-PRL logic. In particular, it should be the case that any theorem proved by a tactic could be proved without the tactic using only the primitive inference rules of the logic. The strong type structure of the ML language contributes to the enforcement of this property, as does the fact that `refine` is the only constructor for the type `proof`. Furthermore, the calculation of subgoals in the function `refine` by reference to the PRL refinement function, `deduce_children` guarantees that the subgoals of a refinement are correct, assuming the implementation of the logic is.

The only place where new proof nodes are constructed in tactics is in the validations produced by `refine`. Guaranteeing that the nodes produced by validations represent correct usage of the inference rules will therefore guarantee that the proof resulting from any tactic is logically correct. To know that the inference step represented by the validations are correct, it is necessary to verify that the list of achievements supplied to the validation correspond to the children of the goal under the refinement rule. Figure 9 gives an improved version of `refine` that realizes this; the actual implementation of `refine` differs in that there is additional *bookkeeping* information at each proof node that must be kept up to date. With this modification of `refine`, one may prove the following.

Theorem 1 *The result of a tactic (either transformation or refinement) is a valid λ-PRL proof.* □

Thus a tactic may fail to return a complete result, but can never return a (logically) incorrect result.

5.6 Equality of the Object Language Types.

In PRL, there is a flexible syntax for denoting terms and formulas; every term or formula can be represented in any number of ways by using the *def* mechanism. This fact complicates the implementation of the equality predicate, =, for terms and formulas. In λ-PRL, terms are considered equal if they denote the same object, and formulas are considered equal if the formulas they denote are α–convertible to the same formula. To understand the difficulty that this presents, it is necessary to understand how equality is implemented in ML.

```
let refine rule = λ proof .
    let children = deduce_children rule (sequent proof) in
        let validation =
            λ achievement .
                if (sequents children) = (sequents achievement) then
                    make_proof (sequent proof, rule, achievement)
                else
                    failwith 'Wrong achievements for subgoals.' in
                (children, validation);;
```

where **sequents** applied to a list of proofs produces a list of the sequents of the proofs.

Figure 9: An improved abstract implementation of **refine**.

One of the advantages of the system of polymorphic typing employed in ML is that all type checking is completed before execution (even when interpreted, ML code is *compiled* into LISP code before execution). As a result, all type information may be discarded before execution proceeds. For all types that are normally primitive in ML, the equality predicate is the same; two objects are equal if the are intensionally (i.e., structurally) equal.

In order to determine when the equality predicate is being applied to objects that are of type **term** or **formula**, it was necessary to introduce a limited amount of type information for use at execution time. This was accomplished by *tagging* the representation of terms and formulas. The equality predicate was altered to check objects for the tag, and apply the appropriate sort of equality test. The tags were chosen in such a way that no object could accidentally happen to contain the tags. The inclusion of type information at execution time is violently opposed to the philosophy of the ML polymorphic type system, but unfortunately was necessary.

5.7 Entering λ-PRL Terms and Formulas in ML.

To allow the use of constant terms and formulas in ML expressions, a special form of quotation has been introduced. Entering terms or formulas enclosed by braces will cause them to be parsed and assigned a type (**term** or **formula**) by the PRL parser. To facilitate the syntactic extensions of PRL, lexical analysis of ML expressions occurs in two phases. In the first phase, the PRL lexical analyzer expands out syntax macros. In the second phase, the ML lexemes are determined by the ML lexical analyzer. When parsing a term or formula, i.e., when scanning input between braces, only the PRL lexical analyzer determines the lexemes. This arrangement allows the syntactic extension facilities of PRL to be used for ML expressions stored in library objects, and allows the these extensions to be used when entering the names of tactics while editing a proof.

6 Writing Tactics.

6.1 Writing Simple Tactics.

We now examine how some sample tactics are written. The basis of all tactics are calls to the function **refine**. The following tactic, when applied to a goal that is existentially quantified, provide the witness 0 for the existentially quantified variable.

```
let zero_witness = refine (some_intro [{0}]);;
```

There are a couple features of this definition that may require comment. First, the witness for the existential variable must be a PRL term, thus the occurence of "{0}". The constructor for the existential introduction rule, **some_intro**, requires a list witness that corresponds to the list of variables quantified by a single existential quantifier. Thus it is necessary to enclose the 0 term in square brackets; square brackets are the list delimiters in ML. Because **refine** maps rules to tactics, **zero_witness** will correctly have type **tactic**. To make explicit the fact that **zero_witness** is a mapping, we might have given the following equivalent definition.

```
let zero_witness proof_tree = refine (some_intro [{0}]) proof_tree;;
```

If this tactic is applied to a goal that is not existentially quantified, then the application of refine will fail, and since the failure is not caught in **zero_witness**, the tactic itself will fail. A parameterized generalization of the **zero_witness** tactic could be expressed as

```
let witness witness_term = refine (some_intro witness_term);;
```

This tactic would be invoked during proof editing by typing "witness" followed by a term, for example, "**witness** $\{x+3\}$". Because it does not depend upon the existing children of the goal, **witness** could be employed as either a refinement or transformation tactic.

6.2 Combining Tactics Using Tacticals.

Tacticals are ML functions that map tactics to tactics. By using tacticals, existing tactics may be combined or changed to form new tactics. If we wanted, for example, the **witness** tactic to provide a witness for an existentially quuantified variable, and then try to complete the proof by simple reasoning, we could combine it with the **immediate** tactic using the **THEN** tactical. The **immediate** tactic is a tactic for proving simple sequents and is provided as part of the library of tactics. The **THEN** tactical applies the left–hand tactic and then applies the right-hand tactic to each subgoal of the first application.

```
let witness witness_term =
    (refine (some_intro witness_term)) THEN immediate;;
```

The next sample tactic is a tactic that we call **skolem**. This tactic takes a term as an argument and refines the goal until no more universally quantified variables are proceeding the conclusion. It then assumes that the formula is an existential one, and refines using some–intro with the given term. Thus the argument term should be thought of as a function of the universally quantified variables proceeding the first existential quantifier in the formula and any other variables free in the environment of the formula. The argument term is a generalized Skolem function. With the above definition of **witness**, the tactic **skolem** might be defined as

```
let skolem witness_term =
    universal THEN (witness witness_term) THEN trivial;;
```

The **THENL** tactical is a variant of **THEN** which accepts a list of tactics as the second argument rather than a single tactic. It applies to each child of the first tactic (i.e., the left argument) the corresponding tactic in the list of tactics. In addition to **THEN** and **THENL**, two other tacticals are of general usefulness: **REPEAT** and **ORELSE**. The tactical **REPEAT** will repeatedly apply a tactic until the tactic fails. That is, the tactic is applied to the goal of the argument proof, and then to the children produced by the tactic, and so on. The **REPEAT** tactical will catch all failures of the argument tactic, and can not generate a failure. For example,

```
let repeat_intro = REPEAT (refine intro);;
```

will perform (simple) introduction on the proof until it no longer applies (i.e., until the goal of one of the introduced refinements is atomic, or is a disjunctive or quantified formula). If **repeat_intro** is applied to a goal that can not be refined using simple introduction, then the tactic is equivalent to **IDTAC**, the identity tactic. The **ORELSE** tactical takes two tactics as arguments. It produces a tactic that applies the first tactic to a proof, and if that tactic fails, applies the second tactic. Thus,

```
let goal_simplify = REPEAT ((refine intro) ORELSE (refine arith));;
```

will repeatedly try to refine using the introduction rule, and if that fails, then it will apply the decision procedure *arith*.

The achievement relation between goals and events used in PRL tactics is quite weak; a proof *achieves* a goal if the sequent of the proof is equal to the sequent of the goal. It is occasionally desirable to have a stonger form of achievement. For example, we might wish to require that a tactic completely prove a goal, or we may wish to require that a tactic makes some progress towards proving the goal. We may implement tactics with these properties using the tacticals COMPLETE and PROGRESS. Let tac be any tactic. Then

```
let finish = COMPLETE tac;;
```

is a tactic that will either completely prove the goal or will fail. The COMPLETE tactical is implemented by checking that the result of the tactic applied to the goal has an empty subgoal list.

```
let COMPLETE tactic =
    λ goal . if null (first event)
        then event
        else fail
    where event = tactic goal;;
```

The PROGRESS tactical is implemented by verifying that the result of the argument tactic applied to a goal does not result in a subgoal list that contains exactly the original goal. The resulting tactic will fail unless the argument tactic performed at lesat one refinement step.

```
let PROGRESS tactic =
    λ goal . let result = tactic goal in
        if (first result) = [goal]
            then fail
            else result;;
```

A possible implementation of ORELSE is

```
let ORELSE tactic1 tactic2 goal =
    (tactic1 goal)  ? (tactic2 goal);;
```

However, a better implementation will take account of the fact that the first tactic may not fail, but may fail to make progress, in which case the second tactic should be applied.

```
let ORELSE tactic1 tactic2 goal =
   (PROGRESS tactic1 goal)  ? (tactic2 goal);;
```

The REPEAT tactical may be implemented in terms of the other tacticals. It is crucial that progress be required of the argument tactic here; otherwise we could produce a tactic that indefinitely does nothing.

```
letrec REPEAT tactic =
    ((PROGRESS tactic) THEN (REPEAT tactic)) ORELSE IDTAC;;
```

The THEN tactical is conceptually just as easy to implement, but requires a bit of list processing in combining the validations to make a new validation. IDTAC may be defined as

```
let IDTAC tactic = λ goal . ([goal], head);;
```

6.3 Validity and Strong Validity.

Two important properties of tactics first identified by Gordon, Milner, and Wadsworth [23] are validity and strong validity. Their original definitions were based upon the relation *achieves*. Recall that in the PRL context, a proof p' *achieves* a proof p if the seqents of p and p' are equal. This is quite a weak relation. Notice for example that any proof achieves itself. To get definitions of validity and strong validity that are analogous to the ones intended by Godon, Milner, and Wadsworth, we define a stronger from of achievement: *completely achieves*.

Definition: We say that a proof $\hat{p}$ *completely achieves* a proof p if $\hat{p}$ achieves p and $\hat{p}$ is a complete proof.

We can now define validity and strong validity. These definitions differ from those in [23] in the replacement of the achievement relation with complete achievement and in the specialization of vocabulary to the PRL context.

Definition: A tactic T is said to be *valid* if for every proof p, if

$$T(p) = [p_1, \ldots, p_n], v$$

then for any proofs $\hat{p_1}, \ldots, \hat{p_n}$ that completely achieve $p_1, \ldots, p_n$, the proof $v[\hat{p_1}, \ldots, \hat{p_n}]$ completely achieves p.

Definition: A tactic T is said to be *strongly valid* if T is valid and for every proof p with a provable sequent, if

$$T(p) = [p_1, \ldots, p_n], v$$

then $p_1, \ldots, p_n$ are completely achievable.

In a general context, a tactic need be neither valid nor strongly valid. One would like all tactics to be valid. However, strong validity is too restrictive since many tactics which employ heuristics or are intended for use only on particular kinds of goals are not strongly valid, but are still quite useful. For example, the universal tactic described above is not strongly valid since if it is applied to a universally quantified formula that can only be proved by induction, then the resulting subgoals will be unprovable (i.e., not completely achievable).

In a general context it is difficult to enforce a requirement that all tactics be valid [23]. In λ-PRL it is difficult to write invalid tactics, and as we describe below we could enforce the validity of all tactics. The refinement logic of λ-PRL ensures that if the subgoals of a sequent refined by a primitive refinement rule are completely provable, then the sequent is completely provable. By the construction of validations in `refine`, we have the following:

Theorem 2 *For every rule r,* (`refine` r) *is a valid tactic.* □

Furthermore, as noted in [23], the tacticals preserve validity (the results of `PROGRESS` and `COMPLETE` must be valid since these tacticals do not change the validation or subgoals of the argument tactic).

Theorem 3 *If T_1 and T_2 are valid tactics, and L is a list of valid tactics, then* T_1 `THEN` T_2, T_1 `ORELSE` T_2, `REPEAT` T_1, T_1 `THENL` L, `PROGRESS` T_1, *and* `COMPLETE` T_1 *are valid tactics.* □

Thus all tactics in λ-PRL ae valid so long as the validations are constructed using `refine` and the above tacticals, and the validations are not seperated from the associated subgoals. One can easily imagine using the ML type structure to represent the resultant type of a tactic (`proof list # validation`) so that validations can not be manipulated except by the primitive ML functions: `refine`, the tacticals given above, and a functions for applying a vaidation to the associated subgoals. This would ensure that all tactics in PRL were valid.

6.4 A Larger Example Tactic.

Let us now examine how the tactic `immediate` is implemented. The tactic is built from thirteen simple tactics, each of which will correctly operate on a very limited set of goals. The combination, however, works for a wide class of goals. We will show how to define a representative subset of the component tactics from which immediate is defined, and then show the definition of immediate. Each of the component tactics will refine with a particular refinement rule; thus there is a tactic for and-introduction, one for or-introduction, one for hypothesis, and so forth. The tactic for and-introduction is

```
let and_intro_tac goal =
    if is_conjunction (conclusion goal)
        then refine intro goal
    else fail;;
```

This tactic first checks that the goal is a conjunction because the introduction rule will succeed on many other kinds of goals, and we only want to refine conjunctive formulas in this tactic. The function `conclusion` extracts the conclusion from a proof, and the predicate `is_conjunction` returns true when applied to a formula if the formula is a conjunction. It should be apparent from the context what these sorts of functions mean in the following; see Appendix B for their meaning if necessary.

We want a tactic that will scan the list of hypotheses and if it finds a hypothesis, h, that equals the conclusion, refine the goal with the hypothesis rule using the hypothesis h as a witness. The tactic for hypothesis must find a formula in the hypothesis set, and then perform an hypothesis refinement. This is complicated slightly by the fact that the hypothesis rule requires the number of the hypothesis that matches the goal. We start the search with hypotheses number 1.

```
let hyp_tac goal =
    try_hyp (hypotheses goal) 1 goal;;

let try_hyp hyp_list hyp_num goal =
    if null hyp_list
        then fail
    else if (head hyp_list) = (conclusion goal)
        then refine (hyp hyp_num) goal
    else
        try_hyp (tail hyp_list) hyp_num+1 goal;;
```

In fact, **hyp_tac** is not implemented quite like this. This control structure (scanning through each hypothesis looking for a particular property of the hypothesis, and then performing a refinement if the property holds) occurs frequently enough that a special functional, **map_hyp**, has been written that will take a tactic–like function, and apply this function on each hypothesis until one of the applications succeeds. The actual implementation of **hyp_tac** employs this functional.

The tactic **or_intro_tac** will perform or–introduction if the goal is a disjunction. This is not as simple as and–introduction since or–introduction refinements require that a disjunct of the disjunctive formula be designated, and that the disjunct be proved as a subgoal. In order to determine which is the proper disjunct to designate, this tactic tries the first disjunct first. If it is unable to completely prove the subgoal of this refinement, then it tries the second disjunct.[2] If it is unable to completely prove the subgoal of this refinement, the the tactic fails. To prove the subgoals, **or_intro_tac** will recursively call **immediate**.

[2] This is a simplification since disjunctions in λ-PRL may have arbitrarilly many disjuncts.

```
let or_intro_tac goal =
    if is_disjunction (conclusion goal)  then
        ((refine (or_intro 1) goal) THEN (COMPLETE immediate))
        ORELSE
        ((refine (or_intro 2) goal) THEN (COMPLETE immediate))
    else
        fail;;
```

The tactic that performs implication-elimination is similar. It scans the hypothesis list until it finds a implication. It then refines using implication-elimination, designating that hypothesis. It then applies **immediate** to prove the first subgoal, the antecedent of the implication, and fails unless it can completely prove it. If it succeeds, then it tries **immediate** on the second subgoal, but does not require that this part of the subproof be complete. The tactic for or-elimination is simlar, but requires that the proof be compelete below the or-elimination refinement; otherwise, proofs may be split on disjunctive hypotheses that are irrelevant to the conclusion currently under consideration, forcing a duplication of proof reasoning below this refinement.

The remaining introduction tactics used in **immediate** are analogous to **and_intro_tac**. There is also a **false_intro_tac** that is analogous to **hyp_tac**. Finally, there is a tactic for the division axiom that will try refinement using this axiom, and then requires that the proof be complete below the refinement. With all of these defined, **immediate** could be defined as

```
let immediate = REPEAT (        hyp_tac
                         ORELSE true_intro_tac
                         ORELSE false_intro_tac
                         ORELSE false_elim_tac
                         ORELSE (refine arith)
                         ORELSE (refine equality)
                         ORELSE and_elim_tac
                         ORELSE implication_intro_tac
                         ORELSE not_intro_tac
                         ORELSE division_axiom_tac
                         ORELSE and_intro_tac
                         ORELSE implication_elim_tac
                         ORELSE or_elim_tac
                       );;
```

The order of the subtactics in **immediate** is quite important since it determines the order in which refinement rules will be applied. The rules used by **immediate** may be roughly categorized by the number of subgoals that are produced by refining with the rule. The first tactics in the above list are the tactics that corespond to rules that do not produce subgoals. This includes hypothesis, true–introduction, false–introduction, false–elimination, and the decision procedures arith and equality. Obviously, whenever one of these rules can apply, it should be applied since it will completely prove the goal. Next come the tactics that correspond to rules that produce only one subgoal, and–elimination and implication–introduction. Finally come the tactics that correspond to rules that may produce more than one subgoal. The point of the latter part of the ordering is that the branching factor of the proof should be kept as low as possible for as long as possible. Of course, all of this is heuristic, and it is possible to find goals for which other orderings would perform better.

6.5 The Quantifier Tactic.

The `immediate` tactic is applicable only to unquantified formulas. The next example tactic is designed to prove certain quantified formulas. It is common in λ-PRL to have theorems of the form $\forall v_1 \ldots \forall v_n . \exists w . P(v_1, \ldots, v_n, w)$. If this theorem were to be proved without using induction, then it would be necessary to apply the introduction rule n times, once for each universally quantified variable. The following tactic will repeatedly refine a proof until the conclusion is not universally quantified.

```
letrec universal proof =
  let goal = conclusion (proof) in
    if is_universal goal then
      (refine (all_intro (quantified_vars goal)) THEN universal) proof
    else
      IDTAC proof;;
```

This simple tactic is characteristic of many useful tactics. These tactics provide a level of abstraction above the level of the primitive rule of inference, allowing proof to be expressed and presented concisely.

As another example tactic, we examine the `quantifier` tactic. This tactic searches the hypotheses of a sequent for a hypothesis with a quantifier structure that is *suitably related* to the quantifier structure of the conclusion. It then constructs a refinement that proves how they are related. Since the implementation details of this tactic are routine, we present a high–level description of the tactic.

When is the matrix of a hypothesis *suitably related* the the matrix of the conclusion? Consider the situation where the hypothesis and the conclusion each have just two quantified variables. There are four cases to consider. Schematically, they are:

1. $\forall x . \forall y . P(x, y) \vdash \forall y . \forall x . Q(x, y)$

2. $\exists x . \exists y . P(x, y) \vdash \exists y . \exists x . Q(x, y)$

3. $\exists x.\forall y.P(x,y) \vdash \forall y.\exists x.Q(x,y)$

4. $\forall x.\exists y.P(x,y) \vdash \exists y.\forall x.Q(x,y)$

The first three cases are valid in the λ-PRL logic, but the fourth is not in general. This observation may be used to prove the following theorem:

Theorem 4 *Let H, C be formulas with matrices M_H, M_C and bodies B_H, B_C. The matrix M_C is said to be a legal permutation of M_H if and only if*

1. *M_C is a permutation of M_H, and*

2. *$\forall$ universally quantified variables a in M_C . $\forall$ existentially quantified variables e in M_C . a proceeds e in $M_H \Rightarrow$ a proceeds e in M_C.*

If M_C is a legal permutation of M_H and $\mathbf{H}_1, B_H, \mathbf{H}_2 \vdash B_C$ is provable, then $\mathbf{H}_1, H, \mathbf{H}_2 \vdash C$ is provable. □

The quantifier tactic is roughly based upon this theorem. The result of an application of this tactic is one unproved subgoal: to prove that the body of the conclusion follows from the hypotheses. Once it has been verified that the matrix of a conclusion is a legal permutation of the matrix of one of the hypotheses, the refinement is constructed by applying the following four rules repeatedly (in order) until no rule applies.

1. If the conclusion is universally quantified, refine using universal–introduction.

2. If the hypothesis is existentially quantified, refine using some–elimination.

3. If the hypothesis is universally quantified, refine using all–elimination with the same variable name.

4. If the conclusion is existentially quantified, refine using some-introduction with the same variable name.

The usefulness of the `quantifier` tactics depends upon the heuristic that if the matrix of the conclusion is the a legal permutation of a matrix of a hypothesis, then the theorem may be proved by beginning the proof by the simple quantifier manipulations given above. In general, this is a good heuristic. There are, however, cases where the `quantifier` tactic will proceed to prove tangential facts that will prevent the proof of the complete theorem. For example, `quantifier` may construct a tangential refinement if the conclusion must be proved by induction, or if the the matrix of the conclusion is a valid permutation of more than one hypothesis matrix. That is, the tactic `quantifier` is not strongly valid. In combining `quantifier` with other tactics, this problem may be avoided by employing the `COMPLETE` tactical. The following tactic combines immediate, universal, and quantifier to form a powerful, but still simple tactic.

```
letrec trivial =
  REPEAT
    ( immediate ORELSE
      ((PROGRESS quantifier) THEN (COMPLETE trivial)) ORELSE
      universal
    );;
```

6.6 Example Transformation Tactics.

As a final example tactic we examine a pair of transformation tactics, mark and copy, that can be used to copy proofs. To use these tactics, the user locates the proof he wants to copy and invokes the mark tactic as a transformation tactic. He then locates the goal where he wants the proof inserted, and invokes copy as a transformation tactic. The application of mark does not change the proof and records the proof (in the ML state) so that it is available when the copy tactic is used. Note that the goal of the proof where the copied proof is inserted is not, in fact can not, be changed by the copy tactic.

The mark tactic is defined as follows. The variable saved_proof is a reference variable of type proof.

```
let mark goal_proof =
    (saved_proof := goal_proof;
     IDTAC goal_proof
    );;
```

The basis of the copy tactic is a verbatim copy of the saved proof. This is accomplished by recursively traversing the saved proof and refining using the refinement rule of the saved proof. The following is a first approximation of the copy tactic.

```
letrec copy_pattern pattern goal =
    (refine (refinement pattern)
     THENL (map copy_pattern (children pattern))
    ) goal;;


let copy goal =  copy_pattern saved_proof goal;;
```

This version will of copy will fail if the saved proof is incomplete (since the selector refinement fails if applied to a proof without a refinement rule) To correct this deficiency, we define a predicate is_refined and change copy_pattern to apply immediate where there is no refinement in the saved proof.

```
letrec copy_pattern pattern goal =
    if is_refined pattern then
        (refine (refinement pattern)
         THENL (map copy_pattern (children pattern))
        ) goal
    else
        immediate goal;;
```

With this as a basis, any number of more general proof copying tactics can be defined. For example, the following version of copy looks up the actual formula being referenced in elimination rules (rather than just the index in the hypothesis list) and locates the corresponding hypothesis in the context where the proof is being inserted. Further, if one of the refinements from the pattern fails in the new context, then immediate is tried (rather than the whole copy failing).

```
letrec copy_pattern pattern goal =
    if is_refined pattern then
        (refine (adjust_elim_rules pattern goal)
            THENL (map copy_pattern (children pattern))
         ORELSE immediate
        ) goal
    else
        immediate goal;;
```

The function adjust_elim_rules checks if the refinement is an elimination rule. If not, it returns the value of the rule unchanged. If it is an elimination rule, it looks up the hypothesis in the pattern indexed by the rule and finds the index of an equal hypothesis in the hypothesis set of the goal. In this case the value returned is the old elimination rule with the index changed to be the index in the hypothesis set of the goal rather than the pattern.

The tactics mark and copy illustrate the usefulenss of transformation tactics and how transformation tactics can be used to extend the λ-PRL proof editor. The result of the copy is almost a verbatim copy of the original proof. However, one could imagine writing more general tactics to construct proofs *in analogy* to existing proofs.

7 Experience and Conclusions.

Experience.

We have come to rely more and more heavily on the tactic mechanism of λ-PRL because it has proven to be so successful for a broad class of very simple proof techniques. A number of users have written several small special purpose tactics, and we have written a variety of general tactics a discussed previously. These general tactics allowed us to achieve in a matter of days most of the capabilities of the immediate rule mechanism of PL/CV [12] which had to be *hand coded* in the basic implementation (on a scale of weeks). Moreover, the tactics that accomplished this are easily understood in every detail and can be employed in building more complicated tactics.

The tactic mechanism has also allowed us to structure the deductive power of the system around specialized *reasoners*. Users have been able to write a collection of tactics and theorems designed to construct proofs about a particular concept. For example, Tim Griffin [24] collected a dozen theorems about the monotonicity of the arithmetic operators, and wrote a few tactics to systematically apply these results to a goal involving the order relation. The collection of tactics and theorems constitutes a *reasoner*, which he calls ARITHPACK, that can be invoked by users and by other other tactics.

Plans.

We plan to explore more domain specific tactics, such as those which have knowledge of a particular concept such as the order relation in ARITHPACK, and we intend to explore more general tactics such as those used in the Boyer–Moore [6] theorem prover to structure inductive proofs. As in other cases of theory construction, it will be necessary to accumulate a number of simple methods before we can begin to build the powerful methods that human problem solvers draw upon to attack the most routine problems.

Another promising path of investigation that we have undertaken is the unification of the object language and the metalanguage. Is is possible to express the ML primitives for ν-PRL in the theory itself and to write functions in ν-PRL which are tactics. In this setting it is possible to prove in advance that certain tactics will succeed and thereby avoid running them (C.f., Davis & Schwartz [13], Boyer & Moore [6]). A language with this *closure property* is theoretically quite interesting as is shown in Constable [11].

Conclusion

We believe that our experience with tactics has demonstrated the effectiveness of the ML tactic mechanism and the implementation and extensions of it discussed here. Not only are the functional metalanguage mechanisms provided by ML well suited to problem domain and compatible with the object theory, but they provide a level of abstraction comparable to that of the object theory. Thus users can build proofs and tactics with nearly equal facility. We think that this is a feature of automated logics which should be perfected and extensively explored.

Appendix A

Summary of Refinement Tactics in λ-PRL.

immediate: This tactic will complete most theorems where the goal follows by simple propositional reasoning from the hypotheses possibly using the decision procedures of λ-PRL. If **immediate** does not completely prove the goal, then it will return a partial proof which is guaranteed to be free of refinements that would prevent the subgoals of the resulting proof from being proved (although it may have refinements that will be unnecessary in the final proof).

sequence: This tactic takes a formula as an argument and does a consequence ("seq") refinement with this formula. It then applies the tactic **immediate** to both subgoals. This is useful if the formula is easily deduced from the goal since it eliminates the need to prove it by hand. An example call to this tactic with the formula "$x + 27 < 0$" would be "**sequence** $\{x + 27 < 0\}$".

cases: This tactic takes a disjunctive formula as an argument, and performs the following. The tactic tries to prove the formula as a consequence, in much the same way as the sequence tactic would. On the second subgoal, the formula is refined using the elimination rule for disjunctions, and the tactic trivial is applied to the result. This tactic is useful wherever the proof proceeds by cases, and particularly useful if it is easy to show that the list of cases is exhaustive. An example application of this tactic would be "**cases** $\{x < 0 \vee \neg(x < 0)\}$".

universal: This tactic repeatedly refines the goal until all the universal quantifiers prefixing the conclusion have been eliminated. This is useful for removing a string of universal quantifiers in one step.

quantifier: If the list of quantifiers of the conclusion is a legal permutation of the quantifiers of one of the hypotheses, then this tactic performs the necessary refinement steps to prove this. Two examples where this arises are if the universal quantifiers have been rearranged (such as in $\forall x{:}int.\forall y{:}int.x < f(y) \vdash \forall y, x{:}int.x < f(y)$) and if an existential quantifier has been moved into a subformula as in
$\forall x{:}int.\exists y{:}int.\forall z{:}int.y = g(x, z) \vdash \forall x{:}int.\forall z{:}int.\exists y{:}int.y = g(x, z)$).

trivial: When applied this tactic tries to prove the goal by applying **quantifier** or **universal** if the goal is quantified, and applying **immediate** to any quantifier–free subgoals. This tactic includes some heuristic assumptions about how the proof will proceed following the application of the tactic. For example, it assumes that induction will not be used to prove universally quantified formulas.

skolem: This tactic takes a term as an argument and refines the goal until no more universally quantified variables are proceeding the conclusion. It then assumes that the formula is an existential one, and refines using some-intro with the given term. Thus the term should be thought of as a function of the universally quantified variables that proceed the first existential quantifier and any other variables that are free in the environment (that is, it is a generalized Skolem function, whence the name). An example of use on the goal of "$\vdash \forall x{:}int.\exists y{:}int.x + 1 = y$" would be "**skolem**$\{x+1\}$". This says that the witness for y is to be $x+1$.

Appendix B

Summary of Extensions to ML Language for λ-PRL.

Constructors and Destructors for Proofs.

- **refine**: **rule** $\rightarrow$ **tactic**. Given a rule, this function builds a tactic that will refine a proof based upon the rule. The function **refine** will fail if unable to refine the goal using the rule.
- **true_goal_proof** : **proof**. A constant proof with an empty environment, no assumptions, and *true* as the goal.
- **conclusion**: **proof** $\rightarrow$ **formula**. Returns the conclusion part of a goal.
- **hypotheses**: **proof** $\rightarrow$ **formula list**. Returns an ordered list of the hypotheses of a goal.
- **environment**: **proof** $\rightarrow$ **binding list**. Returns an ordered list of the bindings in the environment of a goal.
- **refinement**: **proof** $\rightarrow$ **rule**. Returns the current refinement rule. Fails if no rule.
- **children**: **proof** $\rightarrow$ **proof list**. Returns the subgoals produced when the current refinement rule was applied. Fails if no rule has been used.

Predicates, Constructors, and Destructors for Rules.

- **rule_kind**: **rule** $\rightarrow$ **tok**. Returns the type of the rule. Possible values are: ELIM, ALL-ELIM, SOME-ELIM, INTRO, OR-INTRO, ALL-INTRO, SOME-INTRO, HYP, SEQ, ARITH, LEMMA, INT-IND, LIST-IND, DEF, EQUALITY, SIMPLIFY, DIVISION, and TACTIC.

- **elim: int → rule.** Constructs a simple elimination refinement rule. This rule can be used to refine a sequent if the named hypothesis is not quantified.
- **destruct_elim: rule → int.** Returns the integer which represents the hypothesis of an elim rule. The function fails if the rule is not an elim rule.
- **all_elim: int → term list → rule.** Builds a rule for eliminating a universally quantified formula. The terms supplied will be used instantiate the quantified variables of the hypothesis.
- **destruct_all_elim: rule → int#(term list).** Destructs an all_elim rule.
- **some_elim: int → tok list → rule.** Builds a rule for eliminating an existentially quantified formula. The tokens in the token list will be the variables entered into the environment for the quantified variables.
- **destruct_some_elim: rule → int#(tok list).** Destructs a some_elim rule.
- **intro: rule.** Builds a rule for introducing on any formula except a disjunction or quantified formula.
- **or_intro: int → rule.** Builds a rule for introducing on a disjunction. The integer is the designator indicating which disjunct should be proved.
- **destruct_or_intro: rule → int.** Destructs an or_intro rule.
- **all_intro: tok list → rule.** Builds a rule for introducing on a universally quantified formula. The token list is the list of variables to be used in place of the bound variables.
- **destruct_all_intro: rule → (tok list).** Destructs an all_intro rule.
- **some_intro: term list → rule.** Builds a rule for introducing on an existentially quantified formula. The term list are the witnesses to be used.
- **destruct_some_intro: rule → (term list).** Destructs a some_intro rule.
- **hyp: int → rule.** Builds a hypothesis rule. The integer is the number of the hypothesis.
- **destruct_hyp: rule → int.** Destructs a hypothesis rule.
- **seq: formula list → rule.** Builds a consequence rule. The list of formulas are the intermediate goals to be proved.

- **destruct_seq**: **rule** → **(formula list)**. Destructs a consequence rule.

- **arith**: **rule**. The arith rule.

- **division**: **rule**. The division rule.

- **equality**: **rule**. The equality rule.

- **simplify**: **rule**. The simplify rule.

- **lemma**: **tok** → **rule**. The lemma rule. The token is the name of the theorem being employed as a lemma.

- **destruct_lemma**: **rule** → **tok**. Destructs a lemma rule.

- **ind**: **int** → **int** → **int** → **int** → **rule**. Builds an integer induction rule. The four arguments are respectively the downward step size, the lower limit of the base cases, the upper limit of the base cases, and the upward step size.

- **destruct_ind**: **rule** → **(int # int # int # int)**. Destructs an integer–induction rule. The results are the downward step size, the lower limit of the base cases, the upper limit of the base cases, and the upward step size.

- **list_ind**: **(tok list)** → **tok** → **rule**. Builds a list induction rule. The token list is the list of integer variables, and the token is the list variable.

- **destruct_list_ind**: **rule** → **((tok list) # tok)**. Destructs a list–induciton rule. The results are the list of integer variables and the list variable.

- **def**: **term** → **tok** → **rule**. Builds a definition rule. The term is the function or extraction object being referenced. The token is the kind of the reference, and should be one of the following: INT-BASE, INT-UP, LIST-BASE, LIST-IND, or EXT. The kind is determined by the kind of the object being referenced, i.e., a recursive function on intergers, a recursive function on lists, or an extraction function.

- **destruct_def**: **rule** → **term**. Destructs a definition rule.

- **tactic**: **tok** → **rule**. Builds a rule that will apply a tactic as a refinement tactic. The token is the ML expression that represents the refinement tactic.

- **destruct_tactic**: **rule** → **tok**. Destructs a tactic rule.

Predicates and Selectors on Formulas.

- `formula_kind`: `formula` → `tok`. Gets the kind of formula as a token. Possible values are `'OR'`, `'IMPLIES'`, `'AND'`, `'NOT'`, ALL, `'SOME'`, `'TRUE'`, `'FALSE'`, `'LESS'`, and `'EQUAL'`.
- `is_disjunction`: `formula` → `bool`. Returns true if the formula is a disjunction. The following predicates are similarly defined: `is_implication`, `is_conjunction`, `is_negation`, `is_universal`, `is_existential`, `is_true`, `is_false`, `is_less`, `is_equal`.
- `quantified_vars`: `formula` → `tok list`. Returns the quantified variables (as tokens) of a quantified formula. Fails if the formula is not a quantified formula.
- `quantified_var_types`: `formula` → `tok list`. Returns the types of the quantified variables (i.e., tokens `'INT'` or `'LIST'`) for each quantified variable of a quantified formula. It fails if not a quantified formula.
- `body`: `formula` → `formula`. Returns the body of an ALL, SOME, or NOT formula. Fails if the formula is not one of these kinds.
- `antecedent`: `formula` → `formula`. For implication formulas, returns the antecedent. Fails if the formula is not an implication.
- `consequent`: `formula` → `formula`. Returns the consequent of an implication. Fails if the formula is not an implication.
- `disjuncts`: `formula` → `formula list`. Returns the disjuncts of a disjunction. Fails if the formula is not a disjunction.
- `conjuncts`: `formula` → `formula list`. Returns the conjuncts of a conjunction. Fails if the formula is not a conjunction.
- `left_hand_side`: `formula` → `term`. Returns the term on the left-hand side of a LESS or EQUAL formula. Fails if the formula is not one of these two kinds.
- `right_hand_side`: `formula` → `term`. Similar to above except returns the term on the right-hand side of the operator.

Predicates, Constructors, and Selectors for Terms.

- **term_kind**: `term` → `tok`. Returns the kind of a term. The possible values are the tokens `'CONS'`, `'ADD'`, `'SUB'`, `'MUL'`, `'DIV'`, `'MOD'`, `'HD'`, `'TL'`, `'NEG'`, `'INTEGER'`, `'VAR'`, `'FUNCTION'`, and `'LIST'`.
- **sub_terms**: `term` → `term list`. Returns the subterms of a term as a (potentially empty) list of terms. Fails if the term is an integer or variable term.
- **identifier_of_term**: `term` → `tok`. Returns the identifier of a variable term or function term. Fails if not a variable or function term.
- **integer_term_value**: `term` → `int`. Returns the integer of an integer term. Fails if not an integer term.
- **make_var_term**: `tok` → `term` . Make a variable term with the variable name being the token.

Destructors for Bindings.

- **destruct_binding**: `binding` → `tok#tok`. Given a binding, reduces the binding to the variable name and type name, both tokens. The type name will be either `'INT'` or `'LIST'`.
- **variable**: `binding` → `tok`. Return the name part of a binding.
- **range**: `binding` → `tok`. Returns the range (type) part of a binding. The range will be `'INT'` or `'LIST'`.

Auto-tactic.

- **set_auto_tactic**: `tok -> void`. Sets the auto-tactic to the tactic represented by the text of the token.
- **show_auto_tactic**: `void -> tok`. Returns the current setting of the auto-tactic.

Bibliography

[1] Abrahams, P. **Machine Verification of Mathematical Proof.** Doctoral dissertation, MIT, 1963.

[2] Aho, Alfred V., J.E. Hopcroft and J.D. Ullman. **The Design and Analysis of Computer Algorithms.** Addison–Wesley, 1974.

[3] Bates, J.L. and R.L. Constable. **Proofs as Programs.** Department of Computer Science Technical Report, TR 82-530, Cornell University, 1982.

[4] Bates, J.L., **A Logic for Correct Program Development**, Doctoral dissertation, Cornell University, 1979.

[5] Bledsoe, W. Non–Resolution Theorem Proving. *Artificial Intelligence*, 9, 1977, 1-36.

[6] Boyer, R.S. and J.S. Moore. **A Computational Logic.** Academic Press, N.Y., 1979.

[7] Boyer, R.S. and J.S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In R.S. Boyer and J.S. Moore, editors. **The Correctness Problem in Computer Science**, Academic Press, NY, 1981, 103-184.

[8] Brouwer, L.E.J. **Collected Works.** Vol. 1, edited by A. Heyting, North-Holland, 1975.

[9] Cohn, A.J. **Machine Assisted Proofs of Recursion Implementation.** Doctoral dissertation, University of Edinburgh, 1980.

[10] Constable, R.L., and J.L. Bates. **The Nearly Ultimate PRL.** Department of Computer Science Technical Report, TR 83-551, Cornell University, January 1984.

[11] Constable, R.L. Universally Closed Classes of Total Computable Functions. Department of Computer Science Technical Report, TR 84-640, 1984.

[12] Constable, R.L., S.D. Johnson and C.D. Eichenlaub. **Introduction to the PL/CV2 Programming Logic.** Lecture Notes in Computer Science, Vol. 135, Springer-Verlag, 1982.

[13] Davis, M. and J.T. Schwartz. Metamathematical Extensibility for Theorem Verifiers and Proof Checkers. *Comp. and Math. With Applications*, 5, 1979, 217-230.

[14] deBruijn, N.G. A Survey of the Project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, **Essays on Combinatory Logic, Lambda Calculus and Formalism**, Academic Press, 1980, 589-606.

[15] deBruijn, J.G. The Mathematical Language AUTOMATH, Its Usage and, Some of Its Extensions. In *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics, Vol. 125, Springer–Verlag, 1970, 29-61.

[16] De Millo, R.A., R.J. Lipton, and A.J. Perlis, Social processes and proofs of theorems and programs. Communications of the ACM, 22(5), 1979, 271–280.

[17] Fischer, M.J. and M.O. Rabin, *The complexity of theorem proving procedures.* Project MAC Report, MIT, 1974.

[18] Frege, G. **Begriffsschrift, A Formula Language, Modeled Upon that for Arithmetic, for Pure Thought.** Reprinted in J. van Heijenoort, editor, **From Frege to Gödel: A Source Book in Matematical Logic, 1879–1931**, Harvard University Press, Cambridge, Mass., 1967, 1-82.

[19] Gentzen, G. Investigations Into Logical Deduction. Reprinted in M.E. Szabo, **The Collected Papers of Gerhard Gentzen**, North–Holland, Amsterdam, 1969, 68-131.

[20] Gödel, K. The Completeness of the Axioms of the functional Calculus of Logic. Reprinted in J. van Heijenoort, editor, **From Frege to Gödel: A Source Book in Matematical Logic, 1879–1931**, Harvard University Press, Cambridge, Mass., 1967, 583-591.

[21] Gödel, K. On Formally Undecidable Propositions of *Principia Mathematica* and related systems I. Reprinted in J. van Heijenoort, editor, **From Frege to Gödel: A Source Book in Matematical Logic, 1879–1931**, Harvard University Press, Cambridge, Mass., 1967, 596-616.

[22] Gödel, K. On the Length of Proofs. In M. Davis, editor, **The Undecidable.** Raven Press, Hewlett, N.Y., 1965, pp. 82-83.

[23] Gordon, M., R. Milner and C. Wadsworth. **Edinburgh LCF: A Mechanized Logic of Computation**. Lecture Notes in Computer Science, Vol. 78, Springer–Verlag, 1979.

[24] Griffin, T., Private communications, June 1984.

[25] Hartmanis, J. **Feasible Computations and Provable Complexity Properties**, SIAM Conference, Philadelphia, P.A., 1978.

[26] Hilbert D. and P. Bernays. **Grundlagen der Matematik I.** Springer–Verlag, 1968.

[27] Jutting, L.S., **Checking Landau's 'Grundlagen' in the AUTOMATH System**. Doctoral Thesis, Eindhoven University, *Mathematics Centre Tracts No. 83*, Mathematics Centre, Amsterdam, 1979.

[28] Keyser, C.J. Review of *Principia Mathematica*. In *Science*, Vol. 35, 1912, pp. 110+.

[29] Landau, E. **Grundlagen der Analysis**. Chelsea Publishing Co., N.Y., 1930.

[30] Leibniz, Gottfried W. **Logical Papers: A Selection**. Edited and translated by G.H.R. Parkinson, Clarendon Press, Oxford, 1966.

[31] McCarthy, J. Computer Programs for Checking Mathematical Proofs, *Proceedings of the Symposia in Pure Mathematics*, Vol. V. Recursive Function Theory, American Mathematics Society, Providence, R.I., 1962, 219-228.

[32] Minsky, M. Steps Toward Artificial Intelligence. In E. Feigenbaum and J. Feldman, editors, **Computers and Thought**. McGraw–Hill, 1963, 406-450.

[33] Newell, A., M. Shaw, and H. Simon. Empirical exploratins with the logic theory machine. In E. Feigenbaum and J. Feldman, editors, **Computers and Thought**. McGraw–Hill, 1963, 109-133.

[34] Paulson, L. Tactics and Tacticals in Cambridge LCF, University of Cambridge Computer Laboratory Technical Report Number 39, 1983.

[35] Poincaré, Henri, La logique de l'infini, *Scientia*, 12, 1912, 1-11.

[36] Polya, G. **How To Solve It**. Princeton University Press, 1945.

[37] The PRL staff. **PRL: Proof Refinement Logic Programmer's Manual (Lambda PRL VAX Version)**. Computer Science Department, Cornell University, 1984.

[38] Scherlis, W.L. and D.S. Scott. First Steps Toward Inferential Programming, *Proc. IFIP Congress*, Paris, 1983.

[39] Siekmann, J., and G. Wrightson. **Automation of Reasoning**, Vol. I and II. Springer–Verlag, 1983.

[40] Smith, B. **A Reference Manual for the Environmnetal Theorem Prover, An Incarnation of AURA**, Argonne National Laboratory, Technical Report in preporation, 1984.

[41] Suppes, P., *University–level computer–assisted instruction at Stanford: 1968–1981.* Institute for Mathematical Studies in the Social Sciences, Standford University, 1981.

[42] Teitelbaum, T. and T. Reps. *The Cornell Program Synthesizer: A syntax–Directed Programming Environment.* Communications of the ACM, 24(9), September 1981, 563-573.

[43] Wang, H. Proving programs by pattern recognition – 1. *Communications of the ACM*, 3(4), 1960, 229-243.

[44] Weyhrauch, R. Prolegomena to a Theory of Formal Reasoning. *Artificial Intelligence*, *13*, 1980, 133-170.

[45] Whitehead, A.N. and B. Russell. **Principia Mathematica**. Vol. 1, Cambridge University Press, Cambridge, 1925.