Proofs as Programs

JOSEPH L. BATES and ROBERT L. CONSTABLE
Cornell University

The significant intellectual cost of programming is for problem solving and explaining, not for coding. Yet programming systems offer mechanical assistance for the coding process exclusively. We illustrate the use of an implemented program development system, called PRL ("pearl"), that provides automated assistance with the difficult part. The problem and its explained solution are seen as formal objects in a constructive logic of the data domains. These formal explanations can be executed at various stages of completion. The most incomplete explanations resemble applicative programs, the most complete are formal proofs.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Automated logic, constructive logic, formal logic, intelligent systems, program specification, program correctness, very high level programming languages

1. THE NATURE OF PROGRAMMING

The Setting

What is the difference between programming and mathematical problem solving? To explore the question, consider this simple but real programming problem.[1] Given an integer sequence of length n, $[a_1, \ldots, a_n]$, write a program to find the sum $\sum_{i=p}^{p+q} a_i$ of a *consecutive* subsequence $[a_p, a_{p+1}, \ldots, a_{p+q}]$ that is maximum among all sums of consecutive subsequences $[a_i, a_{i+1}, \ldots, a_{i+k}]$. Call such consecutive subsequences *segments*. For example, given [−3, 2, −5, 3, −1, 2] the maximum segment sum is 4, achieved by segment [3, −1, 2]. When a problem description refers to ordinary mathematical concepts such as integers, sequences, and sums, we recognize it as a certain kind of mathematical problem, one requiring an algorithmic solution. But there is at least one major difference

[1] This problem came to us from Jon Bentley (who encountered it while consulting) via David Gries.

This research was supported in part by National Science Foundation grant MCS-81-04018.
Authors' address: Cornell University, Ithaca, NY 14853

between programming and algorithmic mathematics.[2] The solution to a programming problem is a concrete program, a piece of code that can be executed by some computer. So there is an element of *formality* in the result. As in good mathematics, the problem must be solved exactly and rigorously, but *in addition* the solution must conform to methods of expression completely determined in advance by the programming language.

What is the intellectually difficult part of programming? Certainly, a great deal of effort might be invested in learning a *formal coding language* in which to *code* the problem, such as FORTRAN or ALGOL. A good deal of effort might be invested in getting the particular piece of code to execute on a specific machine, for example, typing, editing, submitting, and so on. The task may even require mechanical assistance from diagnostic compilers, smart editors, and the like. Nevertheless, in all but the most routine problems, the significant effort in programming is problem solving, that is, in understanding the problem, analyzing it, exploring possible solutions, writing notes about partial results, reading about relevant methods, solving the subproblems, checking results, and eventually assembling the final solution. During this process, almost no mechanical help is available. Moreover, only a small part of the final assembly of notes and explanations ever becomes part of the formal code.

How is the solution to a mathematical problem presented? It is often in the form of a *proof*, which may be a sequence of equations or a sequence of lemmas and previously proved theorems involving elaborate nonequational reasoning such as induction and case analysis. The solution displays the result of the problem-solving process in such a way that the difficult steps are explained and exposed to public scrutiny.

How is the solution to a programming problem presented? In extreme cases of inadequate *documentation* the program may be presented *raw*, without explanation. In that case, there is no trace in the final product of the intellectual effort that went into producing it. More typically, the solution is presented as a program plus imprecise documentation written in natural language (usually produced in haste after the program has been written). This is especially bad because a good explanation may be more important than the program, especially if the program must later be modified or if it becomes critical to know its correctness. Yet the task of reconstructing an explanation from the formal code or from the informal comments is very difficult compared to the reverse process.

We are interested in finding ways to help the programmer carry out the most difficult and important part of his task: solving the problem and explaining the solution. We are interested in finding ways for computers to help produce and subsequently use good explanations. To see how this might be done, let us examine the sample problem further.

The first task is to make the problem specification precise. We introduce necessary definitions. Given a sequence of integers of length n, say $[a_1, a_2, \ldots, a_n]$, we say that a subsequence of the form $[a_i, a_{i+1}, \ldots, a_{i+p}]$ is a *segment*, that

[2] Some programming problems have a more explicit computational flavor which distinguishes them even more from ordinary mathematics (e.g., find a maximum segment sum using at most $O(n)$ steps or using n processors concurrently). Other programming problems are distinguished by mention of data types such as payroll files, which are not common to mathematical discourse.

is, a sublist of adjacent elements. A segment can be specified by giving the index of its first member and then either the length or the index of its last member. The task before us is to find a sum $\sum_{j=i}^{i+p} a_j$ that is maximum among all segment sums.

We can now write the problem specification precisely in mathematical notation: find M such that

$$M = \max\left(1 \le k \le q \le n: \sum_{j=k}^{q} a_j\right).$$

Since the set over which we are computing the maximum is finite (otherwise the maximum is not even a well-defined operation), the value M can be computed by brute force; for example, just list all segments, compute their sums, and take the largest. In noncomputational mathematics one might proceed in this inefficient way, but the essence of computer science is to compute well.

Confronted with a problem of the structure "for all n find a p such that $A(p, n)$" there are really only a few tactics for solving it. One possibility is that the construction of p and proof of $A(p, n)$ is uniform in n, as in the example "for all n find p such that p is not divisible by any $y \le n$." Here we take $p = n! + 1$ and prove the proposition without regard for the structure of n. The possibilities for such a uniform analysis depend heavily on which functions are available for building p and the proof of $A(p, n)$ directly.

Another possibility is that we proceed by induction of one form or another on n. This is suggested whenever the answer p must be built in stages. Another possibility in problems of this sort is that some property of $A(n, p)$ can be generalized to add an extra parameter, say $A(m, n, p)$, and then we can use induction on m. This technique is called *generalization* in Polya [29]; Dijkstra [16], Gries [18], and Reynolds [32] call it *weakening* in the context of programming.

Notice that the formal specification of the problem has suggested the methods for solving it. Induction is suggested from among them because the problem can be solved trivially for sequences of length one, and it seems likely that we can decide uniformly how to solve it after adding one new element.[3] So suppose it has been solved for sequences of length n, yielding sum M on the segment at i of length p. Suppose now that we add a new element a_{n+1}. Then the following possibilities are exhaustive.

(1) The new maximum sum does not include a_{n+1}.
(2) The new maximum sum does include a_{n+1}.

How can we tell whether (1) or (2) holds? We cannot simply compare M with $M + a_{n+1}$ because M may be the sum of a segment no longer contiguous with a_{n+1}. We really must know how large a sum is possible from a segment ending at $n + 1$ (i.e., how large a sum can be found by moving back into the sequence to

[3] In the context of a programming logic, we can consider the technique of while-induction and its loop invariant as just another proof technique. See [14] for a treatment of this rule in the style of the present paper.

form $[a_j, \ldots, a_{n+1}]$. Call the maximum such sum L_{n+1}. Knowing L_{n+1}, we can take the new maximum sum to be $\max(M, L_{n+1})$.

Suppose we try to compute L_{n+1}. Since we are proceeding inductively, we will know L_n. (Clearly, L_1 will be a_1.) How to compute L_{n+1} from L_n? The value L_{n+1} will be $L_n + a_{n+1}$ unless $L_n + a_{n+1} \leq a_{n+1}$, in which case we know that the maximum segment sum including a_{n+1} is simply $[a_{n+1}]$. So the computation of L_{n+1} is $\max(a_{n+1}, L_n + a_{n+1})$. This analysis tells us precisely how to solve the problem.

Insight was necessary to introduce the concept of L_{n+1}, but the structure of the problem led us to realize that L_n would be available. Moreover, the structure focused our attention on a relatively simple subproblem of finding M for a sequence of length $n + 1$ (called M_{n+1}), given the sum for a sequence of length n (say M_n).

These observations can be more compactly described in terms of properties of the maximum operation. Notice that taking the maximum of a two-argument function is equivalent to iterating the maximum operation on one-argument functions, as follows:

(1) $\max(1 \leq k \leq q \leq n: f(k, q)) = \max(1 \leq q \leq n: \max(1 \leq k \leq q: f(k, q)))$.

To extend the range of the sum from n to $n + 1$, we have

(2) $\max(1 \leq k \leq q \leq n + 1: f(k, q)) = \max\{\max(1 \leq q \leq n: \max(1 \leq k \leq q: f(k, q))), \max(1 \leq k \leq n + 1: f(k, n + 1))\}$.

Taking $f(k, q) = \sum_{j=k}^{q} a_j$, notice that $\max(1 \leq q \leq n: \max(1 \leq k \leq q: \sum_{j=k}^{q} a_j)) = M_n$ and $\max(1 \leq k \leq n + 1: \sum_{j=k}^{n+1} a_j) = L_{n+1}$. Thus the second equation says that $M_{n+1} = \max\{M_n, L_{n+1}\}$. In addition, since we know

$$(3)\quad \max\left(1 \leq k \leq n + 1: \sum_{j=k}^{n+1} a_j\right) = \max\left\{\max\left(1 \leq k \leq n: \sum_{j=k}^{n} a_j + a_{n+1}\right), a_{n+1}\right\},$$

and

$$\max\left(1 \leq k \leq n: \sum_{j=k}^{n} a_j + a_{n+1}\right) = \max\left(1 \leq k \leq n: \sum_{j=k}^{n} a_j\right) + a_{n+1},$$

it follows that

$$L_{n+1} = \max(L_n + a_{n+1}, a_{n+1}).$$

Thus the entire body of the inductive proof can be described as algebraic transformations of the maximum function. (See [10] for a formal proof along those lines.)

Outline of the Paper

We use the maximum segment sum example in Section 2 of the paper to illustrate the ideas that motivated the design of our program refinement system, called PRL. In Section 3 we describe the PRL system very briefly and show how the example is treated using it. In the conclusion, Section 4, we reflect on some of the general ideas raised by this new kind of programming system. We hope to

impart to the reader some sense of the unique style of programming made possible by systems of this kind, of which PRL is the first example. We do not intend to explain in this short paper how PRL works nor what new technical ideas have emerged from using it. There are references given for such matters.

2. PATTERNS OF EXPLANATION

Experience in Mathematics

How can we explain the solution to a programming problem? If we look to recent common practice we see how not to explain it, namely, by comments, written in pidgin English, attached to the code. If, on the other hand, we look to mathematics, where the issue has been of concern for hundreds of generations, then we see successful paradigms. In particular, the concept of a proof has been developed to convey complex and detailed explanations. A proof serves to organize all the information needed to solve a problem and, moreover, introduces information according to specific needs.

The notion of a proof serves not only to organize informations, but to direct the analysis of a problem and produce the necessary insights. It is as much an analytical tool as it is a final product [25]. It is this feature of proofs that our system will exploit to aid the programmer.

Some of the methodology of mathematics has been codified in the style of its presentation, the heart of which is the proof. The pattern of definition, theorem, remark, definition, lemma, example, and so forth carries with it a tradition of explanation. The mathematician is taught to decompose theorems into sequences of lemmas, to build abstraction upon abstraction using definitions to hide details in these abstractions, and to illustrate delicate cases or blind alleys by examples. In the context of mathematical investigations, many great minds, such as Descartes, Leibniz, Poincaré, and Polya have addressed the problem of *method*, of how we know and how we explain. They have discussed "rules" for discovery of proofs and "guidelines" for writing them. It is in this context that we can explore various means of solving programming problems. We can encourage proof presentation by successive refinement. We can compare various ways of filling in detail (e.g., "top-down" and "bottom-up"). We can even provide "rules of programming" to help people learn how to solve algorithmic problems.

Formality

For programming problems, such as our example, that deal with elementary (first order) properties of numbers and finite sequences, we know how to be precise about the notion of a *problem*, a *solution*, and an *explanation*. A *problem* is a formula in a logical theory, the *solution* is some computable function, and the *explanation* is a formal proof. A typical 1970s conceptualization of programming requires only that the computable function description be formal. But in fact it appears that there are substantial reasons to formalize the explanations as well.

The principal reason to formalize the explanation is that it becomes a real data object. We can obtain mechanical assistance in generating, checking, modifying, and using it in other unforeseen ways. It then becomes a mathematical object in its own right, like an integer, and we can learn to compute with it.

To illustrate the role of explanations, let us consider a proof of the existence of a maximum segment-sum of a list of integers. We convert the analysis of Section 1 into a careful proof. Anticipating an interest in formalizing this proof, we use the notation of symbolic logic to describe the problem. The connectives "and," "or," "implies," and "not" are represented by &, V, $\Rightarrow$, and $\sim$ respectively; and the quantifiers "for all integers x", "for all lists A", and "for some natural number y" are represented as "*all x:int*", "*all A:list*", and "*some y:nat*", respectively. We use $|A|$ to denote the length of a list, and $A(i)$ to denote the ith element of a list, provided $0 < i \leq |A|$, and to denote 0 otherwise.

One proof results from defining the maximum operation and proving the properties used in Section 1. The definition of $\max(1 \leq q \leq k \leq n: f(q, k))$ can be given in terms of $\max(1 \leq k \leq q: h(k))$, which can in turn be defined recursively as

$$\max(1 \leq k \leq 1: h(k)) = h(1)$$

$$\max(1 \leq k \leq q + 1: h(k)) = \max\{\max(1 \leq k \leq q: h(k)), h(q + 1)\}.$$

Notice that the parameter q has the type of the natural numbers *nat*, and the type of h is that of a function from *nat* to *integers*. Thus max is a "second order operation." This rather natural occurrence of so called higher level operations explains why we are interested in a very expressive type theory in PRL(see [10]). However, the core PRL theory, to be described in Section 3, does not include second order operations.

A proof which follows the first method of analysis of Section 1 can be directly formalized in the core PRL language, to be described later. This proof does not appeal to general properties of the maximum operation, but deals instead entirely with integers and lists. Since the PRL logic uses lists of integers as a primitive data type, we cast the problem in these terms. Lists are naturally built from the head, so it is convenient to recast the argument used above in terms of adding a new element $a1$ at the head of A. It is also convenient to perform the induction on the list A itself rather than on its length. These modifications actually simplify the formal treatment of the problem; but there would be no essential difficulty formalizing exactly the first argument used above.[4]

all A:list.some$M$:int.some I:int.
(all p,q:int.$(1 \leq p \leq q \leq |A| \Rightarrow M \geq \sum_{j=p}^{q} A(j))$) &
(some a,b:int.$(M = \sum_{j=a}^{b} A(j))$) &
(all p:int.$(1 \leq p \leq |A| \Rightarrow I \geq \sum_{j=1}^{p} A(j))$) &
(some e:int.$(I = \sum_{j=1}^{e} A(j))$)

Proof by list induction $a.B$

Basis case: $A = [\,]$ (A is the empty list)
choose $M = I = 0$, notice that $|A| = 0$, so that there are no p, q in the interval. Also notice that $\sum_{j=1}^{0} A(j) = 0$, so the choices of a, b, e are 1.

[4] In fact, to test PRL's optimizations on list processing, we did formalize that argument as well. It is interesting to observe that to a human these two arguments are nearly identical, yet their external structure is grossly different.

Induction case: $A = a.B$, and we assume the result for list B.

Since the base case is degenerate, we must consider whether B is empty or not. We know $B = [\,] \mid\mid \sim B = [\,]$ as a basic axiom about lists. Now we proceed by cases on this disjunction.

Case $B = [\,]$. In this case the maximum segment sum is the element a, which is also the maximum initial segment sum.

Case $\sim B = [\,]$. In this case we choose MB and IB as the values of the maximum segment sum and maximum initial segment sum for the list B which satisfies the induction hypothesis. Now we proceed in two major steps.

(1) We first determine I. The claim is that $I = \max(a, a + IB)$
We must show that
(i) for all p in $1 \le p \le |A|$,

$$I \ge \sum_{j=1}^{p} A(j) \quad \text{and}$$

(ii) for some e, $I = \sum_{j=1}^{e} A(j)$.

Consider (i), take any p in $1 \le p \le |A|$;
for $p > 1$,

$$\sum_{j=1}^{p} A(j) = a + \sum_{j=2}^{p} A(j) = a + \sum_{j=1}^{p-1} B(j).$$

Since $1 \le p - 1 \le |B|$, we know that $IB \ge \sum_{j=1}^{p-1} B(j)$, thus

$$a + IB \ge a + \sum_{j=1}^{p-1} B(j) = \sum_{j=1}^{p} A(j);$$

for $p = 1$,

$\sum_{j=1}^{1} A(j) = A(1) = a$, and clearly $a \ge a$. Thus for all p in the interval,

$\max(a, a + IB) \ge \sum_{j=1}^{p} A(j)$.

Consider (ii), if $I = a$, then $e = 1$ and if $I = a + IB$, then if

$$IB = \sum_{j=1}^{e} B(j) \qquad \text{then} \quad I = \sum_{j=1}^{e+1} A(j)$$

because index e in list B is position $(e+1)$ in list $a.B$, so $(e+1)$ is the new end of the maximum initial segment.

(2) Next we determine M, the claim is that $M = \max(MB, I)$. We must show
(i) for all p, q where $1 \le p \le q \le |A|$,

$$M \ge \sum_{j=p}^{q} A(j) \quad \text{and}$$

(ii) for some a, b, $M = \sum_{j=a}^{b} A(j)$.

Consider (i), for any p, q in $1 \le p \le q \le |A|$; if we have $1 < p$ so that $2 \le p \le q \le |A|$, then for any j in $2 \le j \le |A|$, $A(j) = B(j-1)$. Therefore in this

range,

$$MB \geq \sum_{j=p}^{q} A(j) = \sum_{j=p-1}^{q-1} B(j).$$

For $p = 1$ and $1 \leq q \leq |A|$, we know from 1(i) that

$$I \geq \sum_{j=1}^{q} A(j).$$

Thus taking $M = \max(MB, I)$, we have $M \geq \sum_{j=p}^{q} A(j)$.

Consider (ii), if $\max(MB, I) = MB$, then

$$MB = \sum_{j=a}^{b} B(j) = \sum_{j=(a+1)}^{(b+1)} A(j) \quad \text{since} \quad A(j+1) = B(j),$$

So the new limits of the maximum segment sum are $(a+1)$, $(b+1)$. If $\max(MB, I) = I$, then $a = 1$ and $b = e + 1$ follow from 1(ii).

This ends the induction part. □

This proof is constructive, in a sense made precise in Section 3. Intuitively, it is plausible that the proof is also a procedure for finding M and L. We know how to execute every step. That is, whenever the existence of a number is claimed, the proof shows how to calculate it from other numbers; these other numbers are found by applying the proof procedure to a smaller list. Whenever the proof proceeds by a case analysis on $P \vee Q$, there is a method of computing which of P or Q holds (e.g., $B = [\,] \mid \sim B = [\,]$).

Executing Proofs

The example has shown us more than we might have expected. It began as an example of an explanation, but the possibility arises that it can also become the complete solution because the proof itself, if formalized, can be executed. To see how this is possible in general, let us consider the meaning of various constructive statements.

A constructive proof of *some y:int. R(x, y)* will build a witness y for the assertion $R(x, y)$. For instance, a proof that *some y:int.* $(y > x)$ will give an expression for y such as $(x + 1)$ or $(2x + 1)$. The particular value chosen depends on the proof used. A proof that *some y: int.* $(y > x$ & prime$(y))$ will result in a method for finding a prime number greater than x.

A proof by induction of an existential statement such as *some y:int. R(x, y)* has the following pattern.

all x:list. some y:int. R(x, y)
Proof (by induction on x):
  Base: construct some value y_0 where $R([\,], y_0)$
  Induction: assume *some y:int. R(tl(x), y)*
    Show *some y:int. R(x, y)*
    The proof builds a particular term t for y using
    x and the value y' assumed to exist for
    $tl(x)$. So t can be denoted $t(x, y')$.
  Qed
Qed

The part of the proof building the value y is a recursive procedure of the form $p(x) = \textit{if } x = [\,] \textit{ then } y_0 \textit{ else } t(x, p(tl(x)))\textit{fi}$. As long as the expressions y_0 and $t(x, y)$ are computable, so is the procedure. Indeed, we observe that p is an example of a *primitive recursive procedure* [23].

A constructive formal system has the property that every proof step can be interpreted as a construction. This is explained in [22], and applied to programming in [9, 2]. So in fact it makes sense to *execute* constructive formal proofs. In the case of a proof of $\textit{all } x\textit{:list}\,.\ \textit{some } y\textit{:int}\,.\ R(x, y)$, the proof is a function p such that $R(x, p(x))$.

Although constructive proofs can be executed in principle, it is not yet known how efficiently this can be done. It might be necessary to pursue the solution further in the direction of known mechanisms for efficient computation, such as those available in high-level languages such as ALGOL. But it is possible to go in this direction within the context that regards proof as explanations and explanation as the most important product of programming. This can be done by treating commands such as assignment as part of the logical system, as was done in [13] and in [2], for example. But in the course of the work reported in [2], it became increasingly plausible that the commands were not necessary either for efficient execution or for cogent explanation. Indeed, the language without commands was far simpler to explain and appeared tractable to implement. So the goal of our program refinement research became that of building and testing an implementation of a constructive theory of mathematics. The key new ingredients would include a component to extract code from constructive proofs, called an *extractor*, and an interactive proof-generating environment to help the user build formal proofs. This *proof synthesizer* would encourage a top-down refinement style of proof construction, as described in [2, 24]. It would also employ the technology of modern programming environments, especially the Cornell Program Synthesizer [37]. The work of Dean Krafft in building a synthesizer environment for PL/CV2, called AVID [24], provided encouraging evidence that such systems would make the task of formal proof-generation tolerable for a logic sufficiently close to PL/CV2.

In the next section we describe some aspects of the logic and system resulting from achieving these goals.

3. THE PRL LOGIC

Background

The method of treating a proof as a program is applicable in a variety of constructive theories, from those about numbers to those about sets. A considerable part of our effort on the "PRL project" has been spent in defining a very general theory in which these methods work; this is a *type theory* in the sense of Whitehead and Russell [38], deBruijn [12], Martin-Löf [27], Andrews [1], and the related work of Constable [3, 10, 11, 12]. Some of our ideas are presented in [3, 10]. Considerable effort was also spent in designing a system with which to support this very general theory. The system should provide a modern environment for interactive proving and problem solving, and be based on the notion of a library of results organized into books, chapters, sections, and so on. The user should have help in generating material for publication in the library and a

"smart" editor for viewing and modifying results in the library. There should also be a means of invoking formal metareasoning to extend the system safely, building, for example, guaranteed proof tactics [11, 17] and heuristic problem solvers [4, 7, 8, 40].

There are many difficult technical problems associated with building a system of this generality (discussed elsewhere). This complexity led us to an incremental development of the system and its logic. We began with a core logic of integers and lists of integers and a core system supporting a simple library, a structure editor similar to the Cornell Program Synthesizer [37] and AVID [24], but with user-defined templates, a proof extractor based on Bates' thesis [2], a metareasoning facility obtained by embedding PRL in Edinburgh LCF as an object theory [17], and a decision procedure combining a simplifier, a congruence closure equality reasoner, and an arithmetic reasoner over integers. This is the core version of PRL, which is implemented, and which we briefly describe in this section.

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_i are terms and op is an operator, and *listings* $[e_1, \ldots, e_n]$, where e_i are integer terms. The *constants* include nonnegative decimal numerals, 0, 1, 2, . . . , the unary function, the infix binary operators +, −, *, /, and various atomic functions: *mod*, *hd*, *tl*, and · . The function constants *mod*, *hd*, *tl*, and · have the types:

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

The atomic formulas of the theory are $e_1 = e_2$ for arbitrary terms e_i of the same type and $e_1 < e_2$ for e_i of type integer.

Compound formulas are $\sim A$, $A \& B$, $A \mid B$, $A \Rightarrow B$ for A and B formulas. The usual precedence holds among these connectives (not, and, or, implies): ~, &, |, ⇒; and ⇒ is right associative. In addition, compound formulas include

$$\begin{aligned} &all\ x_1, \ldots, x_n\text{: type} \ .\ A \\ &some\ x_1, \ldots, x_n\text{: type} \ .\ A \end{aligned}$$

where A is a formula and x_i are variables. Quantifiers bind more weakly than connectives, so they have a wide scope. *Free* and *bound* variables are defined as usual.

An *environment* env is a list of variables and their types such as [x:int, B:list]. We use [e] ∪ x:int to denote appending x:int to [e]. A *goal* has the form

$$[\text{env}]\text{Assm} \gg \text{conc}$$

where Assm is a list of formulas called the *assumptions* and conc is a single formula called the *conclusion.*

A *proof* is an expression of the form

goal *by* rulename
p_1
.
.
.
p_n

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 [13], see also Shostak [35]); (3) list rules; (4) rules to reference the library and defined objects; and (5) rules to invoke tactics built in the metalanguage ML of Edinburgh LCF. Here we illustrate some of these, starting with the predicate calculus rules. We use the notation $B(t/x)$ to denote the formula obtained from B by substituting the term t for all free occurrences of x in B. In the elimination rules assume that n is the number of the formula between S and S'.

&	$S \gg A\&B$ by intro 1. $S \gg A$ 2. $S \gg B$	$S,A\&B,S' \gg G$ by elim n 1. $S,S',A,B \gg G$
|	$S \gg A \mid B$ by intro 1 1. $S \gg A$ $S \gg A \mid B$ by intro 2 1. $S \gg B$	$S,A \mid B,S' \gg G$ by elim n 1. $S,S',A \gg G$ 2. $S,S',B \gg G$
$\Rightarrow$	$S \gg A \Rightarrow B$ by intro $S,A \gg B$	$S,A \Rightarrow B,S' \gg G$ by elim n 1. $S,A \Rightarrow B,S' \gg A$ 2. $S,A \Rightarrow B,S',A,B \gg G$
all	$[e]\ S \gg$ all $x{:}A.B$ by intro $[e] \cup x{:}A\ S \gg B$	S,all $x{:}A.B,S' \gg G$ by elim n,t $S,S',B(t/x) \gg G$
some	$S \gg$ some $x{:}A.B$ by intro t $S \gg B(t/x)$	$[e]\ S$,some $y{:}A.B,S' \gg G$ by elim n $[e] \cup y{:}A\ S,S',B \gg G$
consequence	$S \gg G$ by seq A 1. $S \gg A$ 2. $S,A \gg G$	
hypothesis	$S,A,S' \gg A$ by hyp n	
false elim	S,false,$S' \gg G$ by elim n	

The induction rule for integers has this form when specialized to base 0.

$[e]\ S \gg$ all x:int.P by ind

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

The variable x cannot yet appear in e; if it does, the rule can be called "ind y", where y is a new variable that will be used in the hypotheses in place of x.

One can also perform induction with k base cases, say $b_1, \ldots, b_k$. The induction rule for lists has this form when specialized to base [ ].

$[e]S \gg$ all A:list.P by ind $x.B$
1. $S \gg P([\,]/A)$
2. $[e] \cup x$:int $\cup$ B:list S, $P[B/A] \gg P(x.B/A)$

The variables x, B must not yet appear in e; if they do, they can be renamed, as in the integer induction case.

Libraries

The PRL system helps the user generate a *library* of results and allows users certain operations on the members of the library. The results can be of four kinds.

(1) (THM) named proofs of theorems, whose syntax is: name *thm* proof. The proof can be used subsequently by mentioning the name in the lemma reference rule.

(2) (EXT) named functions extracted from proofs, with syntax: name *extract* theorem name, where the theorem must be of the form *all* $x_1 \ldots x_n$: type.*some* y: type.P. The function may be used in subsequent applications.

(3) (DEF) definition of new notation in terms of existing notations, syntax: name *def* template == right-hand side, where template is a list of characters and parameters of the form ⟨id:comment⟩, and the right-hand side is any piece of text with interspersed parameter references. For example, | ⟨A:list⟩ | == length (⟨A⟩) defines the notation $|A|$ to be the length of list A. Library members constructed after such a definition may use the notation of the template, with the system construing their meaning as the right-hand side. Since PRL editors are structure editors, these user-defined notations are never parsed, hence there are no restrictions on the notations defined, except that they be displayable.

(4) (REC) primitive recursive definitions of functions over *int* or *list*. The syntax for integer definitions is

$f(x$:int, $\ldots)$:type =
$x \Rightarrow t_0$
$a \Rightarrow t_1$
$\vdots$
$b \Rightarrow t_{n-1}$
$x \Rightarrow t_n$

with the property that for any integer x and other arguments to f,

$x < a \Rightarrow f(x, \ldots) = t_0$
$x = a \Rightarrow f(x, \ldots) = t_1$
$\vdots$
$x > b \Rightarrow f(x, \ldots) = t_n$

Given that $t_1 \ldots t_{n-1}$ have no reference to f, t_0 may invoke f recursively, with first argument $x + c$ and arbitrary other arguments; for c, a constant between 1 and

$b - a + 1$, t_n may invoke f recursively with first argument $x - c$ and arbitrary other arguments.

The syntax for list definitions is

$f(x$: list, ...):type =
$0 \Rightarrow t_0$
$\vdots$
$n \Rightarrow t_n$
$x \Rightarrow t_{n+1}$

with the property that for any list x and other arguments to f,

length $x = 0 \Rightarrow f(x, \ldots) = t_0$
$\vdots$
length $x = n \Rightarrow f(x, \ldots) = t_n$
length $x > n \Rightarrow f(x, \ldots) = t_{n+1}$

given that $t_0, \ldots, t_n$ do not invoke f, and t_{n+1} may call f recursively with first argument tlx, $tltlx$, ... or $tl^{n+1}x$, and arbitrary other arguments. It is possible to define several mutually recursive functions in one definition block, but the schemes of all the functions must be identical.

System Features

Interaction with the PRL system is through a command language and two editors, *ted* for editing text and *red* for editing refinement proofs. There are commands to manipulate the library, invoke the editors, evaluate functions, and control input/output.

Objects are entered in the library using one of the two editors; preparatory to building them a named place is created in the library by the command

create ⟨name⟩⟨kind⟩⟨location⟩

where DEF, REC, THM, EXT are the kinds—for example, create thm2 THM after thm1. The labels DEF, REC, and so on refer to the kinds of objects catalogued above. The *status* of an object is indicated by one of the symbols ?, —, #, * before the name. A * denotes a complete and correct object; a # denotes an incomplete object, say a partial proof; a — denotes a bad object, say an erroneous proof; and a ? denotes a raw or unchecked object (perhaps an empty location).

The editors are invoked by viewing an object. If one views a proof, the proof editor is automatically used, otherwise the text editor. The text editor is also used to edit the statements and proof rules of theorems. Various aspects of the system operation are illustrated in the examples that follow.

Sample Scenario

Here are fragments of a session with PRL, to prove the maximum segment- sum example. Already built are small libraries of facts about integers and lists, among

these facts are the following definitions and theorems:

```
* ge
  DEF ⟨int⟩ ≥ ⟨int⟩
* ai
  DEF ⟨list⟩[int]
* inddef
  REC index: int, list → int
* len
  DEF | ⟨list⟩ |
* interval
  DEF ⟨int⟩ ≤ ⟨int⟩ ≤ ⟨int⟩
* max
  THM ≫ all x, y:int.some z:int.(x < y ⇒ z = y)&(~x < y ⇒ z = x)
* maxf
  EXT max:int, int → int
```

By viewing these objects we can see their structure, for example, the command "view ge" reveals,

$\langle a$:int$\rangle \geq \langle b$:int$\rangle$ == $\sim\langle a\rangle < \langle b\rangle$

and "view ai" reveals,

$\langle a$:list$\rangle[\langle n$:int$\rangle]$ == index$(\langle n\rangle, \langle a\rangle)$

and "view index" reveals,

```
index(i:int, a:list):int =
  1 ⇒ hd a,
  i ⇒ index(i − 1, tl a)
```

The command "view interval" reveals,

$\langle a$:int$\rangle \leq \langle b$:int$\rangle \leq \langle c$:int$\rangle$ == $\sim\langle b\rangle < \langle a\rangle$ & $\sim\langle c\rangle < \langle b\rangle$

These definitions enter the proof essentially as macros.

The development of the maximum segment sum example begins with the definition of the sum of a list a, regarded as an array, from index p to index q. Here are the details.

(1) *Create recursive definitions.* Let sum4$(n, i, a) = a[i] + a[i + 1] + \cdots + a[i + n]$. This is defined recursively as

```
sum 4(n:int, i:int, a:list):int =
  0 ⇒ a[i],
  n ⇒ a[i] + sum4(n − 1, i + 1, a);
```

The *downward* case, $n < 0$, is omitted, resulting in the default that sum4$(n, i, a) = 0$ for $n < 0$. The *base* case is sum4$(0, i, a) = a[i]$, and the *up* case is the last line. These clauses are referred to in proofs by the *definition rule*, as in

```
def sum4(n, i, a) up
```

which results in the new assumption,

$\text{sum4}(n, i, a) = a[i] + \text{sum4}(n - 1, i + 1, a)$

given that we show $0 < n$.

The summation operation we really want is $\text{sum3}(p, q, a) = a[p] + a[p + 1] + \cdots + a[q]$, which is defined as

```
sum3(p:int, q:int, a:list):int =
  0 ⇒ sum4(q, p, a),
  p = sum4(q − p, p, a);
```

We prove these facts about sum3.

```
* sum3thm1
  THM ≫ all a:list. all n:int.
    1 ≤ n ⇒ sum3(n, n, a) = a[n]

* sum3thm2
  THM ≫ all n:int, all a:list. all p, q:int.
    1 ≤ p ≤ q & n = q − p ⇒
    sum3(p, q, a) = a[p] + sum3(p + 1, q, a)

* sum3thm3
  THM ≫ all n:int. all a:list. all p, q:int.
    1 ≤ p ≤ q & n = q − p ⇒
    sum3(p, q, a) = sum3(p, q − 1, a) + a[q]
```

The next step is to create certain definitions specific to the maximum segment sum example.

(2) *Create specialized definitions.* We need to mention those special segments that start at the first index, which we call *initial* segments. Here are some definitions.

```
initsegment
DEF
⟨a:int⟩ is as large as any initial segment sum of ⟨A:list⟩
== all i:int. 1 ≤ i ≤ |⟨A⟩| ⇒ ⟨a⟩ ≥ sum3(1, i, ⟨A⟩)

endinitseg
DEF
⟨a:int⟩ is the initial segment sum ending at ⟨e:int⟩ of ⟨A:list⟩
== ⟨a⟩ = sum3(1, ⟨e⟩, ⟨A⟩)
```

We need also the definition of a segment and the condition for a segment to be maximum. These are

```
segdef
DEF
⟨n:int⟩ is the sum of the segment of ⟨A:list⟩
from ⟨p:int⟩ to ⟨q:int⟩
==(⟨n⟩ = sum3(⟨p⟩, ⟨q⟩, ⟨A⟩))
```

```
maxseg
DEF
⟨n:int⟩ is as large as any segment of ⟨A:list⟩
== (all p, q:int. 1 ≤ p ≤ q ≤ |⟨A⟩| ⇒ ⟨n⟩ ≥ sum3(p, q, ⟨A⟩))
```

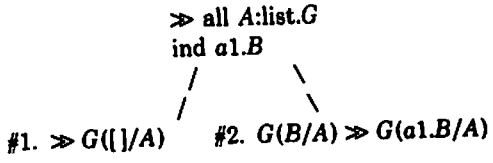
We are now prepared to state the main theorem.

(3) *Edit main theorem goal.* We create a place in the library for the main theorem and then edit the goal, resulting in the following statement:

```
|# top
|[]
|≫ all A:list.some M:int.some I:int.
|  M is as large as any segment sum of A &
|  (some a, b:int.M is the sum of the segment of A from a to b) &
|  I is as large as any initial segment of A &
|  (some e:int.I is the initial segment sum ending at e of A)
```

Next, we outline a proof of the theorem including only those steps needed to execute it.

(4) *Build a proof skeleton.* (i) From the informal sketch in Section 2 we know that we can proceed by list induction with base case nil. We issue the command: ind $a1.B$. We choose $a1$ to suggest $A[1]$, and we use B as a simple name for the tail of A. We know $A = a1.B$. The result of this command is a proof tree with the structure:



Here is the actual terminal output:

```
|# top
|[]
|≫ all A:list.some M:int.some I:int.M is as large as any segment sum of A
|    & (some a, b:int.M is the sum of the segment of A from a to b)
|    & I is as large as any initial segment of A
|    & (some e:int.I is the initial segment sum ending at e of A)
|
|BY ind a1.B
|
|1•[]
|  ≫ some M:int.some I:int.M is as large as any segment sum of [ ]
|      & (some a, b:int.M is the sum of the segment of [ ] from a to b)
|      & I is as large as any initial segment of [ ]
|      & (some e:int.I is the initial segment sum ending at e of [ ])
|
|2# [int a1; list B]
|  1. some M:int.some I:int.M is as large as any segment sum of B
|      & (some a, b:int.M is the sum of the segment of B from a to b)
|      & I is as large as any initial segment of B
```

```
|     & (some e:int.I is the initial segment sum ending at e of B)
| ≫ some M:int.some I:int.M is as large as any segment sum of a1.B
|     & (some a, b:int.M is the sum of the segment of a1.B from a to b)
|     & I is as large as any initial segment of a1.B
|     & (some e:int.I is the initial segment sum ending at e of a1.B)
==========
```

A proof of the base case is trivial because sum3($i, j, [\,]$) = 0 for all i, j. We only need to instantiate each existential quantifier with 0, and trivial reasoning will finish the proof.

The informal proof of the induction case has the structure:

```
B = [ ]| ~B = [ ]
case B = [ ]
  choose M = I = a1.
case ~B = [ ]
  1. choose values of I, M for the list B,
       call them IB, MB
  2. instantiate the quantifiers for M and I
     with maxf(MB, maxf(a1, a1 + IB)) and
       maxf(a1, a1 + IB), respectively
  3. prove the four subgoals
       G1: M is maximum
       G2: M is a segment
       G3: I is maximum initial
       G4: I is an initial segment
```

For the purpose of executing the proof, we only need to instantiate the quantifiers. Here are the subgoals under 2, as they appear in the actual proof.

```
| BY intro maxf(MB, maxf(a1, a1 + IB))
|
| 1# [int a1, MB, IB; list B]
|   1. MB is as large as any segment sum of B
|      & (some a, b:int.MB is the sum of the segment of B from a to b)
|      & IB is as large as any initial segment of B
|      & (some e:int.IB is the initial segment sum ending at e of B)
|   2. ~B = [ ]
|   ≫ some I:int.maxf(MB, maxf(a1, a1 + IB)) is as large as any segment sum of a1.B
|      & (some a, b:int.maxf(MB, maxf(a1, a1 + IB)) is the sum of the segment of a1.B
|      from a to b)
|      & I is as large as any initial segment of a1.B
|      & (some e:int.I is the initial segment sum ending at e of a1.B)
==========

| BY intro maxf(a1, a1 + IB)
|
| 1# [int a1, MB, IB; list B]
|   1. MB is as large as any segment sum of B
|      & (some a, b:int.MB is the sum of the segment of B from a to b)
|      & IB is as large as any initial segment of B
|      & (some e:int.IB is the initial segment sum ending at e of B)
|   2. ~B = [ ]
```

```
|  ≫ maxf(MB, maxf(a1, a1 + IB)) is as large as any segment sum of a1.B
|     & (some a, b:int.maxf(MB, maxf(a1, a1 + IB)) is the sum of the segment of a1.B
       from a to b)
|     & maxf(a1, a1 + IB) is as large as any initial segment of a1.B
|     & (some e:int.maxf(a1, a1 + IB) is the initial segment sum ending at e of a1.B)
==========
```

(5) *Execute the partial proof.* It is interesting to evaluate the partial proof to see whether the specification of the problem is what we expected. We can also discover properties of the function which may lead to modification of the specifications.

Here is a picture of the evaluation from a PRL session.

```
* maxsegf:list → int
eval maxsegf([−1, 0, 1, 2, −2, 4]) = 5
```

It is important to realize that we are seeing a *new mode of programming* here which is unique to PRL. The proof skeleton that we are executing is comparable in many ways to a program for the specified task. One important way in which it is comparable is that its size and the time taken to produce it are of the same order of magnitude as the size and time taken to produce a program. So PRL can be used comfortably as a programming language. But in these incomplete proofs there is already critically more information than in the pure program.

The PRL partial proof contains an outline of an explanation about why the problem is solvable. There may be only a few more lines of text than in a commented program, but these lines are known to be pertinent to a detailed and rigorous explanation of why the problem is solvable. These lines provide the skeleton upon which a complete explanation can be built. In the case of a program for this task, there is no formal reason that any explanation be included at all, let alone one formally related to the code.

We look next at the structure of the remainder of the proof. As we provide more detail, our confidence in the argument increases until we reach the point of a complete proof. At this point we are in the realm of "verified programming." If we are confident of the underlying system and logic, then we are confident of the solution. Since the system is a fixed finite object, we can spend considerable resources in guaranteeing its correctness and thereby increase our confidence in an arbitrary number of solutions.

(6) *More proof detail.* Recall the structure of the inductive case. We instantiate the quantifiers and then prove the conjunction

$$S \gg G_1 \;\&\; G_2 \;\&\; G_3 \;\&\; G_4.$$

The clauses G_i are written in the order desirable for the problem statement; first there is information about M, the main value, then about I, the auxiliary value. But it is very important to notice from the informal proof that these subgoals must be proved in a particular order:

1. show G_3 then
2. show G_4 using G_3 then
3. show G_1 using G_3 and G_4 then
4. show G_2 using G_1.

If we attack these as independent subgoals we will fail, thus we cannot let the automatic decomposition of goals proceed without guidance. This typifies a weakness of completely automatic methods at this stage of our knowledge.

The way we structure this part of the proof is to first invoke the rule of consequence with G_3 & G_4. Here is the structure:

$S \gg G_1$ & G_2 & G_3 & G_4 by seq G_3 & G_4
1. $S \gg G_3$ & G_4
2. S, G_3 & $G_4 \gg G_1$ & G_2 & G_3 & G_4

To prove 1 we use seq G_3, which gives

$S \gg G_3$ & G_4 by seq G_3
#1. $S \gg G_3$
#2. $S, G_3 \gg G_3$ & G_4

These structures are revealed in the following snapshots of the terminal session building this proof.

```
| BY seq maxf(a1, a1 + IB) is as large as any initial segment of a1.B
|   & (some e:int.maxf(a1, a1 + IB) is the initial segment sum ending at e of a1.B)
|
==========
```

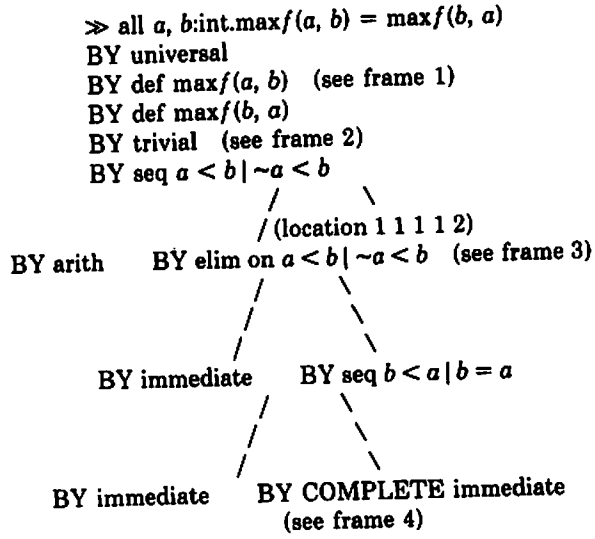
This rule generates the following subgoals, displayed on the screen exactly as shown below.

```
| EDIT THM mainthm2

| #...1111
| 1# [int a1, MB, IB; list B]
|   1. MB is as large as any segment sum of B
|        & (some a, b:int.MB is the sum of the segment of B from a to b)
|        & IB is as large as any initial segment of B
|        & (some e:int.IB is the initial segment sum ending at e of B)
|   ≫ maxf(a1, a1 + IB) is as large as any initial segment of a1.B
|        & (some e:int.maxf(a1, a1 + IB) is the initial segment sum ending at e of a1.B)
|
| 2# [int a1, MB, IB; list B]
|   1. MB is as large as any segment sum of B
|        & (some a, b:int.MB is the sum of the segment of B from a to b)
|        & IB is as large as any initial segment of B
|        & (some e:int.IB is the initial segment sum ending at e of B)
|   2. maxf(a1, a1 + IB) is as large as any initial segment of a1.B
|        & (some e:int.maxf(a1, a1 + IB) is the initial segment sum ending at e of a1.B)
|   ≫ maxf(MB, maxf(a1, a1 + IB) is as large as any segment sum of a1.B
|        & (some a, b:int.maxf(MB, maxf(a1, a1 + IB) is the sum of the segment of a1.B
|        from a to b)
|        & maxf(a1, a1 + IB) is as large as any initial segment of a1.B
|        & (some e:int.maxf(a1, a1 + IB) is the initial segment sum ending at e of a1.B)
==========
```

To finish this proof we need a number of lemmas about sum3 and max*f*. We show a complete proof of one of them, to offer a look at other important characteristics of PRL such as the use of proof tactics.

(7) *Complete proofs.* Here is an example of a very simple argument which can be produced on the system in about two minutes after the goal is entered. Proofs of this kind yield to a very simple heuristic search which introduces certain disjunctions such as $a < b \mid \sim a < b$. We have not explored such tactics very far, but we do rely heavily on tactics such as *trivial*, which perform the kinds of "immediate reasoning" widely exploited in our earlier work on PL/CV [13]. The tactic *trivial* searches for a proof using the elimination rules on &, ⇒, | and introduction rules on &, ⇒. It also performs all introduction, some elimination, and certain cases of all elimination and some introduction. The tree structure of the proof follows:



The reader navigates through this tree using a mouse (or a key pad) to descend or ascend. The screen is rapidly redrawn to display the new nodes. The frames displayed below are snapshots of the screen as we walk through the completed proof. However, no presentation of the proof on paper can capture the dynamic character of reading or creating it interactively.

Frame 1

```
| EDIT THM maxthm2
|
| * top 1
| [int a, b]
| ≫ maxf(a, b) = maxf(b, a)
|
| BY def maxf(a, b)
|
| 1* [int a, b]
|   1. (a < b ⇒ maxf(a, b) = b)&(~a < b ⇒ maxf(a, b) = a)
|   ≫ maxf(a, b) = maxf(b, a)
|
```

Frame 2

```
EDIT THM maxthm2
* top 1 1 1
[int a, b]
  1. (a < b ⇒ maxf(a, b) = b)&(~a < b ⇒maxf(a, b) = a)
  2. (b < a ⇒ maxf(b, a) = a)&(~b < a ⇒ maxf(b, a) = b)
  ≫ maxf(a, b) = maxf(b, a)

BY trivial

1* [int a, b]
  1. (a < b ⇒ maxf(a, b) = b)
  2. (~a < b ⇒ maxf(a, b) = a)
  3. (b < a ⇒ maxf(b, a) = a)
  4. (~b < a ⇒ maxf(b, a) = b)
  ≫ maxf(a, b) = maxf(b, a)
```

Frame 3

```
EDIT THM maxthm2
*... 1 1 1 2
[int a, b]
  1. (a < b ⇒ maxf(a, b) = b)
  2. (~a < b ⇒ maxf(a, b) = a)
  3. (b < a ⇒ maxf(b, a) = a)
  4. (~b < a ⇒ maxf(b, a) = b)
  5. a < b | ~a < b
  ≫ maxf(a, b) = maxf(b, a)

BY elim 5

1* [int a, b]
  1. (a < b ⇒ maxf(a, b) = b)
  2. (~a < b ⇒ maxf(a, b) = a)
  3. (b < a ⇒ maxf(b, a) = a)
  4. (~b < a ⇒ maxf(b, a) = b)
  5. a < b
  ≫ maxf(a, b) = maxf(b, a)

2* [int a, b]
  1. (a < b ⇒ maxf(a, b) = b)
  2. (~a < b ⇒ maxf(a, b) = a)
  3. (b < a ⇒ maxf(b, a) = a)
  4. (~b < a ⇒ maxf(b, a) = b)
  5. ~a < b
  ≫ maxf(a, b) = maxf(b, a)

==========
```

Frame 4

| * . . . 1 2 2 2
| [int a, b]
| 1. $(a < b \Rightarrow \max f(a, b) = b)$
| 2. $(\sim a < b \Rightarrow \max f(a, b) = a)$
| 3. $(b < a \Rightarrow \max f(b, a) = a)$
| 4. $(\sim b < a \Rightarrow \max f(b, a) = b)$
| 5. $\sim a < b$
| 6. $b < a \mid b = a$
| $\gg \max f(a, b) = \max f(b, a)$
|
| BY COMPLETE immediate
|

4. CONCLUSION

The version of PRL described here is a completed and working system which runs in a UNIX environment with Franz Lisp or on a Symbolics 3600 in Zetalisp, as described in the users' manual [31]. The system consists of about 33,000 lines of Lisp for PRL and 7000 lines for ML. PRL is used as a demonstration prototype, for teaching and for experiments.

Although there is much to be done to make a system like PRL competitive with conventional programming systems, even the existing implementation is a valuable tool for many kinds of problems. We were able to write the maximum segment sum proof skeleton in less than one hour. This skeleton is the direct analogue of a conventional program for the same task. In another hour of work we were able to reduce the correctness of this example to a few very simple lemmas such as sum3thm2. These lemmas are easy to understand because they are part of ordinary mathematics, and can be proved by transcribing the usual proofs, as we have illustrated. For a critical problem, it may be worth the effort to prove all of the lemmas. The more such systems are used, the easier they become, because libraries of useful facts and proof methods accumulate.

It is clear that systems such as PRL are much more than program development tools; they are also environments for doing formal mathematics. Similar tools for this purpose, such as symbolic mathematics packages and automatic theorem provers, will congregate around these nuclei. There will emerge areas of rigorous computational mathematics which will be done best in these computer-assisted environments. Programming in these realms will be done as we have illustrated here.

We have designed and implemented a richer logic based on a theory of types [10, 11, 27], called Nuprl; its structure is similar to that of PRL, since our original design was for this richer logic. However, the underlying logic is so expressive that in it we can formalize virtually any concept in computational mathematics. This kind of theory has opened many new opportunities for writing formal computational mathematics, from real analysis, graph theory, and automata theory to algebra and denotational semantics. Among the many promising directions to follow, our experience with proof tactics shows that developing the

formal metamathematics may be one of the most fruitful. This will be a major thrust for us in the next year.

With PRL and Nuprl, we have demonstrated that we know how to build a new kind of environment for programming. There is a large class of problems for which we claim that a production system of this kind is significantly superior to any alternative, extant or speculative.

ACKNOWLEDGMENTS

We would like to thank our colleagues on the PRL project for their advice and suggestions. We especially acknowledge the detailed criticisms of Fred Schneider and David Gries. David Gries also supplied us Jon Bentley's problem on segment sums. We also appreciate the efforts of Donette Isenbarger in preparing the manuscript.

REFERENCES

1. ANDREWS, P. B., COHEN, E. L., AND MILLER, D. A. A look at TPS. In *6th Conference on Automated Deduction. Lecture Notes in Computer Science, 138.* Springer-Verlag, New York, 1982, 50–69.
2. BATES, J. L. A logic for correct program development. Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., 1979.
3. BATES, J., AND CONSTABLE, R. L. Definition of Micro-PRL. Tech. Rep. TR 82-492, Computer Science Dept., Cornell Univ., Oct. 1981.
4. BISHOP, E. *Foundations of Constructive Analysis.* McGraw-Hill, New York, 1967.
5. BISHOP, E. Mathematics as a numerical language. In *Intuitionism and Proof Theory.* J. Myhill, et al., Eds., North-Holland, Amsterdam, 1970, 53–71.
6. BLEDSOE, W. Nonresolution theorem proving. *Artif. Intell. 9* (1977), 1–36.
7. BOYER, R. S., AND MOORE, J. S. *A Computational Logic.* Academic Press, New York, 1979.
8. BUNDY, A. *The Computer Modelling of Mathematical Reasoning.* Academic Press, New York, 1983.
9. CONSTABLE, R. L. Constructive mathematics and automatic program writers. In *Proceedings of IFIP Congress,* (Ljubljana, 1971), 229–233.
10. CONSTABLE, R. L., AND BATES, J. L. The nearly ultimate PRL. Dept. of Computer Science Tech. Rep. TR 83-551, Cornell Univ., Apr. 1983.
11. CONSTABLE, R. L. Intensional analysis of functions and types. Dept. of Computer Science Internal Rep. CSR-118-82, Univ. of Edinburgh, June 1982.
12. CONSTABLE, R. L., AND ZLATIN, D. R. The type theory of PL/CV3. *ACM Trans. Program. Lang. Syst. 6,* 1 (Jan. 1984), 94–117.
13. CONSTABLE, R. L., JOHNSON, S. D., AND EICHENLAUB, C. D. *Introduction to the PL/CV2 Programming Logic. Lecture Notes in Computer Science, 135.* Springer-Verlag, New York, 1982.
14. CONSTABLE, R. L. Programs as proofs. *Inf. Process. Lett. 16,* 3 (Apr. 1983), 105–112.
15. DEBRUIJN, N. G. A survey of the project AUTOMATH. In *Essays on Combinatory Logic, Lambda Calculus, and Formalism.* J. P. Seldin and J. R. Hindley, Eds., Academic Press, New York, 1980, 589–606.
16. DIJKSTRA, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
17. GORDON, M., MILNER, R., AND WADSWORTH, C. *Edinburgh LCF: A Mechanized Logic of Computation. Lecture Notes in Computer Science, 78,* Springer-Verlag, New York, 1979.
18. GRIES, D. *The Science of Programming.* Springer-Verlag, New York, 1982.
19. GUARD, J. R., OGLESBY, F. C., BENNETT, J. H., AND SETTLE, L. G. Semiautomated mathematics. *J. ACM 18* (1969), 49–62.
20. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12* (Oct. 1969), 576–580.

21. JUTTING, L. S. Checking Landau's "Grundlagen" in the AUTOMATH system. Ph.D. dissertation, Eindhoven Univ., *Math. Centre Tracts No. 83*. Math. Centre, Amsterdam, 1979.
22. KLEENE, S. C. On the interpretation of intuitionistic number theory. *JSL 10* (1945), 109–124.
23. KLEENE, S. C. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, N.J., 1952.
24. KRAFFT, D. B. AVID: A system for the interactive development of verifiable correct programs. Ph.D. dissertation, Cornell Univ., Aug. 1981.
25. LAKATOS, I. *Proofs and Refutations*. Cambridge University Press, Cambridge, 1976.
26. MANNA, Z., AND WALDINGER, R. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst. 2*, 1 (Jan. 1980), 90–121.
27. MARTIN-LÖF, P. Constructive mathematics and computer programming. In *6th International Congress for Logic, Method, and Philosophy of Science*, (Hannover, Aug. 1979).
28. NORDSTROM, B. Programming in constructive set theory: Some examples. In *Proceedings 1981 Conference on Functional Programming, Languages, and Computer Architecture*, (Portsmouth, 1981), 141–153.
29. PAULSON, L. Structural inductions in LCF. In *Proceedings International Symposium on Semantics of Data Types*. Springer-Verlag, New York, 1984.
30. POLYA, G. *How To Solve It*. Princeton University Press, Princeton, N.J., 1945.
31. PROOFROCK, J. A. PRL: Proof refinement logic programmer's manual (Lambda PRL VAX version). Computer Science Dept., Cornell Univ., 1983.
32. REYNOLDS, J. C. *The Craft of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1981.
33. SCOTT, D. Constructive validity. In *Symposium on Automatic Demonstration. Lecture Notes in Mathematics, 125*. Springer-Verlag, New York, 1970, 237–275.
34. SHOSTAK, R. E., SCHWARTZ, R., AND MELLIAR-SMITH, P. M. STP: A mechanized logic for specification and verification. In *6th Conference on Automated Deduction. Lecture Notes in Computer Science, 138*. Springer-Verlag, New York, 1982, 32–49.
35. SHOSTAK, R. E. A practical decision procedure for arithmetic with function symbols. *J. ACM 26* (Apr. 1979), 351–360.
36. STENLUND, S. *Combinators, Lambda-Terms, and Proof-Theory*. D. Reidel, Dordrecht, 1972.
37. TEITELBAUM, R., AND REPS, T. The Cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM 24*, 9 (Sept. 1981), 563–573.
38. WHITEHEAD, A. N., AND RUSSELL, B. *Principia Mathematica*. Vol. 1, Cambridge University Press, Cambridge, 1925.
39. WIRTH, N. *Systematic Programming: An Introduction*. Prentice-Hall, Englewood Cliffs, N.J., 1973.
40. WOS, L., OVERBEEK, R., EWING, L., AND BOYLE, J. *Automated Reasoning*. Prentice-Hall, Englewood Cliffs, N.J., 1984.

Received June 1983; revised June 1984; accepted July 1984