PROOFS AS PROGRAMS

Joseph L. Bates
Robert L. Constable

TR 82-530
February 1983

Department of Computer Science
Cornell University
Ithaca, NY 14853

This research was supported in part by National Science Foundation Grant MCS-81-04018

PROOFS AS PROGRAMS

Joseph L. Bates

Robert L. Constable

Cornell University

Ithaca, N.Y. 14853

Abstract

The significant intellectual cost of programming is for problem solving and explaining and not for coding. Yet, programming systems offer mechanical assistance exclusively with the coding process. Here we describe an implemented program development system, called PRL ("pearl"), that provides automated assistance with the hard part. The program and its explanation 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.

This research was supported in part by National Science Foundation Grant MCS-81-04018.

I. 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.† 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 between programming and algorithmic mathematics.†† 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, e.g. FORTRAN or Algol, etc. A good deal of effort might be invested in getting the particular piece of code to execute on a specific machine, e.g. typing, editing, submitting, etc. The task may even require mechanical assistance, e.g. diagnostic compilers, smart editors, etc. Nevertheless, in all but the most routine problems, the significant effort in programming is problem solving, i.e. 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

†This problem came to us from Jon Bently via David Gries. Jon encountered it while consulting.

††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

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, case analysis and the like. 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 $[a_1,a_2,...,a_n]$, we say that a subsequence of the form $[a_i,a_{i+1},...,a_{i+p}]$ is a *segment*, i.e. 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 $\sum_{j=i}^{i+p} a_j$, which is maximum among all segment sums.

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

$$M = \max(1 \leq k \leq q \leq n \colon \sum_{j=k}^{q} a_j)$$

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, i.e. 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 $\leq$ 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 *weakening* in Dijkstra [12] and Gries [15].

Notice that the formal specification of the problem has suggested the methods of 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.† So suppose it has been solved for sequences of length n, yielding sum M on the segment at i of length p. We present the induction hypothesis graphically in figure 1.

a_1		a_i	....	a_p		a_n

Figure 1

†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 [11] for a treatment of this rule in the style of this article.

a_1		a_i	....	a_p		a_n	a_{n+1}

Figure 2

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 can't simply compare M with $M + a_{n+1}$ because M may be the sum of a segment no longer contiguous with 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 form $[a_j,...,a_{n+1}]$. 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 include L_n unless $L_n + a_{n+1} \leq a_{n+1}$. If $a_{n+1} > L_n + a_{n+1}$ then 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≤k≤q≤n: f(k,q)) = max(1≤q≤n: max(1≤k≤q: f(k)q)))

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

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

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

Notice that $\max(1 \leq q \leq n\text{: } \max(1 \leq k \leq q\text{: } \sum_{j=k}^{q} a_j)) = M_n$

and $\max(1 \leq k \leq n+1\text{: } \sum_{j=k}^{n+1} a_j) = L_{n+1}$. Thus the second equation says that $M_{n+1} = \max\{M_n, L_{n+1}\}$.

In addition since we know

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

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

II. 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 attached to the code written in pidgin English. On the other hand, if 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. Moreover a proof introduces information according to specific needs.

The notion of a proof serves not only to organize information, but it can be used 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. 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, etc. 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, which 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 1970's 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 it, checking it, modifying it and using it in other unforseen ways. It becomes then 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 will convert the analysis of section I into a careful proof. Anticipating an interest in formalizing this proof, we will use the notation of symbolic logic to describe the problem. The connectives "and", "or", "implies" and "not" will be represented by &, $\vee$, $\Rightarrow$ and $\sim$ respectively, and the quantifiers "for all integers x", "for all lists A", "for some natural number y" will be represented as "*all* x:*int*", "*all* A:*list*", and "*some* y:*nat*" respectively. We use len(A) to denote the length of a list, and A(i) to denote the i-th element of a list provided $0 < i \leq \text{len}(A)$.

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

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

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

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 [2]). However, the core PRL theory to be described in section III does not include second order operations.

A proof which follows the first method of analysis of section I can be directly formalized in the core PRL language described later. This proof does not appeal to general properties of the maximum operation but deals instead entirely with integers and lists. Here is a presentation of the problem and its solution in that language.

> *all* n:*nat* . *all* A:*list* . *some* (M,L):*int* .
>
> *some* (a,b,s):*nat* . *all* (p,q):*nat* .
>
> $(1 \leq p \leq q \leq n \ \& \ len(A) = n) \Rightarrow M = \sum_{i=a}^{b} A(i) \ \& \ M \geq \sum_{j=p}^{q} A(j) \ \&$
>
> $L = \sum_{i=s}^{n} A(i) \ \& \ L \geq \sum_{j=p}^{n} A(j))$

If we prove this statement "constructively", then we will find the values M and L. So let us begin a proof and see how to keep it constructive.

Proof by induction on n, the length of the list.

Base Case: n=1

If len(A)=1, then the only segment is the entire list, M = L = a_1.

Induction Case:

Suppose that the result is true for all lists of length n. Then consider the statement of the theorem for n+ 1 and any list A. The length of A is either n+ 1 or not. If not then pick M and L to be anything. If len(A) = n+ 1, then look at the list A′ of length n obtained by removing the head element. This list A′ satisfies the induction hypothesis so there exist numbers M_n, L_n.

1. We must find L_{n+1}.

   Claim: $L_{n+1} = max(A(n+1), L_n + A(n+1))$

   We must show that

   (i) $L_{n+1} \geq \sum_{i=p}^{n+1} A(i)$ for any p and

   (ii) $L_{n+1} = \sum_{i=s}^{n+1} A(i)$ for some s.

   By definition $\sum_{i=p}^{n+1} A(i) = \sum_{i=p}^{n} A(i) + A(n+1)$ and we know

   $L_n \geq \sum_{i=p}^{n} A(i)$ hence for $p \leq n$ by simple arithmetic we know

$$L_n + A(n+1) \geq \sum_{i=p}^{n} A(i) + A(n+1)$$

For $p = n+1$, since $\sum_{i=p}^{n} A(i) = 0$ by definition of $\sum_{i=n+1}^{n}$ we know

$$A(n+1) \geq \sum_{i=p}^{n} A(i) + A(n+1)$$

Hence for all $p \leq n+1$

$$max(A(n+1), L_n + A(n+1)) \geq \sum_{i=p}^{n+1} A(i), \text{ showing (i).}$$

To show (ii) notice that if $L_{n+1} = A(n+1)$, then $s = n+1$, and if $L_{n+1} = L_n + A(n+1)$ then since $L_n = \sum_{i=r}^{n} A(i)$ we take $s = r$.

2. We must also find M_{n+1}

Claim: $M_{n+1} = max(M_n, L_{n+1})$. We must show

(j) $M_{n+1} \geq \sum_{i=p}^{q} A(i)$ for all p,q where $1 \leq p \leq q \leq n+1$.

and

(jj) $M_{n+1} = \sum_{i=c}^{d} A(i)$.

To show (j) we know that

$$M_n \geq \sum_{i=p}^{q} A(i) \text{ for all p,q where } 1 \leq p \leq q \leq n$$

The only new elements considered in (j) are of the form $\sum_{i=r}^{n+1} A(i)$ for all r where $1 \leq r \leq n+1$. But for these we know $L_{n+1} \geq \sum_{i=r}^{n+1} A(i)$. Thus $max(L_{n+1}, M_n) \geq \sum_{i=p}^{q} A(i)$ where $1 \leq p \leq q \leq n+1$. To show (jj) notice that if $M_{n+1} = M_n$, then $M_{n+1} = \sum_{i=a}^{b} A(i)$ otherwise $M_{n+1} = \sum_{i=s}^{n+1} A(i)$.

Qed

Qed

Intuitively it is plausible that this 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. len(A) = n+ 1 or not.

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:*nat* . R(x,y), will build a witness y for the assertion R(x,y). For instance a proof that *some* y:*nat* . (y>x) will give an expression for y such as (x+ 1) or (2x+ 1) etc. The particular value chosen depends on the proof used. A proof that *some* y *nat* . (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:*nat* R(x,y) has the following pattern.

all x:*nat* . *some* y:*nat* . R(x,y)

Proof (by induction on x).

Base: construct some value y_0 where $R(0,y_0)$

Induction: assume *some* y:*nat* . R(x,y).

Show *some* y:*nat* . R(x+ 1,y)

The proof builds a particular term t for y using

x+ 1 and the value y_x assumed to exist for

x. So t can be denoted $t(x+1,y_x)$.

Qed

Qed

The part of the proof building the value y is a recursive procedure of the form p(x) = *if* x=0 *then* y_0 *else* t(x,p(x-1)) *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*, [18].

A constructive formal system has the property that every proof step can be interpreted as a construction. This is explained in [17] and is applied to programming in [6,1]. So it in fact makes sense to *execute* constructive formal proofs. In the case of a proof of *all* x:*nat* . *some* y:*nat* . 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. So 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 like Algol. But it is possible to go in this direction within the context that regards proof as explanation 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 [10] and in [1] for example. But in the course of the work reported in [1], 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 [1,19]. It would also employ the technology of modern programming environments, especially the Cornell Program Synthesizer [24]. The work of Dean Krafft in building a synthesizer environment for PL/CV2, called AVID [19], 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.

III. 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 Martin-Löf [20] and the related work of Constable [7,8]. Some of our ideas are presented in [2]. Considerable effort was also spent designing a system with which to use this very general theory. The system will provide a modern environment for interactive proving and problem solving. It will be based on the notion of a library of results organized into books, chapters, sections, etc. The user will have help in generating material for publication in the library and will have a "smart" editor for viewing and modifying results in the library. There will also be means of invoking formal metareasoning to extend the system safely, building for example guaranteed proof tactics [8,14].

There are many difficult technical problems associated with building a system of this generality. These will be 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 [24] and AVID [19], a proof extractor based on Bates' thesis [1], and a metareasoning facility obtained by embedding PRL in Edinburgh LCF as an object theory [14]. 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 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, ... They include the unary function -, the infix binary operators: +, -, *, /, and various atomic functions: *mod, hd, tl,* and $\cdot$. The list constants are [ ], $[n_1,...,n_p]$ for n_i integers.

The function constants mod, hd, tl and $\cdot$ have these types:

mod: *int* × *int* → *int*
hd : *list* → *int*
tl : *list* → *list*
$\cdot$: *int* × *list* → *list*

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 ~A, A&B, A∨B, A ⇒ B for A and B formulas. The usual precedence holds among these connectives: ~, &, |, ⇒ and ⇒ is right associative. In addition, compound formulas include

all $x_1,...,x_n$: type . A
some $x_1,...,x_n$: type . A

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

An *environment* env is a list of variables and their types. A *goal* has the form

[env] Assm ⊢ 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 $f(p_1,...,p_n)$ 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 of Edinburgh LCF. 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. Let S_i denote sets of hypotheses. S', S'' are subsets of S.

$S \vdash A \;\&\; B$ *by andin*
  $S' \vdash A$
  $S'' \vdash B$

$S, A \;\&\; B \vdash C$ *by andel*
  $S, A, B \vdash C$

$S \vdash A \vee B$ *by orinl*
  $S' \vdash A$

$S \vdash A \vee B$ *by orinr*
  $S' \vdash B$

$S, A \vee B \vdash C$ *by orel*
  $S, A \vdash C$
  $S, B \vdash C$

$S \vdash A \Rightarrow B$ *by impin*
  $S, A \vdash B$

$S, A \Rightarrow B \vdash C$ *by impel*
  $S \vdash A$
  $S, A, B \vdash C$

$S \vdash \exists x{:}T.A$ *by existin*, t
  $S' \vdash A(t/x)$

[e] $S, \exists x{:}T.A \vdash C$ *by existel*
  [e,x:T] $S, A \vdash C$

[e] $S \vdash \forall x{:}T.A$ *by allin*
  [e,x:T] $S \vdash A(x)$

$S, \forall x{:}T.A \vdash C$ *by allel*,t
  $S, \forall x{:}T.A, A(t) \vdash C$

In addition we use a rule for combining subproofs called the cut or consequence rule.

$S \vdash G$ *by conseq* C
  $S \vdash C$
  $S, C \vdash G.$

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

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

(2) named functions extracted from proofs, with syntax: name *extract* theorem name

where the theorem must be of the form *all* $x_1...x_n$: type . *some* y: type . P. The function may be used in subsequent applications.

(3) 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 and the right hand side is any piece of text with interspersed parameter references. 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.

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

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

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

$$
\begin{array}{l}
\mathrm{x} < \mathrm{a} \Rightarrow \mathrm{f(x,\ldots)} = t_0 \\
\mathrm{x} = \mathrm{a} \Rightarrow \mathrm{f(x,\ldots)} = t_1 \\
\quad\quad \vdots \\
\mathrm{x} > \mathrm{b} \Rightarrow \mathrm{f(x,\ldots)} = t_n
\end{array}
$$

given that $t_1...t_{n-1}$ 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, and t_n may invoke f recursively with first argument x-c and arbitrary other arguments, for c a constant satisfying $1 \leq c \leq b - a + 1$.

The syntax for list definitions is

$$
\begin{array}{l}
\text{f(x:list,...):type} = \\
\quad 0 \Rightarrow t_0 \\
\quad\quad \vdots \\
\quad n \Rightarrow t_n \\
\quad x \Rightarrow t_{n+1}
\end{array}
$$

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

$$
\begin{array}{l}
\text{length } x = 0 \Rightarrow f(x,...) = t_0 \\
\quad\quad\quad \vdots \\
\text{length } x = n \Rightarrow f(x,...) = t_n \\
\text{length } x > n \Rightarrow f(x,...) = t_{n+1}
\end{array}
$$

given that $t_0,...,t_n$ do not invoke f, and t_{n+1} may call f recursively with first argument tl x, tl tl x, ... 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.

Examples

Here is a complete sample proof for the well known Euclidean division algorithm. Since the logic is not designed to display proofs on paper as they are developed, we can only approximate the kind of interactive proof generation that the system encourages. We do this by presenting the proof at various stages of development. To facilitate writing we abbreviate extensively. We suppress the display of unnecessary assumptions, as the system does; and we introduce a large number of simple definitions with mnemonic names, e.g. indhyp for "induction hypothesis" and goal for the quantifier free proposition expressing the theorem. Here are the definitions.

$$
\begin{array}{l}
\textit{def } \text{hyp}(n,a,b) = n \geq 0 \;\&\; a \geq 0 \;\&\; b > 0 \;\&\; a - b < n, \\
\quad\;\; \text{goal}(a,b,q,r) = (a = b*q + r \& 0 \leq r < b), \\
\quad\;\; G(n,a,b) = \textit{some } q,r{:}\textit{int} \,.\, (\text{hyp}(n,a,b) \Rightarrow \text{goal}(a,b,q,r)), \\
\quad\;\; \text{indhyp}(n) = \textit{all } a,b{:}\textit{int} \,.\, G(n,a,b).
\end{array}
$$

The main goal is:

$\vdash$ *all* n:*int* . *all* a,b:*int* . *some* q,r:*int* . (hyp(n,a,b) $\Rightarrow$ goal(a,b,q,r))

With full abbreviation the goal is simply

$\vdash$ *all* n:*int* . indhyp(n).

The essence of this algorithm is induction. In the typical setting for this algorithm, there are so few primitives that we are required to build even the divide function. Although in principle we need induction on only the nonnegative integers, the system requires us to consider all integers. So we call for a proof by induction from 0 as the base case. After entering the command *induction,0* the response is a list of subgoals as shown below.

induction,0

1. $[n{:}\mathit{int}]$ $n<0$, indhyp(n+ 1) $\vdash$ indhyp(n)
2. $[n{:}\mathit{int}]$ $n=0$, $\vdash$ indhyp(0)
3. $[n{:}\mathit{int}]$ $n>0$, indhyp(n-1) $\vdash$ indhyp(n)

In the first case, since $n<0$ and the hypothesis includes the assumption that $n\geq 0$, it is trivial to achieve the goal. We first indicate that we want arbitrary integers a and b in order to introduce the all quantifier. Then we arbitrarily choose integers for q,r because the choice is irrelevant; we take 0,0. Here is the complete refinement of the first subgoal. The command is shown on the left and the system's response is beneath it.

allin

$[n{:}\mathit{int},a,b{:}\mathit{int}]$ $\vdash$ G(n,a,b)

existin,0,0

$[n{:}\mathit{int},a,b{:}\mathit{int},q,r{:}\mathit{int}]$ $n<0$, $q=0$, $r=0$ $\vdash$ hyp(n,a,b) $\Rightarrow$ goal(a,b,q,r)

impin

$n<0$, hyp(n,a,b) $\vdash$ goal(a,b,q,r)

false

$n<0$, hyp(n,a,b) $\vdash$ *false*

by arithmetic ($n<0$, $n\geq 0$)

Next we attack the second main subgoal. This is the basis case. Again the first step is to consider arbitrary a,b. So we direct the system to perform all-introduction.

allin

$[n{:}\mathit{int},a,b{:}\mathit{int}]$ $\vdash$ G(n,a,b).

We know that when n=0, a-b<0, so a<b, and we can take q=0 and r=a since a=b*0+ a and 0≤a<b. We thus direct the system to use the values 0 and a for q and r respectively.

existin 0,a

[n:*int*,a,b:*int*,q,r:*int*] n=0, q=0, r=a ⊢ hyp(n,a,b) ⇒ goal(a,b,q,r)

The standard way to prove an implication is to introduce it. So we direct an implication introduction.

impin

n=0, q=0, r=a, hyp(n,a,b) ⊢ goal(a,b,q,r)

Now we use arithmetic on a-b<0 to finish this branch.

arithmetic (a<b).

Next we consider the only remaining subgoal.

3. [n:*int*] n>0, indhyp(n-1) ⊢ indhyp(n).

As in cases 1 and 2 we perform an all-introduction. But now in order to choose values for q and r we must express them in terms of the values q′ and r′ known to exist for the smaller problem of dividing (a-b) by b. To obtain these values we must invoke the induction hypothesis on (a-b) and b. This is done by the all-elimination rule applied to indhyp(n-1) with (a-b) for a and b for b. Let us examine just this segment of the proof.

allin

n>0, indhyp(n-1) ⊢ G(n,a,b)

allel, indhyp(n-1), (a-b), b

n>0, indhyp(n-1), G(n-1,(a-b),b) ⊢ G(n,a,b)

At this stage we must know whether a-b<0 or a-b≥0 in order to know whether to take q=0 and r=a or q=q′ + 1 or r=r′. So a proof by cases is conducted. Here is its structure.

arithmetic, (a-b)<0 ∨ (a-b)≥0

(a-b)<0 ∨ (a-b)≥0 ⊢ G(n,a,b)

orel

1. (a-b)<0 $\vdash$ G(n,a,b)

2. (a-b)$\geq$0 $\vdash$ G(n,a,b)

The first subgoal is proved as in the basis case. For the second subgoal before we can define q and r we need to obtain the values of q,r in *some* q,r:*int* . (hyp(n-1,(a-b),b) $\Rightarrow$ goal((a-b),b,q,r)). We choose q$'$,r$'$ as values, and then in order to use the information in goal((a-b),b,q$'$,r$'$) we prove that hyp(n-1,(a-b),b) holds. The details of these steps can be found in the complete proof which follows. For esthetic reasons we write the commands to the system at the end of the goal to which they are applied. Thus to generate the above subgoals we write

(a-b)<0 $\vee$ (a-b)$\geq$0 $\vdash$ G(n,a,b) *by orel*

$\vdash$ *all* n:*int* . indhyp(n) *by induction,*

(negative case)

[n:*int*] n<0, indhyp(n+ 1) $\vdash$ indhyp(n) *by allin*

Let e = n:*int*, a:*int*, b:*int*

[e] n<0, indhyp(n+ 1) $\vdash$ G(n,a,b) *by existin*, 0, 0

$\vdash$ hyp(n,a,b) $\Rightarrow$ goal(a,b,0,0) *by impin*

[e] n<0, hyp(n,a,b) $\vdash$ goal(a,b,0,0) *by false*

[e] n<0, hyp(n,a,b) $\vdash$ *false by arith* (n<0, n$\geq$0)

(base case)

[n:*int*] n=0 $\vdash$ indhyp(n) *by allin*

[e] n=0 $\vdash$ G(n,a,b) *by existin*, 0,a

[e] n=0 $\vdash$ hyp(n,a,b) $\Rightarrow$ goal(a,b,0,a) *by impin*

[e] n=0, hyp(n,a,b) $\vdash$ goal(a,b,0,a) *by arith* (a-b<0).

(positive case)

[n:*int*] n>0, indhyp(n-1) $\vdash$ indhyp(n) *by allin*

  [e] n>0, indhyp(n-1) $\vdash$ G(n,a,b) *by allel*, indhyp(n-1), (a-b),b

    [e] n>0, G(n-1,(a-b),b) $\vdash$ G(n,a,b) *by arith on left*

      [e] (a-b)<0 $\vee$ (a-b)$\geq$0 $\vdash$ G(n,a,b) *by orel*

        [e] a-b<0 $\vdash$ G(n,a,b) *by existin*, 0, a

          [e] a-b<0 $\vdash$ hyp(n,a,b) $\Rightarrow$ goal(a,b,0,a) *by impin*

            [e] a-b<0 $\vdash$ goal(a,b,0,a) *by arith* (a<b)

        [e] a-b$\geq$0, G(n-1,a-b,b) $\vdash$ G(n,a,b) *by existel*

          [e,q$'$:*int*, r$'$:*int*] n>0, hyp(n-1,a-b,b) $\Rightarrow$ goal(a-b,b,q$'$,r$'$) $\vdash$ G(n,a,b) *by existin*, q$'$ + 1, r$'$

(Let e$'$ = (e,q$'$:*int*, r$'$:*int*) and suppress most assumptions.)

            [e$'$] $\vdash$ hyp(n,a,b) $\Rightarrow$ goal(a,b,q$'$ + 1,r$'$) *by impin*

              [e$'$] hyp(n-1,a-b,b) $\Rightarrow$ goal(a,b,q$'$,r$'$), hyp(n,a,b) $\vdash$ goal(a,b,q$'$ + 1,r$'$) *by impel*

                [e$'$] n>0, hyp(n,a,b) $\vdash$ hyp(n-1,a-b,b) *by andin*

                  [e] n>0, hyp(n,a,b) $\vdash$ n-1 $\geq$ 0 *by arith*

                  [e$'$] n>0, a-b<n,b>0 $\vdash$ (a-b)-b<n-1 *by arith*

                  [e$'$] a-b$\geq$0 $\vdash$ (a-b)$\geq$0

                  [e$'$] hyp(n,a,b) $\vdash$ b>0 *given*

                [e$'$] hyp(n,a,b), goal(a-b,b,q$'$,r$'$) $\vdash$ goal(a,b,q$'$ + 1,r$'$) *by arith*

(this is just (a-b) = b*q$'$ + r$'$ & 0$\leq$r$'$ <b$\vdash$ a = b*(q$'$ + 1) + r$'$ & 0$\leq$r$'$ <b.)

Larger Examples

One principle that guides the organization of larger pieces of technical text than the above example is that it be decomposed into pieces that fit on a single page and can be understood more or less in isolation. In a programming language like Lisp this is accomplished by decomposing a large program into a sequence of small functions. In proofs it is accomplished by writing many small lemmas. There is however an overhead in decomposing proofs into lemmas because the statement of the result must be repeated. In procedural programming languages like Algol there is also an overhead associated with decomposition into procedures because declarations and comments must be repeated.

(There is usually some execution inefficiency as well.)

The refinement style presentation of proofs adopted in PRL supports this principle of modular decomposition without substantial overhead. At any point in the proof a single goal is visible and the list of assumptions permitted in meeting the goal can be made explicit. The system allows the user a display of "one page worth" of the proof tree. This display can highlight the relevant assumptions and thus serves some of the purposes of lemmas and subprograms. These features were illustrated above and also in the following fragment of a PRL formalization of the max segment sum example of sections I and II. First we express relevant definitions.

define: B(A,n,L,M,a,b,s,p,q) =

$$(1 \le p \le q \le n \;\&\; n = \mathrm{len}(A)) \Rightarrow$$

$$(M = \sum_{i=a}^{b} A(i) \;\&\; M \ge \sum_{j=p}^{q} A(j)) \;\&\; (L = \sum_{i=s}^{n} A(i) \;\&\; L \ge \sum_{j=p}^{n} A(j))$$

define: G(A,n,L,M) =

some(a,b,s):*nat* . *all*(p,q):*nat* . B(A,n,L,M,a,b,s,p,q)

At the top level the problem specification in PRL is written

⊢ *all* n:*nat* . *all* A:*list* . *some*(L,M):*int* . G(A,n,L,M)

The user then calls for a proof by induction on n with base 1 and the system responds with a list of subgoals as follows.

[n:*nat*] *all* A:*list* . *some*(L,M):*int* . G(A,n,L,M), n<1

⊢ *all* A:*list* . *some*(L,M):*int* . G(A,n-1,L,M)

[n:*nat*] n=1 ⊢ *all* A:*list* . *some*(L,M):*int* . G(A,n,L,M)

[n:*nat*] *all* A:*list* . *some*(L,M):*int* . G(A,n,L,M), n>1

⊢ *all* A:*list* . *some*(L,M):*int* . G(A,n+1,L,M)

The user is free to prove the subgoals in any order. It will be easy to prove the result for n<1 because the hypothesis $1 \le p \le q \le n$ is vacuous. In the case n=1, the value A(1) suffices for L and M.

Consider what can be done in the last case. The all statement is proved by *all introduction* (see the list of rules). The user invokes the rule *allin* and the following subgoal is generated:

[n:*nat*, A:*list*] *all* A:*list* . *some*(L,M):*int* . G(A,n,L,M), n>1

$\vdash$ *some*(L,M):*int* . G(A,n+ 1,L,M)

At this point it is difficult to work on the goal because to get beyond the existential quantifier, *some*(L,M):*int*, expressions for L and M must be chosen for use with the rule *existel*. But such expressions may not be known at this point. To discover them the user can extract more information from the hypotheses. This too requires knowledge, but the pattern of inductive proof suggests what to do. The induction hypothesis supplies information about smaller lists. The list tl(A) is such a list, so the *allel*, tl(A) rule is used to produce the following subgoal (only the relevant hypotheses are written on the left from now on, and we use "goal" to denote *some*(L,M):*int* . G(A,n+ 1,L,M)).

[n:*nat*] *some*(L,M):*int* . G(tl(A),n,L,M) $\vdash$ goal

Now it is possible to choose values for L,M by first getting expressions for L,M on the list tl(A) of length n. We do this by existential elimination to produce the following subgoal.

[n:*nat*, A:*list*,(L_o,M_o):*int*] G(tl(A),n,L_o,M_o) $\vdash$ goal

Now it is possible to choose values for the indexes of Σ as follows.

[n:*nat*, A:*list*,(L_o,M_o):*int*, (a,b,s):*nat*, (p,q):*nat*]

B(tl(A),n,L_o,M_o,a,b,s,p,q) $\vdash$ goal.

At this point we need insight to define L and M, but having determined that the proper choice is L $= max(L_o,L_o + A(n+1))$ and M $= max(L,M_o)$, then the rule of consequence bridges between the goal and the properties easily proved of L and M. For example, let

$$C(A,n,L_o,M_o) = all(p,q){:}nat\ .\ (\max(A(n+1),L_o + A(n+1)) \geq \sum_{j=p}^{n+1} A(j)$$

$$\&\ \max(M_o,\ \max(A(n+1),L_o + A(n+1))) \geq \sum_{i=p}^{q} A(i))$$

Then the rule of consequence on C produces the subgoals

$$H, C \vdash \text{goal}$$

$$H \vdash C$$

where H is the list of hypotheses at the point that consequences is used.

The remainder of the proof is a routine translation of the argument in section II. However, the logic is designed with the highly interactive system in mind, therefore a complete rendition of the formal proof on paper would not present a realistic image of proof development in PRL. This segment in fact illustrates the kind of local reasoning typical of proof development. In a more lengthy report we will discuss more fully the role of the system in efficiently generating proofs.

Acknowledgements

We would like to thank our colleagues on the PRL project for their advise and suggestions. We expecially acknowledge the detailed criticisms of Fred Schneider and David Gries. Not only that, but David also supplied us Jon Bentley's problem on segment sums. We also appreciate the efforts of Donette Isenbarger in preparing the manuscript.

REFERENCES

[1]: Bates, J.L., *A Logic for Correct Program Development*, Ph.D. Thesis, Department of Computer Science, Cornell University, 1979.

[2]: Bates J. and R.L. Constable, "Definition of Micro-PRL", Technical Report TR 82-492, Computer Science Department, Cornell University, October 1981.

[3]: Bishop, E., *Foundations of Constructive Analysis*, McGraw Hill, NY, 1967, 370 pp.

[4]: Bishop, E., "Mathematics as a Numerical Language", *Intuitionism and Proof Theory*, ed. by Myhill, J., et al., North-Holland, Amsterdam, 1970, 53-71.

[5]: Boyer, R.S. and J.S. Moore, *A Computational Logic*, Academic Press, NY, 1979, 397 pp.

[6]: Constable, Robert L., "Constructive Mathematics and Automatic Program Writers", *Proc. of IFIP Congress*, Ljubljana, 1971, 229-233.

[7]: Constable, Robert L., "Programs and Types", *Proc. of 21st Annual Symposium on Foundations of Computer Science*, IEEE, NY, 1980, 118-128.

[8]: Constable, Robert L., "Intensional Analysis of Functions and Types", University of Edinburgh, Dept. of Computer Science Internal Report CSR-118-82, June 1982.

[9]: Constable, Robert L. and D.R. Zlatin, "The Type Theory of PL/CV3", IBM Logic of Programs Conference, *Lecture Notes in Computer Science, Vol.* 131, Springer-Verlag, NY, 1982, 72-93.

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

[11]: Constable, Robert L., "Programs As Proofs", Department of Computer Science, Technical Report TR 82-532, Cornell University, 1982. (To appear in *Information Processing Letters*, 1983.)

[12]: deBruijn, N.G., "A Survey of the Project AUTOMATH", *Essays on Combinatory Logic, Lambda Calculus and Formalism*, (eds. J.P. Seldin and J.R. Hindley), Academic Press, NY, 1980, 589-606.

[13]: Dijkstra, Edsger W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976, 217 pp.

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

[15]: Gries, David, *The Science of Programming*, Springer-Verlag, 1982.

[16]: Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", *Comm. ACM, 12*, Oct. 1969, 576-580.

[17]: Kleene, S.C., "On the Interpretation of Intuitionistic Number Theory", *JSL, 10*, 1945, 109-124.

[18]: Kleene, S.C., *Introduction to Metamathematics*, D. Van Nostrand, Princeton, 1952, 550 pp.

[19]: Krafft, Dean B., "AVID: A System for the Interactive Development of Verifiable Correct Programs", Ph.D. Thesis, Cornell University, Ithaca, NY, August 1981.

[20]: Martin-Lof, Per, "Constructive Mathematics and Computer Programming", *6th International Congress for Logic, Method and Phil. of Science*, Hannover, August, 1979.

[21]: Nordstrom, B., "Programming in Constructive Set Theory: Some Examples", *Proc. 1981 Conf. on Functional Prog. Lang. and Computer Archi*, Portsmouth, 1981, 141-153.

[22]: Scott, Dana, "Constructive Validity", *Symposium on Automatic Demonstration, Lecture Notes in Mathematics*, 125, Springer-Verlag, 1970, 237-275.

[23]: Stenlund, S., *Combinators, Lambda-terms, and Proof-Theory*, D. Reidel, Dordrecht, 1972, 183 pp.

[24]: Teitelbaum, R. and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Comm. ACM, 24, 9*, September 1981, 563-573.