PROGRAMS AS PROOFS

Robert L. Constable

November 1982

TR 82-532

Department of Computer Science
Cornell University
Ithaca, New York 14853

PROGRAMS AS PROOFS

Robert L. Constable
Cornell University

ABSTRACT

Programs are like constructive proofs of their specifications. This analogy is a precise equivalence for certain classes of programs. The connection between formal logic and programs is a foundation for programming methodology superior to that usually adopted. Moreover this equivalence suggests programming languages which are far richer than all others currently in use. These claims are established in this paper introducing parts of the PL/CV programming logics as a source of precision and examples.

KEY WORDS AND PHRASES

Algorithmic logic, automated logic, axiomatic semantics, constructive mathematics, program correctness, PL/CV, programming logic, programming methodology, realizability, while rule.

This work was supported in part by National Science Foundation Grant MCS-81-04018 and NSF-80-03349.

OUTLINE

1. Constructive Proofs

   1.1 what logicians have known

   1.2 compiling formal mathematics

   1.3 algorithms in mathematics

2. Algorithmic Proofs

   2.1 constructive logic

   2.2 proofs with conditionals

   2.3 proofs with assignments

   2.4 iteration and induction

   2.5 executing algorithmic proofs

3. Type Theoretic Proofs

   3.1 types

   3.2 propositions as types

   3.3 new programming languages

1. CONSTRUCTIVE PROOFS

1.1 what logicians have known

Ever since the 1940's work of Kleene [K45], logicians have known that various kinds of constructive proof could be "compiled" into executable code. For instance a proof of an assertion of the form "for all integers n there is an integer y such that R(n,y)" can be compiled into a computable function f such that R(n,f(n)). Results of this variety are known for many constructive formal systems [T73]. The essential ideas behind them can be grasped from an example. We choose a very simple benchmark common in the literature, the Euclidean division algorithm.

The theorem we want to prove is "for all non-negative integers a and b we can find non-negative integers q and r such that if b is positive, then a=b*q+ r and $0 \leq r < b$." To render this and other theorems more succintly we will adopt a conventional symbolic notation using &, $\vee$, $\neg$, $\Rightarrow$ for "and", "or", "not", and "implies" respectively and using $\forall(x_1,\ldots,x_n):A$ and $\exists(x_1,\ldots,x_n):A$ as the typed quantifiers "for all $x_1,\ldots,x_n$ of type A" and "we can find $x_1,\ldots,x_n$ of type A" respectively. As types we take N for the non-negative integers and Z for the integers. We use * to denote multiplication. The theorem now takes the following form.

Theorem (Euclidean Algorithm): $\forall(a,b):N.\ \exists(q,r):N.$
$(b>0 \Rightarrow a=b*q+r\ \&\ 0\leq r<b)$

This theorem is typically proved as a special case of the following lemma.

Lemma: $\forall n:N\ .\ \forall(a,b):Z.$
$(a-b<n \Rightarrow \exists(q,r):N \cdot (a\geq 0\ \&\ b>0 \Rightarrow a=b*q+r\ \&\ 0\leq r<b))$

The lemma can be proved by induction on n, and the theorem follows by taking n to be a non-negative integer larger that a-b, say $max(0,\ a-b)+1$. The induction principle we have in mind can be stated as:

proof
for arbitrary n:N
assume P(n)
.
.
.
P(n+ 1)

$$\frac{P(0) \quad qed}{\forall n{:}N \;.\; P(n)}$$

This means that if we can prove P(0), and if we can prove P(n+ 1) from the assumption that P(n), then P is true for all non-negative integers.

Here is an informal proof of the lemma.

Proof by induction:
Base case:

a<b so take q=0, r=a, then
clearly by arithmetic a=b*0+ a and $0 \leq a < b$.

Induction case for arbitray n:N:

1. Assume the result for any x-y<n, in order to show that for all (a,b):Z where a-b<n+ 1 there exist appropriate q and r.

2. In order to apply the induction hypothesis we need inputs x,y such that $x \geq 0$, y>0 and $x-y<n$. If we take (a-b) for x and b for y, then because b>0 and $(a-b)<n+1$ we know (a-b)-b<n by arithmetic.

3. In order to apply the induction hypothesis we need to know also that $(a-b) \geq 0$. We only have this if $b \leq a$, but when a<b we can proceed as in the base case. Here are the details.

(a-b)<0 or (a-b)≥0 by arithmetic.
We prove the theorem in each case.
case (a-b)<0;
    then a<b so take q=0, r=a.
case (a-b)≥0;
    then the induction hypothesis holds
      with (a-b) for x and b for y,
    so there are q', r' such that
    $(a-b) = b*q' + r'$ & $0 \leq r' < b$.
    $a = b*(q' + 1) + r'$ by arithmetic.
    Take $q = q' + 1$ and $r = r'$ to finish
      the proof in this case.
Qed

We can see the computational content of the proof rather clearly in terms of recursive procedures. When a-b<0, then the computation terminates by producing q=0 and r=a. If a-b≥0, then it is possible to "call the lemma recursively" using (a-b) and b. These are smaller parameters, so we know that the sequence of "calls to the lemma" is finite. After the lemma is called, the result is q', r'. From these values line 5 of the proof shows how to compute the final values. The proof has a structure similar to the following Algol-like recursive procedure.

```
lem: procedure   (n, a, b, q, r)
       declare   n:N  /* size of the inputs */;
                 (a, b):Z  /* dividend and divisor inputs */;
                 (q, r):N  /* quotient and remainder outputs */;
                 (q', r'):N  /* auxillary variables for q and r */;
       assume    a-b<n, a≥0, b>0;
       attain    a=b*q+ r & 0≤r<b;

       if n=0    then  (q:=0, r:=a)
                 else if (a-b)<0
                 then (q:=0, r:=a)
                 else lem(n-1, a-b, b, q', r')
                        /* (a-b) = b*q' + r' & 0≤r' < b */;
                        (q:=q' + 1, r:=r')
                 fi
       fi
end lem
```

Notice that the procedure will execute in cases such as n=3, a=25, b=3 for which the input assumptions cannot be satisfied at each call of lem. In such cases we simply cannot guarantee the program's behavior. The program will be correct for the division algorithm whenever n is greater than or equal to the quotient. (In many programming logics, including PL/CV, n would be part of the termination predicate and as such not an independent parameter, but the point of this procedure

is to illustrate the computational structure of the proof, not to stand on its own.)

We can at this point simply regard the assume and attain statements as comments about the behavior of the procedure.

One can see from this example the correspondence between an appeal to the induction hypothesis and a recursive call of the procedure. Likewise the correspondence between assignment, q:=0, and the choice of values for existentially quantified variables, "let q be 0", is clear.

1.2 compiling formal mathematics

The correspondences suggested by the above example can be made general, say to cover all proofs in a specific constructive theory as Kleene proved for Intuitionistic number theory [K45]. There are a number of subtle points that complicate the general result, for example translating nested implications of the form $A \Rightarrow (B \Rightarrow (C \Rightarrow D))$ involves higher order functions.

This correspondence suggests that the language of formal constructive proofs might be useful as a programming language. This idea was proposed by Constable [C71] and has been studied extensively by [MM73]. Recently J. Bates [B79] and C. Goad [G80] have taken further steps in this direction.

There are obvious advantages to using proofs as programs, namely one can be quite sure that they are "correct programs". A constructive proof by its nature must prove its conclusion. In programming terminology the conclusion is the problem specification. In this context the task of teaching certain aspects of programming is very similar to teaching methods of problem solving and proof. It is obvious that the methodological investigations of G. Polya [P48] and others are applicable as written. It is also obvious that certain techniques of formal logic are applicable, though many computer scientists have only recently realized this [D76, G81].

There are also obvious obstacles to using proofs as programs. In the first place for a large class of concurrent programs the correspondence is unclear. In the same vein for a large class of computational problems there is no adequate formalism, e.g., various real time control problems, various A.I. programs from chemical synthesis to expert systems. Moreover, even when we do possess a formal-

ism and a precise correspondence between proof and program, we still do not know how to compile these proofs into *efficient* code. This problem area is the subject of investigation at Cornell [B79, BC81] and Stanford [G80].

1.3 algorithms in mathematics

In some ways it may seem strange to treat proofs as algorithms especially since algorithms clearly have a place in mathematics which seems at first quite different from that of proofs. Algorithms are used to define functions. We would expect the "Euclidean division algorithm" to be a function d taking a and b to q and r, i.e., $d{:}N \times N^+ \rightarrow N \times N$ *where* $N^+ = \{1,2,3,...\}$. It would be customary to prove that the algorithm performed correctly, e.g., if d(a,b) = <q,r> and we write <q,r>.1=q, <q,r>.2=r, then correctness means

$$a = b*d(a,b).1 + d(a,b).2 \ \& \ 0 \leq d(a,b).2 < r.$$

This is the viewpoint usually adopted in studies of program correctness [M74, dB80] or programming methodology [D76, AA79, G81]. In the case of procedural programs, people have in mind a function mapping states to states in such a way that if the *precondition* (a relation on states) is true, then the resulting function values satisfy the *post condition.*

We will see shortly that it makes sense to treat constructive proofs computationally because the meaning of a constructive proof is computational. We will develop these ideas in the next section.

2. ALGORITHMIC PROOFS

2.1 constructive logic

To constructively prove an assertion of the form $A \Rightarrow B$ we must exhibit a method of transforming a proof of A into a proof of B. The direct way to do this is to assume that we are given a proof of A, say a, and try to build a proof of B from it. The pattern of such a construction in ordinary logic is

proof
  assume A
    .
    .
    .
    B
qed

To constructively prove $A \vee B$ we must prove either A or B, and to prove A & B we must prove both A and B.

The canonical way to use $A \vee B$ in a proof of C is to conduct a case analysis, showing that if A holds then C does and if B holds then C does. The usual pattern for this is

C *by cases*, $A \vee B$
proof
  case A
    .
    .
    .
    C

  case B
    .
    .
    .
    C
qed

Here is a proof using these rules

$(A \Rightarrow C \& B) \Rightarrow (A \vee B \Rightarrow B)$
proof
  assume $A \Rightarrow C \& B$;
    $A \vee B \Rightarrow B$;
    proof
      assume $A \vee B$;
      B *by cases*, $A \vee B$,
      proof
        case A;
          C & B by applying $A \Rightarrow C \ \& \ B$ to A;
          B
        case B
      qed
    qed
qed

2.2 proofs with conditionals

This conventional proof style can easily be augmented by more familiar algorithmic constructs such as the conditional, *if* bexp *then* C *else* D *fi*. This is essentially a case analysis on the assertion

bexp = *true* ∨ bexp = *false*. It is used only when we know that the Boolean expression, bexp, evaluates to either *true* or *false*.

Boolean expressions are not the same as assertions. They are expressions which denote one of the two Boolean values, *true* and *false*. We use the following operations on Booleans.

a	b	a*b	a+ b	¬a
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

The atomic Boolean expressions that we consider are x=y for x and y of the same type and for = a decidable operation. Over the integers we can prove assertions as follows:

$x \neq 0 \Rightarrow x*x>0$
proof
  assume $x \neq 0$
  if x<0 *then* x*x>0 *by arithmetic*
    else x>0 *by arithmetic*, $x \geq 0$, $x \neq 0$;
      x*x>0 *by arithmetic*
  fi
qed

An equivalent proof without the conditional rule is

$x \neq 0 \Rightarrow x*x>0$
proof
  assume $x \neq 0$;
  x<0 ∨ $x \geq 0$ *by arithmetic*;
  x*x>0 *by cases* x<0 ∨ $x \geq 0$,
  proof
    case x<0;
      x*x>0 *by arithmetic*
    case $x \geq 0$;
      x>0 *by arithmetic* $x \geq 0$, $x \neq 0$;
      x*x>0 *by arithmetic*
  qed
qed

A conditional is not only a special form of a "by cases" proof block, but it can also be used to write new assertions such as

if bexp *then* A *else* B *fi*

which is equivalent to the assertion

$$(\text{bexp} \Rightarrow \text{A}) \;\&\; (\neg\, \text{bexp} \Rightarrow \text{B})$$

In the case of programming languages like Algol 60 or 68, the conditional is used in both roles, but in a language like PL/I without conditional expressions, it is used only to augment the proof constructs.

2.3 proofs with assignments

A common feature of informal proofs are definitions of the form "let q=0, r=a". Assignment statements such as q:=0, r:=a clearly resemble these definitions. Indeed if assignments are used so that a variable is never *redefined*, then they can be treated exactly as definitions and their resulting equalities, i.e., from "let q=0" we conclude q=0.

When variables are redefined, as in x:=x+ 1, then clearly we cannot conclude the equality statement x=x+ 1, but moreover the whole idea of a variable and its use in a proof must be re-examined. For example even after the assignment x:=0 there are new issues to face if x is being *redefined* because statements involving x may change their meaning; consider this proof segment:

a>0; b>0;
$b^2>0$ *by arithmetic*, b>0;
b:=0;

$a*b^2>0$ *by arithmetic*, a>0, $b^2>0$.

The assignment b:=0 clearly invalidates the claim that $b^2>0$.

Although the introduction of assignment statements which redefine variables complicates the concept of a proof somewhat, it is possible to write a reasonably elegant set of proof rules that includes them as was done by Constable and O'Donnell [CO78]. The concept of assignment has clearly proved useful in the early development of efficient high level languages such as FORTRAN and Algol; it has even crept into languages with a basically applicative structure such as Interlisp, Franzlisp, ML, etc. But whether or not the assignment concept will remain in high level languages, it is now known from extensive work on programming methodology how to treat it rigorously, and it is known from [CO78] how to build it into a formal logical system without increasing the conceptual overhead for those parts of the system which do not use it.

2.4 iteration and induction

Typically the proof rules for iterative programming constructs, such as the while loop, are shown to be sound using mathematical induction. For example the rule of Hoare [H69]

$$\frac{\{P\&b\}\ S\ \{P\},\ P\&\neg b \Rightarrow Q}{\{P\}\ \textit{while}\ b\ \textit{do}\ S\ \textit{od}\ \{Q\}}$$

is justified by induction on the number of executions of the loop body S.

If the while rule is written in the PL/CV style [CO78, CJE82] as

P;
while b *do*
  assume P
  S
  P
od

P & ¬ b

then it can be seen as another form of induction principle. In fact comparing side by side the induction rule of 1.1 and this while rule reveals the similarity graphically.

PEANO INDUCTION	WHILE INDUCTION
proof *for arbitrary* n *assume* Q(n) . . . Q(n+ 1) Q(0) *qed*	Q; *while* b *do* *assume* Q . . . Q *od*
∀ n:N.Q(n)	$Q\ \&\ \neg\ b$

The base case for Peano induction is the proof of Q(0) whereas for while induction it is the proof of Q prior to the loop statement.

While induction is useful only in the presence of assignment statements which are used to redefine at least one of the variables occurring in the Boolean expression. But regardless of this dynamic interpretation of the loop, it can also be seen as another induction principle offering another way to prove an assertion such as Q & ¬ b.[+] Given the computational interpretation however, one can see

+ In PL/CV one can actually prove arbitrary assertions this way. See [CJE82] for the general rule.

while induction as a proof method which is in some cases *more efficient* than Peano induction. Here is a proof of the Euclidean algorthm using while induction and the assignment rule.

```
divide: procedure (a, b, q, r)
        declare (a, b):Z;
        declare (q, r):N;

        assume a≥0, b>0;
        attain a=b*q+ r, 0≤r<b;

        a=b*0+ a by arith;
        r:=a, q:=0;
        ∃i:N.(i≥0 ∧ r≤i) by intro, r;
        while ((r-b) ≥ 0) do
            assume a=b*q+ r, r≥0;
            arbitrary i:N where r≤i,
            ¬ (r≤0) by arith, r-b≥0, +, b>0;
            a=b*(q+ 1) + (r-b) by arith, a=b*q+ r;
            r-b ≤ i-1
            r:=r-b, q=q+ 1
        od
        r<b by arith, ¬(r-b≥0);
        a=b*q+ r, 0≤r<b
        return
    end
```

2.5 executing algorithmic proofs

One of the principal advantages of a proof system such as PL/CV which includes algorithmic constructs such as conditionals, assignments and iterative induction is that we already know how to effeciently execute a large class of the proofs. For example we recognize the proof at the end of (2.4) as the Euclidean division algorithm. Thus PL/CV is an example of a system of formal constructive mathematics whose proofs can be very efficiently executed, as efficiently as those in any high level programming language.

3. TYPE THEORETIC PROOFS

3.1 types

In addition to the base types we have mentioned, N, Z, we have been implicitly interested in all functions from one type to another. Given types $A_1,...,A_n$ and B, let us write the type of all functions from $A_1,...,A_n$ into B as $proc(A_1,...,A_n)B$. We are also interested in n-tuples (or records) and we write the type of n tuples $<a_1,...,a_n>$ with a_i from A_i as $prod(A_1,...,A_n)$. Both *proc*

and *prod* are similar to the Algol-68 types *proc* and *struct* respectively. We will also allow an Algol-like union, denoted *union*$(A_1,...,A_n)$. In this notation the divide example of 1. has type *proc*(N,Z,Z)*prod*(N,N).

When we examine the divide procedure as it is used in a constructive programming logic such as PL/CV, it makes sense to say that in addition to its numerical parameters n, a and b it has a parameter which is a proof of the input assumptions $a \geq 0,\ b > 0$. Such a proof is needed in a programming logic because without satisfying the input assumptions, the procedure cannot be invoked. The only way to satisfy these assumptions in the logic is to provide a formal proof. This notion of proof is very concrete. So it makes sense to think of the required formal proof as an input.

We could unify the notation and conception of the logic considerably if we allowed assertions to be types whose members were their proofs. Then we could say that the inputs to the divide procedure are of types N, Z, Z and $a \geq 0$ & $b > 0$ respectively. The output could also include a proof of the output specification (variously called the post condition or the attain statement). Again in a programming logic this information must be present; indeed the particular asserted program builds a particular proof of the output specification. In the case of the divide procedure, the output would be two integers, q and r, and a proof of the assertion a=b*q+ r & $0 \leq r < b$.

In order to treat these assertions as types, we must allow the type to depend on values of the other parameters. For example the output type a=b*q+ r & $0 \leq r < b$ is parameterized by its inputs, a, b. In general if B(x) is a type parameterized by x of type A, then we write *proc*(x:A)B(x) to denote those functions which on input a of type A procude a result of type B(a). This is the function space constructor of AUTOMATH [dBr70].

To describe the dependency between the input type $a \geq 0$ & b>0 and its parameters, a, b, we can use a dependent product construct of the form *prod*(x:A, y:B(x)) which consists of those pairs <a,b> such that a is of type A and b is of type B(a). In general we allow *prod*(x:A, y:B(x), z:C(x,y)), etc. This dependent product construct is essentially the infinite union of Scott [S70] and Martin-Lof [ML75]. The input to the divide procedure can be expressed with this constructor as:

$$prod(\mathrm{n}:\mathrm{N},\ \mathrm{a}:\mathrm{Z},\ \mathrm{b}:\mathrm{Z},\ \mathrm{p}:\mathrm{a} \geq 0 \wedge \mathrm{b} > 0).$$

The kind of logic described so far is essentially PL/CV2, but the syntax is somewhat altered so that assertions do not appear explicitly as types.

3.2 propositions as types

The programming logic outlined above is more complex than is necessary. Building on the propositions-as-types principle [H80, dBr70] Per Martin-Lof described a comprehensive theory of types in which propositions can be identified with types. The identification is suggested by this correspondence and has been extensively explained elsewhere [ML75, 79, C81, C82, BC81].

proposition	*type*
A & B	*prod* (A,B)
A $\vee$ B	*union* (A,B)
A $\Rightarrow$ B	*proc* (A)B
$\forall x{:}A.B(x)$	*proc* (x:A)B(x)
$\exists x{:}A.B(x)$	*prod* (x:A,B(x))

The union construction is as in Algol 68 [McG68].

3.3 new programming languages

The types *proc*(x:A)B(x), and *prod*(x:A, y:B(x)), *union*(A,B) clearly extend the usual Algol-like types *proc, struct* and *union.* They in fact provide types not available in any other programming language. If a notion of recursive data types is also provided as in [GMW79, CZ81] then these types subsume those of nearly all other programming languages.

A programming logic for a theory of this kind [CZ81, C82, BC81] is so rich that it appears to be adequate to formalize all of constructive mathematics (see [ML79]). If one considers the problems of reasoning about numerical algorithms and other sequential computational problems arising in mathematics, it is difficult to see how a language which falls short of providing these types will ever be completely adequate. This understanding will not be apparent from a viewpoint which starts with a language like FORTRAN or Algol and proceeds to extend it on an *ad hoc* basis. It is revealed by considering the nature of programming problems and the complete context in which they are posed

and solved. In this context, I believe that the view of a sequential program as an (efficiently) executable proof is more fertile than any other.

ACKNOWLEDGEMENTS

The programming logic PL/CV2 used to illustrate these points was designed and built by a team of my colleagues, Michael J. O'Donnell, Scott D. Johnson, Dean B. Krafft, Tat-Kung Chan, Daniel Zlatin, Carl D. Eichenlaub and Ryan Stansifer. To them and my colleagues on the PRL project, Joseph L. Bates and Alan Demers, I owe a great deal. I also want to thank Donette Isenbarger for carefully preparing the manuscript.

REFERENCES

[AA78]: Alagic, Saud and Michael A. Arbib, *The Design of Well-Structured and Correct Programs,* Springer-Verlag, NY, 1978.

[B79]: Bates, J.L., *A Logic for Correct Program Development,* Ph.D. Thesis, Department of Computer Science, Cornell University, 1979.

[BC81]: Bates J. and R.L. Constable, "Definition of Micro-PRL", Technical Report TR 82-492, Computer Science Department, Cornell University, October 1981.

[C71]: Constable, Robert L., "Constructive Mathematics and Automatic Program Writers", *Proc. of IFIP Congress,* Ljubljana, 1971, 229-233.

[C82]: Constable, Robert L., "Intensional Analysis of Functions and Types", Computer Science Department, University of Edinburgh, June 1982.

[CO78]: Constable, Robert L. and M.J. O'Donnell, *A Programming Logic,* Winthrop, Cambridge, 1978.

[CZ82]: 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.

[CJE82]: Constable, Robert L., S.D. Johnson and C.D. Eichenlaub, *Introduction to the PL/CV2 Pro-gramming Logic, Lecture Notes in Computer Science,* Vol. 135, Springer-Verlag, NY, 1982.

[dB80]: deBakker, Jaco, *Mathematical Theory of Program Correctness,* Prentice-Hall, Englewood Cliffs, NJ, 1980.

[dBr70]: deBruijn, N.G., "The Mathematical Language AUTOMATH", *Symposium on Automatic Demonstration, Lecture Notes in Math,* Vol. 125, Springer-Verlag, NY, 1970, 29-61.

[dBr80]: 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, 579-606.

[D76]: Dijkstra, Edsger W., *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[G80]: Goad, C., "Proofs as Descriptions of Computation", *Proc. of the 5th Conf. on Automated Deduction,* Les Arce, France, *Lecture Notes in Computer Science,* Vol. 87, Springer-Verlag, NY, 1980, 39-52.

[GMW79]: Gordon, M., R. Milner and C. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation, Lecture Notes in Computer Science,* Vol. 78, Springer-Verlag, 1979.

[G81]: Gries, D., *The Science of Programming,* Springer-Verlag, NY, 1981.

[H69]: Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", *Comm. ACM, 12,* October 1969, 576-580 and 583.

[H80]: Howard, W.A., "The Formulas-As-Types Notion of Construction" in *Essays on Combinatory Logic, Lambda Calculus and Formalism,* (eds., J.P. Seldin and J.R. Hindley), Academic Press, NY, 1980.

[K45]: Kleene, S.C., "On the Interpretation of Intuitionistic Number Theory", *JSL,* Vol. 10, 1945, 109-124.

[M74]: Manna, Zohar, *Mathematical Theory of Computation,* Mc-Graw Hill, NY, 1974.

[MM73]: Marini, D. and P.A. Miglioli, "Characterization of Programs and Their Synthesis from a Formalized Theory", *Proc. of Symposium on Math Found of Comp. Science,* High Tatras, 1973.

[ML75]: Martin-Lof, Per, "An Intuitionistic Theory of Types: Predicative Part", *Logic Colloquium,* 1973, (eds. H.E. Rose and J.C. Shepherdson), North-Holland, Amsterdam, 1975, 73-118.

[ML79]: Martin-Lof, Per, "Constructive Mathematics and Computer Programming", *6th International Congress for Logic, Method and Phil. of Science,* Hannover, August, 1979.

[McG78]: McGettrick, Andrew D., *Algol 68: A First and Second Course,* Cambridge University Press, Cambridge, 1978.

[P45]: Polya, G., *How To Solve It,* Princeton, 1945.

[R81]: Reynolds, John C., *The Craft of Programming,* Prentice-Hall International, Englewood Cliffs, NJ, 1981.

[S70]: Scott, Dana, "Constructive Validity", *Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125,* Springer-Verlag, 1970, 237-275.

[T73]: Troelstra, A.S., "Metamathematical Investigations of Intuitionistic Arithmetic and Analysis", *Lecture Notes in Mathematics, 344,* Springer-Verlag, NY, 1973.