MATHEMATICS AS PROGRAMMING†

R.L. Constable

TR 83-565

July 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

†This research was supported in part by NSF grant MCS81-04018. This paper was presented at the Logic of Programs Workshop at Carnegie-Mellon University in June 1983, whose proceedings will appear in Lecture Notes in Computer Science.

MATHEMATICS AS PROGRAMMING

R. L. Constable
Cornell University

In a sufficiently rich programming language it is possible to express a very substantial amount of mathematics in a natural way. I don't mean only that one can write down functions or solve equations, I mean that one can write theorems and proofs. Moreover, expressing mathematics in this way reveals its computational content and makes it available for use with digital computers.

This point is illustrated with reference to a programming language which is sufficiently rich in the above sense. I develop parts of Basic Recursive Function Theory and logic to illustrate the way in which doing some rather abstract mathematics is like programming. I chose BRFT in order to make certain points about the programming language by reflecting part of it inside itself. For example, while Church's Thesis can be false inside the language, it is true outside, reflecting in some sense the fact that while we may believe it, we do not expect to prove it. I chose a bit of logic to illustrate that the virtues of model theory (a certain abstractness and notation independence) are sometimes possible without sacrificing computational meaning.

I. Introduction

For some time now my colleagues and I have been concerned, as have several research groups, with making "programming more mathematical". This is in fact a common theme of this conference. We indeed advocate the practical use of "programming logics", and our latest effort at Cornell is concerned with treating constructive proofs as programs (see [2,7,10]). So the title of this talk might seem rather curious, "Mathematics as Programming". It is the converse of the business at hand.

In this paper then, I will emphasize another goal of our work which is not shared by many other groups. I will argue that this goal is at least as interesting as the common goal. To this end I want to make a point that is not widely understood, _that mathematics is programming in a very real sense_. Though I have believed this and written about it for a long time [7], it has only been in the last few years that I have seen a really simple and obvious road to this vantage point. I will try to take you on a rapid journey down that road, pointing out some of the interesting landscape as we go by - at a disorienting speed. One of the landmarks is a way of defining the partial recursive functions in a programming logic with only total functions, another is a way to do logic abstractly using constructive abstract data types. The journey begins over some rather dry territory, but if you stay with me I think you'll find these two highlights worth seeing.

II. A Theory of Types and Sets

We consider an informal programming logic with a rich type structure from which we define a notion of set. The theory we use here can be formalized in a number of ways (see for instance [3,11,20,21,23]; the reader unfamiliar with a careful theory of types should consult one of these sources for details). The critical notions in this account are the propositions-as-types principle, a notion of universe, and the concepts of set and subset. The notion of a quotient type also plays a role.

Starting from the programming language point of view, with Algol 68 [22,27] or ML [14] as examples, we have these type constructors present.

	ML-like	Algol-like
products (structures or records)	A×B,	prod(x:A,y:B)
unions	A+B	union(A,B)
functions	A→B	proc(x:A)B

There are primitive types such as _int_, _bool_, _char_, and we can consider also _nat_, for the natural numbers and _rat_, for the rationals. We might also use some notion of enumeration type, $\{e_1,\ldots,e_n\}$.

In order to treat types as objects, we add as well the type of types, _type_. We will consider as well a hierarchy, _large type_, _very large type_, etc. This will be a cumulative hierarchy, with _type_ in _large type_, etc. Such a hierarchy in one form or another is the essence of a type theory in the sense of Bertrand Russell [25]. The practical need for such a hierarchy arises from attempts to treat parameterized types and to abstract with respect to type. These notions provide a version of polymorphism by allowing types as parameters to procedures and definitions.

This type theory departs from ML and Algol in another critical way. The concepts of product and union are generalized. The product or record is generalized in the direction of the Pascal variant record [16]. We might call this construct a <u>dependent product</u> (see [11]).

	ML-like	Algol-like
dependent product	$\Sigma x{:}A.B$	prod(x:A,y:B)
	where occurrences of x in B are <u>bound</u> by x:A.	

The function space constructor is generalized in the direction of the AUTOMATH dependent function space.

	ML-like	Algol-like
dependent functions	$\Pi x{:}A.B$	proc(x:A)B
	where occurrences of x in B are <u>bound</u> by x:A.	

Now in the context of this very rich type structure it is possible to define a higher order predicate calculus using the propositions-as-types principle [15]. This is a fundamental organizing principle of AUTOMATH [13] and of Martin-Löf's type theories ([20,21], which we denote M-L75 and M-L82 respectively). We call theories based on this organization "AUTOMATH/Martin-Löf-like type theories."

Among the AUTOMATH/M-L-like theories are those designed at Cornell as programming logics [3,11]. These theories introduce additional concepts to facilitate information hiding and reasoning about programs as objects. First, we allow subsets on a type A, denoted $\{x{:}A|B\}$ for B a proposition on A (type on A), as the type $\Sigma x{:}A.B$ with the proof component, $b{:}B(a/x)$, hidden. Let $Pow(A) = \Sigma x{:}type.\ \exists P{:}A\to type.(x=\{y{:}A|P(y)\})/E$ where E is the equivalence relation defined by the condition

$$E(X,Y) \text{ iff } \forall x{:}A.(pred(X)(x) \Leftrightarrow pred(Y)(x))$$
$$\text{where } pred(\{x{:}A|B\}) = \lambda x.B$$

The rule for introducing elements into subsets is simply

$$\frac{a{:}A,\ b{:}B(a/x)}{a \in \{x{:}A|B\}}$$

The rule for using the information $y \in \{x{:}A|B\}$ is that if exp is defined over $\Sigma x{:}A.B$ and does not "use" the proof component, then exp is defined over $\{x{:}A|B\}$ (see [3] for details).

III. Basic Concepts

In this paper we are principally concerned with results about the nonnegative integers N. The finite intervals, $[n,m]$, are defined as subsets: $[n,m] = \{i{:}N|n\leq i\leq m\}$. We say that a type T is <u>finite</u> iff there is a bijection between T and some $[1,n]$.

There are subsets of $N\rightarrow N$ say $\{f,g\}$ for which we can not tell whether $f=g$ and hence for which we cannot say that $\{f,g\}$ is finite, but we can say it has no more than two elements. We call such types <u>subfinite</u>, they are subsets of finite sets.

One might expect a type to be <u>countable</u> precisely when there is a mapping $f{:}N\rightarrow T$ which is onto T, i.e. such that for all $t{:}T$, there is some $n{:}N$, $f(n)=t$. Such a notion is a bit too strong for subsets $\{x{:}A|B\}$ because perhaps by discarding the information B we can no longer find an n such that $f(n)=a$. So we say that a subset of A $\{x{:}A|B\}$ is <u>countable</u> iff the type $\Sigma x{:}A.B$ is countable. We call a subset of a countable set <u>subcountable</u>. It is easy to show:

Theorem: If I is countable and if for each $i{:}I$, $\{x{:}A|B(i)\}$ is countable, then $\{x{:}A|\exists i{:}I.B(i)\}$ is countable, i.e. a countable union of countable sets is countable.

We also have Cantor's fundamental theorem, stated here for N.

Theorem: $N\rightarrow N$ is uncountable.

IV. Function Classes

The type $N\rightarrow N$ includes all the functions from N to N definable in this theory. The definitions of such functions may involve arbitrary types, say

$A \to B$, arbitrarily high in the hierarchy of types, say $A \to B$ a very large type. We are interested in defining various subsets over $N \to N$. We also want to consider a generalization of it to the type of partial functions and its subtype of partial recursive functions. Since all the functions in $N \to N$ and all the partial functions are computable, it is interesting to compare these types to the partial recursive functions. We can even state a mathematical version of Church's thesis (CT).

First as a prelude to defining the partial recursive functions, let us define the primitive recursive functions. We want to consider functions of arbitrary finite arity, to this end define a parameterized cartesian product as

$$A^{(0)} = N_1 \quad \text{where } N_1 = \{0_1\}$$
$$A^{(1)} = A$$
$$A^{(n+2)} = A \times A^{(n+1)}$$

Officially $A^{(n)}$ is defined by recursion. In M-L82 for example it would be written

$$\text{ind}(u,v)(n,N_1,\text{ind}(z,w)(u,A,A\times w))$$

(where ind(u,v) is written R(u,v) in M-L82).

The functions of arbitrary arity over N are $\Sigma i{:}N.(N^{(i)} \to N)$. Call this $\mathbf{F}$. Given f in this type we write arity(f) for the first component, and fn(f) for the second, so arity(f):N and $\text{fn}(f){:}N^{(\text{arity}(f))} \to N$. We will define the primitive recursive functions as a subtype of $\mathbf{F}$ of the form $R = \{f{:}\mathbf{F} \mid \exists g{:}R^1.f = \text{fn}(g)\}$ where R^1 describes the structure of f.

Informally R^1 is obtained from the base functions successor, $\lambda x.s(x)$, constant $\lambda x.i$, projection $\lambda x_1,\ldots,x_u.x_i$ using the operations of composition and primitive recursion. The projection functions will be written as P(n,i) meaning select the i-th of n arguments. The composition operation, C, satisfies this condition: $C(n)(m)(h,g)(x) = h(g(x))$ where $x{:}N^{(n)}$ and g maps from $N^{(n)}$ to an m-vector. The primitive recursion operator, R, satisfies this condition:

$$R(n)(g,h)(0,x)=g(x),\ R(n)(g,h)(m+1,x)=h(m,R(n)(g,h)(m,x),x).$$

In formalizing all this we use the type $N_2 = \{0_2, 1_2\}$, essentially the booleans, and its associated case analysis operation.

$$case_2(0_2,a,b) = a,\ case_2(1_2,a,b) = b.$$

For eq: $N\times N\to N_2$ we let x eq y = 0_2 iff $x =_N y$. Also for $a,b \in A\times B$ we let 1of(a,b) = a, 2of(a,b) = b. With these definitions P belongs to $\Pi n{:}N^+.\Pi i{:}[1,n].N^{(n)}\to N$ where $N^+ = \{x{:}N|x>0\}$ and satisfies

$P(1,i) = \lambda x.x$

$P(n+1,i) = \lambda x.case_2(i \text{ eq } 1,\ 1of(x),\ P(n,i-1)(2of(x)))$

Using the induction form, the complete definition is

$$P = \lambda n.ind(u,v)(n-1,\lambda i.\lambda x.x,case_2(i \text{ eq } 1,\ 1of(x),v(i-1)(2of(x)))).$$

To define composition $C(n)(m)(h,g)(x) = h(g(x))$ we need these auxillary notions:

Let $T(n,m) = (N^{(n)}\to N)^{(m)}$ define $ap(0) = \lambda g.\lambda x.0_1$
$ap(1) = \lambda g.\lambda x.g(x)$
and $ap(m+2) = \lambda g.\lambda x.(1of(g)(x),\ ap(m+1)(2of(g)(x)))$

So ap(m)(g)(x) applies g in T(n,m) componentwise to $x{:}N^n$, e.g. if $g = (f_1,f_2)$ then $ap(2)(g)(x) = (f_1(x),f_2(x))$.

Officially we add n as a parameter to ap as follows

$$\begin{aligned}ap = \lambda n.\lambda m.case_2(&m \text{ eq } 0,\ \lambda g.\lambda x.0_1,\\ &ind(u,v)(m-1,\lambda g.\lambda x.g(x),\\ &\qquad \lambda g.\lambda x.(1of(g)(x),v(2of(g))(x))))\end{aligned}$$

Now we can define composition

$$C{:}\ \ \Pi n{:}N.\Pi m{:}N.\Pi h{:}(N^{(m)}\to N).\Pi g{:}(N^{(n)}\to N)^{(m)}.(N^{(n)}\to N)$$

by

$$C = \lambda n.\lambda m.\lambda h.\lambda g.\lambda x.h(ap(n)(m)(g)(x)).$$

So C(n)(m) composes h and the vector of functions g even for m=0.
We also need this vector operation

For $g{:}A\to B$ and $x{:}A^{(m)}$, $vect(m)(g){:}A^{(m)}\to B^{(m)}$, $vect(0)(g)(x) = 0_1$
$vect(1)(g)(x) = g(x)$, $vect(m+2)(g)(x) = (g(1of(x)),\ vect(m+1)(g)(2of(x)))$

Define the primitive recursion operator

$$R(n)(g,h)(0,x) = g(x)$$
$$R(n)(g,h)(m+1,x) = h(m,R(n)(g,h)(m,x),x)$$

We need $R:\Pi n:N^{+}.\Pi g:N^{(n-1)}\to N.\Pi h:N^{(n+2)}\to N.(N^{(n)}\to N)$
$R = \lambda n.\lambda g.\lambda h.\lambda m.\lambda x.ind(u,v)(m,g(x),h(u,v,x))$

Now we can define the base case of the inductive definition of the primitive recursive functions over N. Write $z:Ux:A.B$ in place of $z:\Sigma x:A.B$ when we intend z to denote 2of(z).

$$Base = \{f:N\to N \mid f=\lambda x.s(x)\} + (Ui:N.\{f:N^{0}\to N \mid f=\lambda x.i\} + Un:N^{+}.Ui:[1,n].\{f:N^{(n)}\to N \mid f=P(n,i)\})$$

Use 1,2of(x) for 1of(2of(x)), generally take $n_1,n_2,\ldots,n_k$ of(x) for $n_1$of($n_2$of($\ldots n_k$of(x)$\ldots$)).

$$\mathbb{R}'_{(0)}(N) = \Sigma i:N.\{f:N^{(i)}\to N \mid \exists g:Base.(g=inl(f) \vee g=inr(inr(f)) \vee g=inl(inr(f)))\}$$

$$\begin{aligned}\mathbb{R}'_{(n+1)}(N) = \Sigma i:N.\Sigma f:N^{(i)}\to N.\\ &[\exists g:\mathbb{R}'_{(n)}(N).(f=1,2of(g)).\\ &\vee\\ &\quad \exists n:N.\exists m:N.\exists h:\mathbb{R}'_{(n)}(N).\exists g:(\mathbb{R}'_{(n)}(N))^{(m)}.\\ &\quad (\forall i:[1,m].(arity(P(m,i)(g)) = n)\ \&\\ &\quad\ 1of(h)=m\ \&\ f=C(n)(m)(1,2of(h),vect(m)(\lambda z.1,2of(z),g)\\ &\vee\\ &(i>0)\ \&\ \exists g:\mathbb{R}'_{(n)}(N).\exists h:\mathbb{R}'_{(n)}(N).\\ &\quad (1of(g) = i-1\ \&\ 1of(h) = i+2\ \&\\ &\quad\ f = R(i)(1,2of(g),\ 1,2of(h)))]\end{aligned}$$

where arity(f) = 1of(f).

Define the primitive recursive <u>schemes</u> as $\mathbb{R}^{1}(N) = \Sigma j:N.\mathbb{R}'_{(j)}(N)$. With each element of $\mathbb{R}^{1}(N)$ there is a <u>level</u>, level(f) = 1of(f), an arity, arity(f) = 1of(2of(f)), a function, fn(f) = 1,2of(2of(f)), and a complete description of its construction, 2,2,2of(f). For example, $(0,(0,\lambda x.2))$ represents $\lambda x.2:N^{(0)}\to N$ but $(1,1,\lambda x.2,\ inl(0,(0,0,\lambda x.2),\ (0,1,\lambda x.x)))$ represents $\lambda x.2:N\to N$.

Partial Functions and μ-Recursive Functions

Define $\mathbf{P}(N) = \Sigma i{:}N.\Sigma P{:}N^{(i)} \rightarrow Type.((\Sigma x{:}N^{(i)}.P(x)) \rightarrow N)$. Call these the partial functions over N. In this case i represents the arity, P represents the dom and f the op.

We will define the μ-recursive functions as an inductive class in the style of $\mathbf{R}^1(N)$. We take $\mathbf{PR} = \Sigma j{:}N.\mathbf{PR}_{(j)}$. To accomplish this we must define C(n)(m), R(n) on partial functions and define min on partial functions. The interesting part is computing the domain component, dom. Here is how it is done for C and R.

```
C:Πn:N.Πm:N.Πh:{f:P(N)|arity(f) = m}.
           Πg:{f:P(N)^(m) |∀i:[1,m].arity(P(m,i)(f) = n)}.
           P(N)

C = λn.λm.λh.λg.
      (n,
       λx.∃p:(∀i:[1,m].dom(P(m,i)(g))(x)).
             dom(h)(pop(n)(m)(op(g))(tuple(m)(x,p))),
       λx.h(pop(n)(m)(op(g))(x))
                          )
```

where op is defined as fn except on partial functions and pop applies a vector of partial functions to the vector of arguments formed by applying tuple to $x{:}N^n$ and $p{:}\forall i{:}[1,m].dom(P(m,i)(g))(x)$.

Recall $P(N) = \Sigma i{:}N.\Sigma P{:}N^{(i)} \rightarrow Type.((\Sigma x{:}N^{(i)}.P(x)) \rightarrow N)$ and for $f \in P(N)$, arity(f) = i, dom(f) = P and $op(f){:}(\Sigma x{:}N^{(i)}.P(x)) \rightarrow N$.

As in APL there are numerous kinds of vector operation, e.g.

```
prop:Πm:N.Πg:P(N)^(m).ΠA:{F:Πj:[1,m].N^(n)→Type|
                          ∀i:[1,m].dom(P(m)(i)(g)) = F(i)}.

      Πp:(Πi:[1,m].(N^(n) × A(i))) where n = arity(g)

prop(1)(g,A,p) = g(p(1))
prop(m)(g,A,p) = (1of(g)(p(1)), prop(m)(2of(g), λi.p(i+1)))
```

To define the μ-recursive functions, **PR**, we need C, R, min on P(N). C has been defined. Consider now R.

$$R:\Pi n:N^{+}.\Pi g:\{f:P(N)\,|\,arity(f) = n-1\}.\Pi h:\{f:P(N)\,|\,arity(f) = n+1\}.P(N)$$
$$R = \lambda n.\lambda g.\lambda h.(n,\lambda m.\lambda x.\exists y:N.GD(m,x)(y),\ \lambda z.1,2of(z))$$

where GD is

$$\begin{aligned}
&\lambda m.\lambda x.ind(u,v)(m,\lambda y.\exists d:dom(g)(x).(y=g(x,d)),\\
&\qquad \lambda y.\exists z:N.(v(z)\ \&\ \exists d:dom(h)(u,z,x).\\
&\qquad\qquad (y=h((u,z,x),d))))
\end{aligned}$$

$$\begin{aligned}
&min:\Pi n:N.\Pi g:\{f:P(N)\,|\,arity(f) = n+1\}.P(N)\\
&min = \lambda n.\lambda g.\\
&\quad (n,\\
&\quad \lambda x.\exists y:N.(\forall z:[0,y].dom(g)(z,x)\ \&\\
&\qquad \exists p:(\forall z:[0,y].dom(g)(z,x).(op(g)(z,pz) =_{N} 0)),\\
&\quad \lambda z.least(n)(g,1of(z),1,2of(z),\ 1,2,2of(z),\ 0)
\end{aligned}$$

$$\begin{aligned}
\text{where } &least:\Pi k:N.\Pi g:\{f:P(N)\,|\,arity(f) = k+1\}.\Pi x:N^{(k)}.\Pi n:N.\\
&\quad \Pi d:(\Pi j:[0,n].dom(g)(j,x)).\Pi m:[0,n].N
\end{aligned}$$

$$\begin{aligned}
least_{(k)}(g,x,n,d,m) = {}& \text{if } m \geq n \text{ then } n\\
& \text{else if } g(((m,x),d(m))) = 0 \text{ then } m\\
& \qquad \text{else } least_{(k)}(g,x,n,d,m+1)
\end{aligned}$$

(Actually least is defined by recursion on (n-m).)

$$PR = \Sigma j:N.PR_{(j)}$$

$$\begin{aligned}
PR_{(0)} = {}& \Sigma i:N.\Sigma P:N^{(i)}\rightarrow Type.\Sigma f:((\Sigma x:N^{(i)}.P(x))\rightarrow N).\\
& \qquad \exists g:Base.(\forall x:\{y:N^{i}\,|\,P(y)\}.\ \text{"}f(x) = g(x)\text{"})
\end{aligned}$$

$$\begin{aligned}
PR_{(n+1)} = {}& \Sigma i:N.\Sigma P:N^{(i)}\rightarrow Type.\Sigma f:((\Sigma x:N^{(i)}.P(x))\rightarrow N)).\\
& [\exists g:PR_{(n)}.(op(f) = op(g)) \vee\\
& \exists k:N.\exists m:N.\exists h:PR_{(n)}.\exists g:PR_{(n)}^{(m)}.\\
& (arity(h)=m\ \&\ \forall i:[1,m].arity(P(m,i)(g)) = k\ \&\\
& \qquad dom(f) = dom(C(k)(m)(h,g))\ \&\ f = C(k)(m)(h,g)) \vee\\
& \exists k:N^{+}.\exists g:PR_{(n)}.\exists h:PR_{(n)}.(arity(g) = k-1\ \&\ arity(h) = k\ \&\\
& \qquad dom(f) = dom(R_{(n)}(g,h))\ \&\ op(f) = op(R_{(n)}(g,h))) \vee\\
& \exists k:N^{+}.\exists g:PR_{(n)}.(arity(g) = k+1\ \&\ dom(f) = dom(min(g))\ \&\\
& \qquad op(f) = op(min(g)))]
\end{aligned}$$

V. Formal Logic

Let SA stand for Simple Arithmetic, the theory defined in [19] Chapter 4. We can carry out inside our programming language the study of SA presented in [19]. To do so we define SA as a type and define inductively the type of theorems of SA.

$$\text{Let } AX{:}\Pi D{:}Type.\Pi zero{:}D.\Pi one{:}D.\Pi plus{:}D\times D\to D.\Pi times{:}D\times D\to D.$$
$$(plus(zero,one) = one \ \& \ \forall x{:}D.(\neg plus(x,one)=0) \ \&$$
$$\forall x{:}D.(\neg(x=0) \Rightarrow \exists z{:}D.(plus(z,one) = x)) \ \& \ \ldots)$$

Let SA be the abstract type[†]

$$\exists D{:}Type.\exists zero{:}D.\exists one{:}D.\exists plus{:}D\times D\to D.\exists times{:}D\times D\to D.$$
$$(AX(D,zero,one,plus,times)).$$

Given SA we define the terms over SA, Term(SA), and the <u>propositional functions over SA</u>, Prop(SA), inductively as follows:

$$Terms(SA) \subseteq \Sigma i{:}N.(D^{(i)}\to D)$$
$$Terms(SA)(0) = \{f{:}\Sigma i{:}N.(D^{(i)}\to D) \mid$$
$$(arity(f) = 0 \ \& \ op(f) = \lambda x.one \lor op(f) = \lambda x.zero)$$
$$\lor$$
$$(arity(f) = 2 \ \& \ op(f) = plus \lor op(f) = times)$$
$$\lor$$
$$\exists n{:}N.(f = P(n))\} \quad \text{where P is similar to the P of IV.}$$
$$Terms(SA)(n+1) = \{f{:}\Sigma i{:}N.(D^{(i)}\to D) \mid$$
$$\exists g{:}Terms(SA)(n).(f=g) \lor$$
$$\exists m{:}N.\exists h{:}Terms(SA)(n).\exists g{:}(Terms(SA)(n))^{m}.$$
$$(f = C(h,g)).\}$$

$$Terms(SA) = \Sigma j{:}N.Terms(SA(j))$$

Before we can define the propositions over SA we must describe how to treat quantification as a propositional operator. (It is already clear how to

---

[†] We have been calling such structures <u>abstract types</u>, indeed Gordon Plotkin pointed out to me that the proof rules for this type are in fact congruent to those for abstract types in such languages as CLU.

treat &, $\vee$, $\Rightarrow$ as operators on propositions, namely and(p,q) = $\lambda x.(p(x)$ & $q(x))$.)

Let the <u>propositional functions</u> over D be defined as

$$PF(D) = \Sigma i{:}N.(D^{(i)} \to Type).$$

Then define $\exists$: PF(D) $\to$ PF(D) by

$\exists(f) = \lambda x.\exists z{:}D.(f(z,x))$ for arity(f) > 0 and

$\exists(f) = f$ for arity(f) = 0.

Likewise define $\pi(f) = \lambda x.\forall z{:}D.(f(z,x))$ and $\pi(f) = f$ for arity(f) = 0.

Now define the propositional functions Prop(SA) = $\Sigma j{:}N$.Prop(SA)(j) where:

$$\begin{aligned}
\text{Prop(SA)(0)} = \; & \{f{:}\Sigma i{:}N.(D^{(i)} \to \text{Type}) \mid \\
& \exists(g_1,g_2){:}\text{Terms(SA)}.(\text{arity}(g_1) = \text{arity}(g_2) = \text{arity}(f) \;\&\; f = \lambda x.(g_1(x) = g_2(x))) \vee \\
& \exists p{:}\text{Atomic prop(SA)}.(f=p)\}
\end{aligned}$$

$$\begin{aligned}
\text{Prop(SA)(n+1)} = \; & \{f{:}\Sigma i{:}N.(D^{(i)} \to \text{Type}) \mid \\
& \exists g{:}\text{Prop(SA)(n)}.(f=g) \vee \\
& \exists(p_1,p_2){:}\text{Prop(SA)(n)}. \\
& (f = \text{and}(p_1,p_2) \vee \\
& \;\; f = \text{or}(p_1,p_2) \vee \\
& \;\; f = \text{imp}(p_1,p_2) \vee \\
& \;\; f = \exists(p_1) \vee \\
& \;\; f = \pi(p_1)) \vee \\
& \exists m{:}N.\exists p{:}\text{Prop(SA)(n)}.\exists g{:}(\text{Term(SA)})^m.(f = C(p,g)))
\end{aligned}$$

where C is the composition operator defined previously.

We can now consider an inductive definition of the class of theorems of SA, say Thm(SA). If we use a Hilbert-style system, then it will be easy to write an inductive definition of classes Thm(SA)(n) of those theorems provable from the axioms in n steps. Implication and universal quantification are defined in terms of an axiom scheme over propositions. We write a function K:Prop(SA)$\to$Prop(SA)$\to$Prop(SA) such that $K(p)(q) = p \Rightarrow (q \Rightarrow p)$ and a function S:Prop(SA)$\to$Prop(SA)$\to$Prop(SA)$\to$Prop(SA) such that $S(p)(q)(r) = (p \Rightarrow (q \Rightarrow r)) \Rightarrow (p \Rightarrow q) \Rightarrow (p \Rightarrow r)$. We assert that K(p)(q) and S(p)(q)(r) are theorems for any p,q,r. Here is a sketch of the definition. Now we can define the theorems of SA inductively as follows:

Thms(SA)(0) = {f:Prop(SA) | SAaxiom(f) $\vee$ equalityaxiom(f)}

Thms(SA)(n+1) = {f:Prop(SA) |

	$\exists$g:Thms(SA)(n).(f=g) $\vee$
andin	$\exists a_1,a_2$:Thms(SA)(n).(f = and(a_1,a_2)) $\vee$
andel	$\exists$(a,b):Thms(SA)(n).(b = and(a,f) $\vee$ b = and(f,a)) $\vee$
orin	$\exists$d:Thms(SA)(n).$\exists$g:Prop(SA).(f = or(d,g) $\vee$ f = or(g,d)) $\vee$
orel	$\exists$d:Thms(SA)(n).$\exists(g_1,g_2)$:Thms(SA)(n).$\exists d_1,d_2$:Prop(SA). (d = or(d_1,d_2) & g_1 = imp(d_1,f) & g_2 = imp(d_2,f)) $\vee$
impin	$\exists$(p,q,r):Prop(SA).(f = K(p)(q) $\vee$ f = S(p)(q)(r)) $\vee$
allin	$\exists$g:Thm(SA)(n).(f = Π(g)) $\vee$
$\exists$-in	$\exists$p:Prop(SA).$\exists$d:Terms(SA).(arity(p) $\geq$ 1 & $\exists$t:Thm(SA)(n). (t = C(p,d))) & f = $\exists$(p)) $\vee$
allel	$\exists$g:Thms(SA)(n).$\exists$t:Terms(SA).$\exists$p:Prop(SA).(g = Π(p) & f = C(p,t)) $\vee$
$\exists$-el	$\exists g_1,g_2$:Thms(SA)(n).$\exists$p:Prop(SA).(g_1 = $\exists$(p) & g_2 = imp(p,f))}

Put Thms(SA) = Σj:N.Thms(SA)(j). We know from "cardinality" results that Prop(SA) and Thms(SA) are countable types. That is

Thm $\exists e_p:N \xrightarrow{\text{onto}}$ Prop(SA) & $\exists e_t:N \xrightarrow{\text{onto}}$ Thms(SA).

The logically true theorems of SA are those which hold independently of the axioms, AX, that is propositions P such that

* $\forall$D:Type:$\forall$zero:D.$\forall$one:D.$\forall$plus:D×D→D.$\forall$times:D×D→D.P.

P is logically false iff * holds with ¬P for P. Notice that we are not speaking of "provability in SA" here but we are speaking of truth in the programming language. Also, if P is logically true, then ¬P is logically false.

Theorem 4.3.1 (of [19]) General Undecidability Theorem.

For any minimally adequate theory, say SA, there is no recursive function to separate its theorems, say Thms(SA), from logically false sentences, say False(SA).

Pf: Here is a version of the proof in [19] which is correct for the programming logic.

If $\phi_i(i) = 0$, then AX & S(i) is a theorem because $\phi_i(i) = 0$ is representable and AX is provable. So $\neg(AX \Rightarrow \neg S(i))$ is also a theorem.

If $\phi_i(i) = 1$, then $AX \Rightarrow \neg S(i)$ is logically true since $\phi_i(i) = 1$ is representable (so F(i,i,1) is provable from AX and $0 \neq 1$ is provable, so $\neg F(i,i,0)$ is provable from AX), and thus $AX \Rightarrow \neg S(i)$ is provable in any D. By definition $\neg(AX \Rightarrow \neg S(i))$ is logically false.

Let $S_0 = \{i{:}N \mid \phi_i(i) = 0\}$, $S_1 = \{i{:}N \mid \phi_i(i) = 1\}$.
Let $E_0 = \{j{:}N \mid j = e_t(\neg(AX \Rightarrow \neg S(i)))$ for $i \in S_0\}$
$E_1 = \{j{:}N \mid j = e_t(AX \Rightarrow \neg S(i))$ for $i \in S_1\}$

Notice that $E_0 \subseteq$ Thms(SA) and
$E_1 \subseteq$ False(SA).

So any recursive function to separate Thms(SA) and False(SA) would separate E_0, E_1 and hence also S_0, S_1 (for e_t an arithmetization of formulas).

QED.

Acknowledgements

I would like to thank Stuart F. Allen for his numerous helpful comments about this paper and for many stimulating discussions about doing mathematics in type theory. In response to this paper, Stuart has developed a very elegant alternative definition of partial functions which I hope will appear elsewhere.

I also want to thank Donette Isenbarger for her excellent preparation of the manuscript.

References

[1] Aczel, P., "The Type Theoretic Interpretation of Constructive Set Theory", Logic Colloquium '77, A. MacIntyre, L. Pacholaki and J. Paris (eds.), North-Holland, Amsterdam, 1978, 55-66.

[2] Bates, J.L. and R.L. Constable, "Proofs as Programs", Dept. of Computer Science Technical Report, TR 82-530, Cornell University, Ithaca, NY,

1982.

[3] Bates, J.L. and R.L. Constable, "The Nearly Ultimate PRL", Department of Computer Science Technical Report TR 83-551, Cornell University, April 1983.

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

[5] Beeson, M., "Formalizing Constructive Mathematics: Why and How?", Constructive Mathematics (ed., F. Richman), Lecture Notes in Computer Science, Springer-Verlag, NY, 1981, 146-190.

[6] Bourbaki, N., Elements of Mathematics, Vol I, Theory of Sets, Addison-Wesley, Reading, 1968.

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

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

[9] 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.

[10] 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.)

[11] 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.

[12] Curry, H.B. and R. Feys, Combinatory Logic, North-Holland, Amsterdam, 1968.

[13] 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.

[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] 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.

[16] Jensen, K. and N. Wirth, Pascal User Manual and Report, (2nd ed.), Springer-Verlag, New York, 1975.

[17] Jutting, L.S., "Checking Landau's 'Grundlagen' in the AUTOMATH System" (Doctoral Thesis, Eindhoven University), Math. Centre Tracts No. 83, Math. Centre, Amsterdam, 1979.

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

[19] Machtey, Michael and P. Young, An Introduction to the General Theory of Algorithms, North-Holland, NY, 1978.

[20] Martin-Löf, 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.

[21] Martin-Löf, Per, "Constructive Mathematics and Computer Programming", 6th International Congress for Logic, Method and Phil. of Science, North-Holland, Amsterdam, 1982.

[22] McGettrick, A.D., Algol 68, A First and Second Course, Cambridge University Press, Cambridge, 1978.

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

[24] Reynolds, John C, "Towards a Theory of Type Structure", Proc. Colloque su, la Programmation, Lecture Notes in Computer Science, 19, Springer-Verlag, pp. 408-425, 1974.

[25] Russell, B., "Mathematical Logic as Based on a Theory of Types", Am. J. of Math., 30, 1908, pp. 222-262.

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

[27] van Wijngaarden, A.B.J. et al, Revised Report on the Algorithmic Language ALGO 68, Supplement to ALGO BULLETIN, University of Alberta, 1974.