Annals of Discrete Mathematics 24 (1985) 21-38
© Elsevier Science Publishers B.V. (North-Holland)

CONSTRUCTIVE MATHEMATICS AS A PROGRAMMING LOGIC I: SOME PRINCIPLES OF THEORY[1]

Robert L. Constable

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853
U.S.A.

The design of a programming system is guided by certain beliefs, principles and practical constraints. These considerations are not always manifest from the rules defining the system. In this paper the author discusses some of the principles which have guided the design of the programming logics built at Cornell in the last decade. Most of the necessarily brief discussion concerns type theory with stress on the concepts of function space and quotient types.

Key Words and Phrases:

Automated logic, combinators, Edinburgh LCF, partial recursive functions, programming languages and logics, PL/CV, PRL, propositions-as-types, quotient types, strong intentionality, type theory.

INTRODUCTION

How do we choose the languages in which to express exact reasoning, mathematics and programming? In the case of logic and mathematics, language has evolved over a period of perhaps 2,000 years. We know enough to say that such language is not the "God-given expression of Truth". Indeed human genius, especially that of Frege, has played a major role. In the case of programming languages the period of evolution is shorter. It might appear dramatically shorter, but if we consider the building of relatively rigorous and symbolic language, then perhaps only the last 100 years are relevant even for mathematics and logic. Although we can learn a great deal from this period, not many formal languages from it were meant to be used; fewer still were in fact used. By contrast in the past 40 years hundreds of programming languages have been created, each vying for prominent use. Among these a few dozen are serious contenders with distinct characteristics, e.g. FORTRAN, ALGOL (60, W, 68), LISP, PL/I, ADA, etc. How then are these languages chosen? What are the principles of design that make them attractive, usable and enduring?

There have been studies which attempt to sort out the principles behind such languages [32]. But not many deep principles have emerged. We can classify languages based on their control structures and make illuminating comparisons [9,28]. We can discuss procedure calling mechanisms and other features of implementation. With more modern languages we can compare type structures and mechanisms for achieving modularity and protection.

While these differences among programming languages are fascinating and while their study may contribute to better designs and implementations, such differences do not reveal deep principles of language value. The deeper principles emerge at the level of programming logic.[2] Such logics must take a position about the structure of the universe in which we compute, about what is real and what actions are possible, about what is expressible and what is true. It will happen that these deeper principles determine other aspects of language structure and sometimes obviate the need for explanations in terms of implementation or syntactic style. For instance, principles about the nature of type equality may dictate that the equality relation is not decidable. This would rule out certain approaches to syntactic type checking.

I do not claim to know the principles that determine a programming logic's value. I am not even confident that we will ever be able to coherently state any, but I know from our work on programming logics at Cornell since 1971 that certain experiences and ideas emerged as significant and have shaped the systems we built. I will describe some of these ideas, formulating them as principles, and discuss their consequences. In particular I want to examine the decisions we made in designing the two versions of type theory we have used at Cornell, V3 [13] and PRL [2]. Since these theories are both closely related to Per Martin-Löf's well-known type theories, I will phrase my discussion in terms of them, using M-L75 for [29] and M-L82 for [30], and M-L when it doesn't matter which. For details of these systems, refer to the cited literature. (We use ML to denote the widely used programming language part of Edinburgh LCF [24].) First let me put the Cornell work in context.

Our work on programming logics goes back to 1970, when we suggested [8] using formal constructive mathematics as a programming language and investigated some of the theoretical issues. This program was explained more thoroughly in a series of lectures entitled "Constructive Mathematics as a Programming Language" given at Saarbrücken, Advanced Course on Programming in 1974. Here the connections to so called "program verification" were also explored. In 1975 we began the design of a working system which we then called a _programming logic_. This was reported in the book _A Programming Logic_ with Mike O'Donnell [15]. We incorporated procedural programs into a constructive weak second order logic and provided a proof checking system which employed several efficient decision procedures for subsets of the logic [16]. As we tried to treat data types and some version of constructive set theory, we encountered the work of Per Martin-Löf [29] which fit extremely well with our plans (see P1, P6) and with the principles we had isolated. We incorporated our ideas with his and with various notions from Edinburgh LCF into a programming logic V3 [13]. At the same time we were joined by Joseph L. Bates who had several significant ideas for greatly improving the applicability of these methods in a modern computing environment [1]. We proceeded to design a new and more grand system [3] with the acronym PRL, for Program Refinement Logic. We designed an "ultimate PRL" system and an initial core system [2]. Since 1980 Bates has led the implementation of the system, and together we have been experimenting with it and progressing toward the "ultimate" system.

PRINCIPLES

P1: The first principle of our work has been that _the logic of programming should itself be computational_ and thus self-contained. The principles of computation are as fundamental as any other and can be axiomatized directly. We see no theoretical or practical limits unique to computational explanations and no need for non-computational ones. So Occam's razor can prune away those systems of reasoning based on Platonic conceptions of truth or on classical set theory and its models of computability.

There are numerous systems of reasoning with purely computational meaning such as Skolem's quantifier free logic [37], Intuitionistic logic, Bishop's constructive analysis [5], Russian constructive logic, etc. Among these, Bishop's style of constructive reasoning seems to capture the core of all of the sufficiently rich computational logics. It indeed describes the computational core of classical mathematics. Based on Bishop's writings [5,6] we could and did imagine doing mathematics inside a programming logic. When we began, it seemed that a language like Algol 68 possessed the right concepts and that its "theory of data types" might be adequate to describe the mathematical types.[3] In retrospect such a view is not so terribly wrong, but we began our detailed work with a much simpler theory which focused first on elementary data types and procedural programs. As we said in [15], "...the programming logic does not fit the classical pattern of partial or total correctness because commands themselves are treated like statements; the predicate calculus is based on a block structured constructive natural deduction system..."

P2: All objects come with a type. One decision that comes early in the design of a logic or language is whether types are specified with objects or as properties of a universal collection of objects. If one focuses on the nature of computer memory or on the development of recursive function theory, it is tempting to think of a single universe of "constructive objects" from which types arise by classifying these objects. This is the approach of Lisp. One sees it in the logics of Fefermann [22] and recently in the programming language Russell [20]. This principle is also at work in set theory where one starts with a collection of individuals and builds a cumulative hierarchy of sets over them.

We argue that the clearest conception is that by saying how to build objects one is in fact specifying a type. So objects come with their types. This is similar to Bertrand Russell's conception [33] (belying the name of the programming language [20])and to the Algol 68 conception and to the M-L conception.

P3: Products and unions are basic. The nature of pairing is fundamental (although we know that in set theory it is reducible) as is a notion of uniting. Our treatment of unions is dictated by a deeper principle, that is propositions as types (P6).

P4: Functions are effectively computable and total. The notion that functions are effectively computable follows from P1; that they are total we take as a basic principle. The recursive function theory notion of a partial function can be derived from this concept in several ways. The notion of an algorithm or rule we take as "categorical" like that of type. It is a concept which is manifest in specific types. So there are rules for computing functions from A to B, but the same underlying rule may also compute from A' to B.

We classify a function to be in type $A \rightarrow B$ if the inputs are from A and the outputs are in B regardless of what types are used along the way in determining this value.

P5: The structure of functions is discernible. According to Frege functions arise by abstraction from certain linguistic forms. This principle is followed in M-L82 where the canonical form of a function is $(\lambda x)b$. All functions are determined by forms or expressions of the range type. This view is unlike that of Curry [17,18] who thinks of functions as built from primitive functions by application.

From Frege's and Martin-Löf's standpoint, the structure of functions is primarily a linguistic matter. Computation is also primarily a linguistic matter. One is thus led to the notion that equality of functions is extensional since other aspects are not internal to mathematics.

Technically it is not easy to adopt another view of functions. For example, in order to analyze the structure of f in $A \rightarrow B$ one would be forced to consider component functions g which might have arbitrary type (lying arbitrarily high in the hierarchy of universes as well). Nevertheless we have felt that there are sufficient benefits to explore the possibility of building a usable theory in which functions have discernible structure (see section III).

The principal reason that our type theories have provided a means to analyze the structure of functions is that we have encountered informal arguments about program structure which are powerful and necessary to a programming logic [12]. Other ways of capturing such arguments (see for example [10]) seem very cumbersome. We would like to deal with the issue at a fundamental level in hopes of finding a powerful solution.

P6: Propositions are types. Since 1975 with PL/CV we were treating proofs as objects, and we were aware of the Edinburgh LCF scheme for manipulating proofs as objects. Therefore when we read in Stenlund [36] and Martin-Löf [29] of the propositions-as-types principle, we were prepared to accept it completely. Indeed, we now see this as a basic principle of semantics; not as a semantic theorem to be used to interpret the logical operators, but as a principle to organize the entire theory. Revealing the role of this principle was one of the major accomplishments of M-L, and we have followed Martin-Löf's example. Although this notion appears also in deBruijn's AUTOMATH [19] and in Scott's "Constructive Validity" paper [35] and can be traced back to Curry and Feys [17] and Howard [26], we were not struck by its fundamental nature and its proper place in organizing concepts until reading [29].

P7: Type structure is discernible. For reasons similar to those that justify the view that the structure of functions is discernible, we claim that the structure of types is discernible as well. This is especially important in light of the propositions-as-types principle since we want to express inside our theories various algorithms for deciding simple classes of propositions. These algorithms work on the structure of propositions (see [12]).

Since type equality in M-L82 respects structure, our ideas here are close to Martin-Löf's. However in V3 for example, we are able to determine for any type whether and how it is built from the basic constructors, and we are able to argue by induction over the "structured part" of the type universes (see section III).

P8: Equality information is not needed in constructing objects. We knew from [15] that we did not need the information from equality assertions to execute proofs, and we learned from Bates [1] that one did not need this information to extract executable code from proofs. This led us to accept readily the M-L treatment of equality. But we did not recognize this as a basic principle until after studying M-L. We still considered the possibility in V3 [13] of storing with equality assertions the algorithms for deciding them.

Accepting this principle bears heavily on one of the features of our type theories, the use of quotient types, discussed in the next section.

FEATURES OF CORNELL TYPE THEORIES

1. Information Hiding

One of the distinguishing features of constructive reasoning is that frequent reference is made to various features of evidence that support an assertion. So for an existential statement, $\exists x.P$, a witness t for $\exists x$ is mentioned. In asserting that an object a satisfies a property P(a), a proof p of P(a) is implicitly kept available.

Type theories in the style of the M-L or AUTOMATH formalisms explicitly display all of the evidence that might be needed. Carrying around all of this information is a burden to expression. Therefore in practice it is essential to find mechanisms to selectively suppress it. This need has been studied in programming languages with rich type structures, and the notion of an abstract type has emerged (see [20,24,32]). In informal mathematics the problem can be avoided (see [5]), and in formalized classical set theory, evidence is always suppressed.

We propose two new mechanisms for dealing with information hiding, the quotient type A/B and the set type $\{x:A|B\}$. They are closely related. The quotient type provides a way to suppress equality information systematically; we examine it first.

Quotient Types

A type is determined not only by a method of construction but by a criterion of equality on the objects constructed (this notion too can be found in Frege and is prominent in Bishop's writings). The type can be changed by changing either the method or the notion of equality. However, it appears not to be essential to build in a capability to express this latter change. It can be reflected by pairing a type, say A, with a notion of equivalence on A, say E. However economical such a scheme is, it is not natural and it cuts off such types from the other type forming operations. A common example illustrates these issues.

Consider Bishop's definition of a real number as a Cauchy convergent sequence of rationals [5], letting Q denote the rationals,

(1) $R = \Sigma x{:}N\to Q.\Pi(n,m){:}Z^{+}.(|x(n) - x(m)| \le 1/m + 1/n)$.

He defines two real numbers to be equal, E(x,y), iff

(2) $\Pi n\in Z^{+}.(|x(n) - y(n)| \le 2/n)$.

We take the real numbers, $\mathbb{R}$, to be the quotient type of (1) with (2) written $\mathbb{R}$ = R/E. Then the equality relation on the type $\mathbb{R}$, $=_{\mathbb{R}}$, is E.

When we want to define the functions from $\mathbb{R}$ to $\mathbb{R}$ we simply take $\mathbb{R} \to \mathbb{R}$ and by the nature of equality on any type, it is known that these functions respect E.

The concept of a real can be defined without quotients. We take R as the reals. Then we can define the functions from $\mathbb{R}$ to $\mathbb{R}$, say $\mathbf{F}(\mathbb{R},\mathbb{R})$ following Bishop's notation, as $\{f\in R\to R \mid \forall(x,y)\in R.(E(x,y) \Rightarrow E(f(x),f(y)))\}$. But now we not only need to build another apparatus for function spaces, but we must carry around certain information which is not necessary in computing reals. In those cases where the information is needed the quotient construct cannot be used.

Here is a suggestive account of quotient types in the setting of M-L (also see [13]).

(1) $$\frac{A \text{ type} \quad \overset{(x\in A,\ y\in A)}{R \text{ type}}}{A/R \text{ type}}$$

(2) $$\frac{A\in U_n \quad \overset{(x\in A,\ y\in A)}{R\in U_n}}{A/R\in U_n}$$

Call A/R the Quotient of A by R

(3) $$\frac{A_1 = A_2 \quad \overset{(x\in A,\ y\in A)}{R_1 \Leftrightarrow R_2}}{A_1/R_1 = A_2/R_2}$$

(4) $$\frac{a\in A \quad \overset{(x\in A,\ y\in A)}{R \text{ type}}}{a\in A/R}$$

(5) $$\frac{a_1\in A \quad a_2\in A \quad e\in R(a_1/x,\ a_2/y)}{r\in(a_1 =_{A/R} a_2)}$$

So $a_1 =_{A/R} a_2$ iff $R^*(a_1/x, a_2/y)$ where R^* is the reflexive, transitive, symmetric closure of R (since for all types we have $a =_A a$ and $a =_A b$ iff $b =_A a$ and $a =_A b$ and $b =_A c$ implies $a =_A c$).

$$(6)\quad \frac{\begin{matrix}(w\epsilon A) \\ B\ \text{type}\end{matrix} \quad \begin{matrix}(x\epsilon A,\ y\epsilon A,\ e\epsilon R) \\ B(x/w) = B(y/w)\end{matrix} \quad z\epsilon A/R}{B(z/w)\ \text{type}}$$

$$(7)\quad \frac{\begin{matrix}(z\epsilon A/R) \\ B\ \text{type}\end{matrix} \quad \begin{matrix}(w\epsilon A) \\ b\epsilon B\end{matrix} \quad \begin{matrix}(x\epsilon A,\ y\epsilon A,\ w\epsilon R) \\ b(x/w) = b(y/w)\epsilon B\end{matrix} \quad z\epsilon A/R}{b(z/w)\epsilon B}$$

Ideally the conclusions of (6) and (7) would have the form "$z\epsilon A/R \vdash B(z/w)$ type" and "$z\epsilon A/R \vdash b(z/w)\epsilon B$" but the M-L82 style does not keep track of assumptions in the sequent style.

The quotient constructor provides a means for collapsing a lot of information of a certain kind to a single token. It extends the method of treating equality information from the pre-defined equalities to any user-defined equality.

Set Types

In addition to the quotient mechanism for compressing information, we propose a means of actually hiding it. There are important situations in which we need to know that a proof exists, but we do not need any of the details. In recursive function theory for example, the least number operator gives rise to such situations. We describe briefly now how this happens and then in section IV present more details.

There is a relatively straightforward way to build the theory of partial recursive functions inside M-L82. For example, the (small) partial functions $N \rightarrow N$ can be defined as those functions from subtypes of N into N, say $f\epsilon((\Sigma x\epsilon N.T(x)) \rightarrow N)$ for $T\epsilon N \rightarrow U_1$, then the μ-recursive functions can be defined as an inductive subset (see section IV for details). The least number operator, μ, is applied only when we know it succeeds, say we have a proof of $\exists y\epsilon N.f(x,y)=0$. So then $\mu y.(f(x,y)=0)$ can be defined "primitive recursively" from the proof information. One inconvenience of this approach is that partial recursive functions carry their domain information around explicitly. This information does not determine the value of the function, but it is necessary in defining it.

In order to restrict information available to the search operator, we introduce the set concept. Instead of taking $\{x:A \mid B\}$ to mean $\Sigma x\epsilon A.B$ we introduce a new type by these rules

$$\frac{A\ \text{type} \quad \begin{matrix}(x\epsilon A) \\ B\ \text{type}\end{matrix}}{\{x:A \mid B\}\ \text{type}} \qquad\qquad \frac{A_1 = A_2 \quad \begin{matrix}(x\epsilon A_1) \\ B_1 \Leftrightarrow B_2\end{matrix}}{\{x:A_1 \mid B_1\} = \{x:A_2 \mid B_2\}}$$

$$\frac{a\epsilon A \quad \begin{matrix}(x\epsilon A) \\ B\ \text{type}\end{matrix} \quad b\epsilon B(a/x)}{a\epsilon\{x:A \mid B\}} \qquad\qquad \frac{z\epsilon\{x:A \mid B\} \quad \begin{matrix}(x\epsilon A,\ y\epsilon B) \\ d(x)\epsilon C\end{matrix}}{d(z)\epsilon C}$$

We think of $\{x:A \mid B\}$ as $\Sigma x\in A.B$ without information about the second component. (An alternative approach to hiding information is to regard $\{x:A \mid B\}$ as merely a notational convention signifying a restricted way to use $\Sigma x\in A.B$. One might use $Ux\in A.B$ as a convention when the first component should be ignored.)

The μ-operator allows us to use this type under the following circumstances. Suppose $R\in A \to N \to N_2$ so that for every $x\in A$, $n\in N$, $R(x,n) = 0_2$ or $R(x,n) = 1_2$. Then given $x\in\{y:A \mid \exists n:N.(n>k \ \& \ R(y,n)=0_2)\}$, the μ-operator $\mu n>k.(R(x,n) = 0_2)$ defines a unique number without recourse to any information about the proof of $\exists n:N.(n>k \ \& \ R(y,n)=0_2)$ except that it exists. These observations are codified in the following rule (in the style of [30] R is now an expression):

$$\frac{\begin{array}{lll} (x\in A,\ n\in N) & & \\ R\in N_2 & k\in N & x\in\{y:A \mid \exists n:N.(n>k\&R = 0_2)\}\end{array}}{\mu n>k.(R=0_2) \in N}$$

We need an axiom that μ- is functional, i.e. if $R_1 = R_2$ then $\mu n>k.(R_1=0_2) = \mu n>k.(R_2=0_2)$, and we need a computation rule for μ which is this:

$$\begin{array}{ll} \text{if } R(k+1/n) = 0_2 & \text{then } \mu n>k.(R=0_2) = k+1 \\ \text{otherwise} & \mu n>k.(R=0_2) = \mu n>k+1.(R=0_2). \end{array}$$

2. Function Intensionality

For reasons discussed in section II, we want to analyze the structure of functions. It is not obvious how to analyze this structure mathematically when functions are presented as λ-terms because these terms involve the inherently linguistic notion of a variable in an essential way. The λ-terms suggest an analysis of descriptions but not of functions. However, the combinator form of a function involves only the concept of function and application and does suggest a means of analysis. To follow this line of thought requires that we define a certain combinatorial basis for the function spaces $A\to B$ and $\Pi x\in A.B$. This is a collection of basic functions, called combinators, from which all other functions can be built by application; such is the style of the theory of combinators [18,36]. Here is an informal account of how this can be done in the context of M-L82. Details of a related theory can be found in [12,13].

For any universe U_i and $A\in U_i$, $B\in\Pi x\in A.U_i$, $C\in\Pi x\in A.(\Pi y\in B(x).U_i)$ there is a function

$$\begin{array}{l} S_i \in \Pi A\in U_i.\Pi B\in(\Pi x\in A.U_i).\Pi C\in(\Pi x\in A.(\Pi y\in B(x).U_i)) \\ \quad .\Pi f\in(\Pi x\in A.(\Pi y\in B(x).C(x)(y))).\Pi g\in(\Pi x\in A.B(x)).\Pi x\in A.C(x)(g(x)) \end{array}$$

so that

$$\begin{array}{l} S_i\ ABC \in \Pi f\in(\Pi x\in A.(\Pi y\in B(x).C(x)(y))). \\ \quad \Pi g\in(\Pi x\in A.B(x)).(\Pi x\in A.C(x)(g(x))). \end{array}$$

The computation rule defining $S_i\ ABC\ f\ g$ is $S_i\ ABC\ f\ g = \lambda x.f(x)(g(x))$. For convenience we write $S_i\ ABC$ as S_{ABC}, making it clearer that S_{ABC} is a typed version of the S combinator. We can do the same for the K combinator so that $K_i\ AB \in \Pi x\in A.(\Pi y\in B(x).A)$, and $K_i\ AB\ a = \lambda x.a$.

Given these combinators as well as those of level U_i derived from the other constants of the theory such as pairing, (a,b), selection, E, induction, R, etc.;

we can define for any $f \in \Pi x \epsilon A.B$ in any U_i whose definition, f, does not involve notions beyond U_j, a combinator form, say $\mathrm{Comb}_j(f)$, equal to f. We need a form, level(f), which discovers the maximum level, j, used in building f.

These combinator forms can be analyzed inside the theory. For example, given $(S_{ABC}\ f)g$ we can build in operations op and arg to decompose it, e.g.

$$\mathrm{op}\ (S_{ABC}\ f)g = (S_{ABC}\ f) \text{ and}$$
$$\mathrm{arg}(S_{ABC}\ f)g = g.$$

We also must build in operations optype and argtype which compute the type of $(S_{ABC}\ f)$ and g from that of $(S_{ABC}\ f)g$. There must be versions of these functions which work for each level n.

Actually using S and K combinators or anything very similar is totally infeasible because they encode the structure of functions in a form too primitive to use. The size of a combinator for a λ-term, say λx.b, built from S and K and the constants might be quadratic in the size of λx.b (depending on the abstraction algorithm). But there are combinators which allow a much more intelligible description. One such I call the L-combinator. Using it there is a translation of λ-terms of length n to combinators in which the combinator is no more than log(n) times as large. The form of the combinator is L_A where A is a list of addresses of "locations" in the operand which are to be identified by L_A. For example, given $p \in N \to N \to N \times N$ such that $p(x,y) = (x,y)$, the function $\lambda x.p(x,s(x))$ has the form $L_{\{1.1,1.2\}}.p(I())(s())$ where the argument "holes" have addresses 1.1 and 1.2 respectively and where I is the identity combinator Ix=x. Such combinator forms are very similar to λ-terms (λx becomes L_A for A a list of occurrences of x), but L_A can be more easily treated as an operator.

In building a theory in which the structure of functions can be analyzed, it is important to provide a simple treatment of extensional equality as well, because it is a far more commonly needed equality than the intentional equality required to support the structural analysis. One way to insure a sufficiently simple treatment of extensional equality is to provide it as atomic, along side of the intentional equality. This requires distinguishing the function space with one equality from that with the other. For example $A \Rightarrow B$ and $\forall x \epsilon A.B$ can be used for the intentional space and $A \to B$ and $\Pi x \epsilon A.B$ for the extensional.

3. Type Intentionality

Since type equality in M-L82 respects structure, our ideas here are close to Martin-Löf's. However in V3 for example, we are able to determine for any type whether and how it is built from the basic constructors, and we are able to argue by induction over the "structured part" of the type universes. These capabilities can be consistently added to M-L82 by rules of the following sort (as above these "rules" only suggest a set of concepts using the M-L82 format, they are not definitively integrated into M-L82 to extend it).

First there is a form to decide the outer structure of a type in any universe, although only the form for the first universe is shown here.

$$\frac{x \epsilon U_0 \quad \begin{array}{c}(z \epsilon U_0)\\ C \text{ type}\end{array} \quad \begin{array}{c}(k=1,\ldots,10)\\ e_k \epsilon C(x/z)\end{array} \quad \begin{array}{c}(j \epsilon N)\\ e_1 \epsilon C(x/z)\end{array}}{\mathrm{typecase}\ (x,\ e_1,\ \ldots,\ e_{10}) \in C(x/z)}$$

There is a rule insuring functionality of typecase in all of its arguments, e.g.

$$\frac{x = x' \text{ in } U_0 \qquad \begin{array}{c}(k = 1, \ldots, 10)\\ e_k = e'_k \in C(x/z)\end{array}}{\text{typecase}(x, e_1, \ldots, e_{10}) = \text{typecase}(x', e'_1, \ldots, e'_{10}) \in C(x/z)}$$

There are eight rules for reducing typecase such as

$$\begin{array}{ll}
\text{typecase}(N_i, e_1,\ldots,e_{10}) & = e_1(i/j)\\
\text{typecase}(N, e_1,\ldots,e_{10}) & = e_2\\
\text{typecase}(a =_A b, e_1,\ldots,e_{10}) & = e_3\\
\text{typecase}(A\times B, e_1,\ldots,e_{10}) & = e_4\\
\text{typecase}(A+B, e_1,\ldots,e_{10}) & = e_5\\
\text{typecase}(A\to B, e_1,\ldots,e_{10}) & = e_6\\
\text{typecase}(\Sigma x\epsilon A.B, e_1,\ldots,e_{10}) & = e_7\\
\text{typecase}(\Pi x\epsilon A.B, e_1,\ldots,e_{10}) & = e_8\\
\text{typecase}(W x\epsilon A.B, e_1,\ldots,e_{10}) & = e_9
\end{array}$$

Using these rules we can define functions $U_0 \to N_2$ which decide whether a type has a certain structure. For example isprod $= \lambda x.$typecase$(x, 0_2, 0_2, 0_2, 1_2, 0_2, \ldots, 0_2)$ has value 1_2 iff x is a product type, A×B. Notice that typecase$(x, 0_2, \ldots, 0_2, 1_2)$ will equal 1_2 iff x is not one of the atomic types or structured types.

There are also functions to decompose the structured types. These provide what we call <u>strong intentionality</u>.

$$\frac{x \in U_i}{\begin{array}{l}\text{left}(x) \in U_i\\ \text{right}(x) \in U_i\end{array}} \qquad \frac{x = x' \in U_i}{\begin{array}{l}\text{left}(x) = \text{left}(x') \in U_i\\ \text{right}(x) = \text{right}(x') \in U_i\end{array}}$$

We then need axioms to define left, right such as

$$\frac{A \in U_i \quad B \in U_i}{\begin{array}{l}\text{left}(A\times B) = A\\ \text{right}(A\times B) = B\end{array}} \qquad \frac{x \in U_0 \quad \text{typecase}(x, 0_2, 0_2, 0_2, 1_2,\ldots, 0_2) = 1_2 \in N_2}{x = \text{left}(x) \times \text{right}(x)}$$

for each of the binary operators, x, +, →.

We also need forms for analyzing the structure of types built from families of types.

$$\frac{x \in U_0 \quad \text{typecase}(x, 0_2,\ldots, 0_2, 1_2, 0_2, 0_2) = 1_2 \in N_2}{\begin{array}{l}\text{index}(x) \in U_0\\ \text{fam}(x) \in \Pi y\epsilon\text{index}(x).U_0\\ \\ x = \Pi z\epsilon\text{index}(x).\text{fam}(x)(z)\end{array}}$$

We also have rules such as

$$\frac{A \in U_i \quad \overset{(x \in A)}{B \in U_i}}{\begin{array}{l}\text{index}(\Pi x{\in}A.B) = A \\ \text{fam}(\Pi x{\in}A.B) = \lambda x.B\end{array}}$$

MATHEMATICS IN TYPE THEORY

We can illustrate the ideas discussed in the previous sections by building a piece of mathematics in the type theory formalism. In particular, we examine the theory of primitive recursive and partial recursive function classes (using Kleene's μ-recursive function formalism). This development will illustrate the value of set types and quotient types and will also show the standard way of dealing with the structure of algorithms.

1. Basic Concepts

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

There are subsets of N→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 subfinite, they are subsets of finite sets.

One might expect a type to be countable precisely when there is a mapping f:N→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 $\{x{:}A \mid B\}$ 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 $\{x{:}A \mid B\}$ is countable iff the type $\Sigma x{:}A.B$ is countable. We call a subset of a countable set subcountable. It is easy to show:

Theorem: If I is countable and if for each i:I, $\{x{:}A \mid B(i)\}$ is countable, then $\{x{:}A \mid \exists i{:}I.B(i)\}$ 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→N is uncountable.

2. Function Classes

The type N→N includes all the functions from N to N definable in this theory. The definitions of such functions may involve arbitrary types, say A→B, arbitrarily high in the hierarchy of types, say A→B a very large type. We are interested in defining various subsets over N→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→N and all the partial functions are computable, it is interesting to compare these types to the partial recursive functions. We can even state an equivalent to Church's thesis (CT) in type theory as a mathematical statement.

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 <u>successor</u>, $\lambda x.s(x)$, <u>constant</u> $\lambda x.i$, <u>projection</u> $\lambda x_1,\ldots,x_u.x_i$ 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.

$$\text{case}_2(0_2,a,b) = a,\ \text{case}_2(1_2,a,b) = b.$$

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 $N^+ = \{x{:}N \mid x>0\}$ and satisfies

$$P(1,i) = \lambda x.x$$
$$P(n+1,i) = \lambda x.\text{case}_2(i \text{ eq } 1,\ 1\text{of}(x),\ P(n,i-1)(2\text{of}(x)))$$

Using the induction form, the complete definition is

$$P = \lambda n.\text{ind}(u,v)(n-1,\lambda i.\lambda x.x,\text{case}_2(i \text{ eq } 1,\ 1\text{of}(x),v(i-1)(2\text{of}(x)))).$$

To define composition $C(n)(m)(h,g)(x) = h(g(x))$ we need these auxiliary 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 $ap(m+2) = \lambda g.\lambda x.(1\text{of}(g)(x),\ ap(m+1)(2\text{of}(g)(x)))$

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

$$\begin{aligned}ap = \lambda n.\lambda m.\text{case}_2(&m \text{ eq } 0,\ \lambda g.\lambda x.0_1,\\ &\text{ind}(u,v)(m-1,\lambda g.\lambda x.g(x),\\ &\qquad \lambda g.\lambda x.(1\text{of}(g)(x),v(2\text{of}(g))(x))))\end{aligned}$$

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 $z:\cup x:A.B$ in place of $z:\Sigma x:A.B$ when we intend z to denote $2of(z)$.

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

Use $1,2of(x)$ for $1of(2of(x))$, generally take $n_1,n_2,\ldots,n_k$ $of(x)$ for $n_1of(n_2of(\ldots n_kof(x)\ldots))$. Let inr and inl be the injections into the disjoint union.

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

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 $\mathbf{P}(N) = \Sigma i{:}N.\Sigma P{:}N^{(i)}\to Type.((\Sigma x{:}N^{(i)}.P(x))\to N)$. Call these the partial functions over N. In this case i represents the arity, P represents the dom and the last component is the function itself.

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.

$$C{:}\Pi n{:}N.\Pi m{:}N.\Pi h{:}\{f{:}P(N)\,|\,arity(f) = m\}.$$
$$\Pi g{:}\{f{:}P(N)^{(m)}\,|\,\forall i{:}[1,m].arity(P(m,i)(f) = n)\}.$$
$$P(N)$$

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

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

$$prop{:}\Pi m{:}N.\Pi g{:}P(N)^{(m)}.\Pi A{:}\{F{:}\Pi j{:}[1,m].N^{(n)}\to Type\,|$$
$$\forall i{:}[1,m].dom(P(m)(i)(g)) = F(i)\}.$$

$$\Pi p{:}(\Pi i{:}[1,m].(N^{(n)} \times A(i)))\text{ where } n = arity(g)$$

$$prop(1)(g,A,p) = g(p(1))$$
$$prop(m)(g,A,p) = (1of(g)(p(1)),\ prop(m)(2of(g),\ \lambda i.p(i+1)))$$

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

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

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

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

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

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

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

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

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

Given $F = \Sigma i{:}N.(N^{(i)}\to N)$ we can define the subset of recursive functions by the condition $\{f{:}F\,|\,\exists g{:}\mathrm{PR}.(1,2\mathrm{of}(g) = 1\mathrm{of}(f)\ \&\ \forall n{:}N.(1,2,2\mathrm{of}(g)(n))\ \&\ 1,2,2,2\mathrm{of}(g) = 2\mathrm{of}(f))\}$. This class will be isomorphic to the class $\{g{:}\mathrm{PR}\,|\,\forall n{:}N.(1,2,2\mathrm{of}(g)(n))\}/E$ where $E(g_1,g_2)$ iff $1,2\mathrm{of}(g_1) = 1,2\mathrm{of}(g_2)$ & $1,2,2,2\mathrm{of}(g_1) = 1,2,2,2\mathrm{of}(g_2)$. We can prove that these subtypes are countable whereas F is uncountable. This is sensible because the elements of PR represent a very restricted way of building functions from N→N. For example, functions h:A→B for non-numerical types A and B are not used. Nevertheless, from <u>outside</u> the theory we can see that PR does in some sense represent all computable functions.

3. A Note on Partial Recursive Functions

The type PR captures Kleene's notion of an algorithm in a certain sense. We could also define in a similar way the notion of a Turing machine and the Turing computable functions from N to N. So type theory, just as classical set theory, can internalize the concept of an algorithm and the notion of a partial recursive function (see also [11]). But there is another more direct way in which type theory captures the concept of an arbitrary algorithm.

Consider those functions defined on types such as $\{x{:}N|B\} \to N$. They can involve the μ operator (and we could also allow a least fixed point operator, say FIX (as in [11])). In fact, for every μ-recursive algorithm, there is some function $f{:}\{x{:}N|B\} \to N$. Therefore if we consider the set of function terms that can be written in type theory, then for each algorithm there is a term. Indeed a computer system could execute any such term on any integer input, and some of the resulting computations would diverge. So in a sense all the algorithms are present in type theory exactly in the form seen in ordinary programming languages. The difference is that we cannot say anything about the arguments on which the algorithms diverge. That is because we are not interested in discussing the algorithms on such inputs.

CONCLUSION

Constructive type theory has contributed to our understanding of programming languages and logics; it also suggests a wide range of problems, from the very theoretical to the very practical. The preceding brief discussion was intended to reveal the dynamics of the subject and raise some of the theoretical issues, especially those dealing with information hiding. It also shows how the concept of a partial function can be accomodated, and in fact with simple additions to [3,13,30], namely the μ-operator or the fixed point operator, partial functions can be treated as naturally as in any other theory, say [24,34]. In a future paper we shall show how the concepts of recursive data type can be similarly treated, e.g. a simple addition of generators for type elements leads to a natural account of streams and other important data types.

ACKNOWLEDGEMENTS

I appreciate Per Martin-Löf's attention to this work and his sharing of ideas with me about the presentation of rules. I want to thank Joseph Bates for his perceptive comments on a draft of this paper and Stuart Allen for many stimulating discussions about type theory. Joe Bates and I are finishing a description of the type theory used in Nu-PRL, the document is tentatively titled "The Nearly Ultimate PRL" and will describe how type theory is used as a refinement style programming logic.

I also thank Donette Isenbarger for her careful and patient preparation of this document under the pressure of various deadlines.

FOOTNOTES

[1]This work was supported in part by NSF grants MCS-80-03349 and MCS-81-04018.

[2]As programming languages become richer, and as their definitions become more formal, the distinction between a programming language and a programming logic may disappear. A theory such as M-L82 [30] is indistinguishable from the applicative programming language described by Nordstrom [31]; the theories V3 [13] and PRL [2] are as much programming languages as they are logics. So these remarks apply to such "third generation" programming languages as well.

[3] In 1975 we considered using a subset of Algol 68 as the basis for our programming logic, but there wasn't sufficient local expertise to cope with the complexity of defining and implementing the subset. There was such expertise for PL/I. But we did use Algol 68 like notation in [15].

REFERENCES

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

[2] Bates, J.L. and Constable, R.L., Proofs as Programs, Technical Report, TR 82-530, Dept. of Computer Science, Cornell University, (1982).

[3] Bates J. and Constable, R.L., Definition of Micro-PRL, Technical Report TR 82-492, Dept. of Computer Science, Cornell University (October 1981).

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

[5] Bishop, E., Foundations of Constructive Analysis (McGraw Hill, NY, 1967).

[6] Bishop, E, Mathematics as a Numerical Language, in: Myhill, J. et al. (eds.), Intuitionism and Proof Theory (North-Holland, Amsterdam, 1970).

[7] Boyer, R.S. and Moore, J.S., A Computational Logic (Academic Press, NY, 1979).

[8] Constable, R.L., Constructive Mathematics and Automatic Program Writers, Proc. of IFIP Congress, Ljubljana (1971) 229-233.

[9] Constable, R.L. and Gries, D., On Classes of Program Schemata, SIAM J. Comput., 1:1 (March 1972) 66-118.

[10] Constable, R.L., Mathematics As Programming, Proc. of Workshop on Logics of Programs, Lecture Notes in Computer Science (1983).

[11] Constable, R.L., Partial Functions in Constructive Formal Theories, Proc. of 6th G.I. Conference, Lecture Notes in Computer Science, 135, Springer-Verlag (1983).

[12] Constable, R.L., Intensional Analysis of Functions and Types, Internal Report, CSR-118-82, Dept. of Computer Science, University of Edinburgh pp. 74 (June 1982).

[13] Constable, R.L. and Zlatin, D., Report on the Type Theory (V3) of the Programming Logic PL/CV3, Logics of Programs, Lecture Notes in Computer Science, 131, Springer-Verlag, NY (1982) 72-93. (To appear in TOPLAS, Jan. 1984.)

[14] Constable, R.L., "Programs and Types", Proc. of 21st Ann. Symp. on Found. of Comp. Science, IEEE, NY, 1980, pp 118-128.

[15] Constable, R.L. and O'Donnell, M.J., A Programming Logic (Winthrop, Cambridge, 1978).

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

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

[18] Curry, H.B., Hindley, J.R. and Seldin, J.P., Combinatory Logic, Volume II (North-Holland, Amsterdam, 1972).

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

[20] Donahue, J., and A.J. Demers, "Revised Report on Russell", Department of Computer Science Technical Report, TR 79-389, Cornell University, September 1979.

[21] Dummett, M, Frege Philosophy of Language, (Duckworth, Oxford, 1973).

[22] Feferman, S., Constructive Theories of Functions and Classes, in: Logic Colloquium '78 (North-Holland, Amsterdam, 1979).

[23] Fraenkel, A.A., and Bar-Hillel, Y., Foundations of Set Theory (North-Holland, Amsterdam, 1958).

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

[25] Gries, D, The Science of Programming (Springer-Verlag, 1982).

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

[27] Krafft, D.B., AVID: A System for the Interactive Development of Verifiable Correct Programs, Ph.D. Thesis, Dept. of Computer Science, Cornell University (August 1981).

[28] Luckham, D.C., Park, M.R. and Paterson, M.S., On Formalized Computer Programs, JCSS, 4 (1970) 220-249.

[29] Martin-Löf, P., An Intuitionistic Theory of Types: Predicative Part, in: Rose, H.E. and Shepherdson, J.C. (eds.), Logic Colloquium, 1973 (North-Holland, Amsterdam, 1975).

[30] Martin-Löf, P., Constructive Mathematics and Computer Programming, in: 6th International Congress for Logic, Method and Phil. of Science (North-Holland, Amsterdam, 1982).

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

[32] Pratt, T.W., Programming Languages: Design and Implementation (Prentice-Hall, Englewood Cliffs, 1975).

[33] Russell, B., Mathematical Logic as Based on a Theory of Types, Am. J. of Math., 30 (1908) 222-262.

[34] Scott, D., Data Types as Lattices, SIAM Journal on Computing, 5:3 (September 1976).

[35] Scott, D., Constructive Validity, Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125, Springer-Verlag (1970) 237-275.

[36] Stenlund, S., Combinators, Lambda-terms, and Proof-Theory, D. Reidel (Dordrecht, 1972).

[37] Skolem, T., Begründung der elementären Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Varanderlichen mit unendlichen Ausdehnunggsbereich, Videnskapsselskapets Skrifter 1, Math-Naturv K16 (1924) 3-38, (also in: From Frege to Gödel, 302-333 and in: Skolem's Selected Works in Logic, Fenstad, J.E. (ed.) Oslo, 1970).

[38] Teitelbaum, R. and Reps, T., The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, CACM, 24:9 (September 1981) 563-573.