PROGRAMS AS TYPES

by

Robert L. Constable

TR 80-423

May 1980

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

Programs are interpreted as types in a constructive type theory. Rules for a logic of programs can then be derived from rules for types. This approach is the basis of nonelementary reasoning in the PL/CV3 (program) verification system. This paper summarizes the type theory and shows how to develop higher order logic and algorithmic (or programming or dynamic) logic in the theory. The theory described here is an evolution from de Bruijn's AUTOMATH and Martin-Löf's Intuitionistic Theory of Types.

§1 INTRODUCTION

The subject of these lectures is program verification, in particular the latest work of my students, colleagues and me on the Cornell verification system (PL/CV) - with emphasis on the role of user defined data types. I view the subject in the context of a difficult long standing problem considered by some of the greatest philosophers and mathematicians, from Leibnitz to Wittgenstein, from Frege to Hilbert and Gödel. It is the problem of formal knowledge and mechanical thought. The questions are: "How do we know?" "What can machines know?" "How can machines help us know?"

This set of questions, like many great philosophical problems has become progressively more scientific. It now lies at the heart of computer science in the form, "How can machines help us know that they are doing what we want them to?" Work on these questions may improve our ability to build reliable computing systems. But that practical goal is not the driving force of this work.

Over the decades these questions have been posed most precisely and explored most deeply for mathematical knowledge. We will build on this tradition and ask about formal mathematics, "How can machines help us know mathematics?" "How are formal and informal mathematics related?"

Investigation of these questions has a theoretical component. Results such as Gödel's incompleteness theorem are relevant (but more for their techniques of proof and analysis than for their direct application.) But the question is also empirical, concerned with the capabilities of real computers dealing with real data. A paucity of data has limited the empirical work, but the potential for practical gain in the area of program verification is causing the generation of numerous pieces of formal mathematics.

The tools for empirical investigation of these questions are precisely the tools of computer science, which in the last two decades have become more powerful and sophisticated. Indeed, success in attacking this problem in the area of program verification feeds back to produce more powerful tools for attacking the problem more vigorously. Not only does this investigation enrich computer science, but it helps define the subject because in a large measure computer science is concerned centrally with the abilities of machines and the nature of formal languages.

§2 THE FORMAL FAMILY OF LOGICS

1. OVERVIEW

1.1 constructive objects

We are interested in reasoning about the data that can be manipulated by digital computers. The raw data of computers is symbolic, but these symbols represent ("symbolize") a mental world, and to reason about these objects, we must understand the mental world. For example, the natural numbers 1,2,3,...,10,..., 100,... are presented to us and machines as numerals, but we understand the abstraction to number in terms of a basic intuition about indefinite repetition.† It is the abstract concept of number and its underlying intuition which guides mathematics and programming with numerals.

To reason about other data types which can be concretely represented in the computer, such as graphs, lists, trees, etc. we must understand them as mental objects belonging to a mathematical universe. If we are to represent the concrete actions of the computer in the mental realm, then this must be the realm of constructions. From a logical point of view, we must develop a <u>theory of constructions</u>.

†We can understand the abstraction process as one of forming an equivalence relation in the class of all "notations" for numbers. Perhaps a number is specified precisely when we say how to convert a class of symbols into decimal numerals.

1.2 primitive types

We know that among the objects of the universe we shall have integers, Z and the subset N of non-negative integers, {0,1,2,...}. There will be characters Σ and the type of strings Σ^*. There will be Booleans, {true, false}. Let us suppose that there is some finite collection of disjoint primitive types $B_1,B_2,\ldots,B_b$ including Z,N,Σ^*,Bool. We also include the empty type, denoted ϕ.

1.3 binary type constructors

We are familiar with certain ways of building new types from old, for example $A \times B$ is the type of ordered pairs $<a,b>$ with a in A, b in B, and $A + B$ is the disjoint union of A and B. We are comfortable with $A \rightarrow B$ as the type of computable functions (terminating programs) with inputs from A and outputs in B. In each case a binary type constructor, $\times$, $+$, $\rightarrow$, builds new types from given types A and B. In addition, for each type constructor there are object constructors for building objects of the new types. This information is summarized in table 2.1 below.

	type constructor	object constructor	
	A, B types	a in A, d in B	
×	A × B	<a,b>	
+	A + B	<1,a> <2,b>	
→	A → B	conditional	IF
		(*if* d *then* e_1 else e_2 *fi*)	
		composition	Comp
		recursion	R
		S,K-combinators	S,K
		identity	I

Table 2.1

The type constructors we have described here exist in numerous programming languages and verification systems. The following table 2.2 summarizes some information concerning the notation used in four selected systems. Algol 68 [van Wijngaarden, *et al* 69] and SETL [Kennedy, Schwartz 75] are probably familiar. LCF is the Edinburgh verification system (Logic for Computable Functions) recently reported quite fully in [Gordon, Milner, Wadsworth 79]. Russell is a programming language designed at Cornell based on Dana Scott's concept of data type [Demers, Donahue 79]. AUTOMATH is a family of languages for writing formal mathematics which is a precursor of this family [de Bruijn 68].

	×	+	→
Algol 68	STRUCT(A id,B id)	UNION(A,B)	PROC(A)B (terminating elements of this type)
SETL	tupl	as represented by finite sets	as represented by finite sets
LCF			
ML	ty # ty	ty + ty	ty → ty (defined elements of this type)
PPλ	"	"	"
Russell	capsule id(...)	defined	available only through Image
AUTOMATH	defined	defined	$\prod_{x\epsilon\alpha} \cdot \beta x$ (see 1.6)

Table 2.2

1.4 definitional equality

New types can be defined by combinations of the binary constructors, so we may write

$$\begin{aligned} \text{DEFINE } T_1 &= (A + B) \rightarrow C, \\ T_2 &= A + (B\times C), \\ T_3 &= (T_1 \rightarrow T_2) + T_3 . \end{aligned}$$

Such definitions give rise to the concept of definitional equality. From the above definitions we can conclude

$$T_1 + T_2 = ((A + B) \rightarrow C) + (A + (B\times C))$$

Definitional equality applies to type expressions and is simply the closure of the equality and substitution rules applied to the defining equation.

At this point we have not introduced additional algebraic structure on types, e.g., we do not say that $A + B = B + A$ (because they are not equal), nor do we say $A + \phi = A$, nor even $(A\times B)\times C = A\times(B\times C)$.

1.5 dependent product construction

The product and union constructors are most convenient when they are n-ary so that $A_1\times\ldots\times A_n$ and $A_1+\ldots+A_n$ are legal terms. (This eliminates pressing need for an associativity axiom.) We adopt an Algol-like notation for the product, $\underline{\text{prod}}(A_1 id_1, \ldots, A_n id_n)$ where the identifiers id_i are used as selectors. Thus, given an element p of the product, $p.id_j$ selects the j-th component. Of course, these n-ary operations can be defined in terms of the binary operations.

One more simple generalization of products is very convenient. Sometimes we want the i-th component of the product to depend on the j-th for j<i. For example, when defining the notion of a finite automaton, we might write

$\underline{\text{prod}}(\text{D states, D alpha, states}\times\text{alpha} \to \text{states f, states } s_0)$.

We want to know that f depends on the first two compoenents of the product.

In order to have dependent products, we must have <u>parameterized types</u>. These are thought of as functions from the parameter domain into the class of types. Such constructs are perfectly sensible in this system and will be discussed at length in 1.8. The notion is not allowed in Algol 68 nor in the AUT68 version of AUTOMATH. It is allowed in LCF and Russell, and it is allowed in Martin-Löf's type theory [Martin-Löf 73].

1.6 product and sum of indexed family of types

The type constructors × and + can be generalized not only to n-ary constructors, but to constructors over an <u>indexed family of types</u>. Such operations are familiar in classical set theory, for example, Bourbaki [Bourbaki, 68 (Vol I, Chapter II, §4.1, §5.3 1968)] writes: Let $(X_i)_{i\varepsilon I}$ be an indexed family of sets

$\bigcup_{i\varepsilon I} X_i$ union,

$\bigcap_{i\varepsilon I} X_i$ intersection,

$\prod_{i\varepsilon I} X_i$ product.

We proceed in a similar way. Given a type I and an indexed family of types B(x), we can form

$\Sigma x\varepsilon I.\ B(x)$ sum

$\Pi x\varepsilon I.\ B(x)$ product

Our definition (following Martin-Löf) is that $\Sigma x\varepsilon I.\ B(x)$ is the type of all pairs $\langle x,b(x)\rangle$ where b(x) is of the type B(x), and $\Pi x\varepsilon I.\ B(x)$ is the type of all functions f which map x in I to an element f(x) in B(x). Let us see an example.

Let $[0,i]$ denote the initial segment of integers $0,\ldots,i$. Then $\Pi x \varepsilon N.[0,x]$ denotes the type of all functions f such that $f(x) \varepsilon [0,x]$. As another example, let N^n denote the n-fold cartesian product, then $\Pi x \varepsilon N.\ N^x$ has as elements those functions f such that $f(x) \varepsilon N^x$. One such function is $f(x) = \langle 1,2,\ldots,x \rangle$.

In an expression such as $\Pi x \varepsilon A.B(x)$ or $\Sigma x \varepsilon A.B(x)$, x is a <u>bound variable</u> whose scope is $B(x)$. We agree that $\Pi x \varepsilon A.B(x) = \Pi y \varepsilon A.B(y)$ (likewise for Σ) for any variable y not occurring in B or A .

These operations are not common in programming languages, but they have appeared exactly as we use them; Π in AUTOMATH [de Bruijn 68] and Π and Σ together in [Scott 70, Martin-Löf 73].

1.7 recursive types

New types can be defined recursively as in

$$\text{Tree} = N + \text{Tree} \times \text{Tree}\ .$$

In general we allow definitions of the form

$$T_i = F_i(T_1,\ldots,T_n) \quad i = 1,\ldots,n$$

where $F_i(T_1,\ldots,T_n)$ is a type expression in the T_i . To explain these types we also introduce the auxillary notion of languages generated by the definition, denoted

$$L(T_1,\ldots,T_n)_i \quad i = 1,\ldots,n\ .$$

Intuitively the languages are the collections of types generated by the i-th equation. Thus, L(Tree) consists of the types

$$N,\ N \times N,\ (N \times N) \times N,\ N \times (N \times N),\ \ldots\ .$$

More precisely, there is an empty type denoted ϕ , and $L(t_1,\ldots,T_n)_i$ is defined inductively where the base is obtained by substituting ϕ for each T_i . The recursive type T_i is isomorphic to $\Sigma x \epsilon L(T_i,\ldots,T_n)_i . x$.

Recursive type definitions $T_i = F_i(T_1,\ldots,T_n)$ $i = 1,\ldots,n$ and $S_j = G_j(S_1,\ldots,S_m)$ $j = 1, \ldots,m$ are equal iff $n = m$ and there is a correspondence (permutation) t , $F_i = G_{t(i)}$ such that $F_i(x_1,\ldots,x_n) = G_{t(i)}(x_1,\ldots,x_n)$.

Given a recursively defined type, we can introduce functions on it by recursion. We illustrate this for primitive recursive types of the form $T = \text{Basis} + R(T)$ where R is a type expression. A T-primitive recursive function t can be defined by the scheme

$$f(t,x) = \begin{cases} g_1(x) & \text{if } t \epsilon \text{Basis} \\ g_2(x) & f(t_1',x),\ldots,f(t_n',x),t) \end{cases}$$

where t_i' are obtained from t by operations of selection and application.

For example, given $\text{Tree} = N + \text{Tree} \times \text{Tree}$, define

$$\text{tmax}(t) = \begin{cases} t & \text{if } t \epsilon N \\ \max(\text{tmax}(t.1),\text{tmax}(t.2)) \end{cases}$$

where $t.i$ is the i-th component of the product, and max is the integer maximum function.

1.8 propositions as types

One lesson from our earlier work on verification is that proofs and programs should be objects of the universe. Similar lessons were learned from AUTOMATH and LCF, among the oldest verification efforts. The most elementary proofs are those for simple propositions, for instance a proof of $374218573 < 374218673$ is simply a computation. For atomic propositions, which depending on the formal system might be of the form $0 = 0$, $0 \neq 1$, $0 < 1$, ..., the proofs are simply axioms.

If proofs are to be objects and all objects are to be typed, then proofs must have types. The obvious association of proofs is with the propositions they prove. This leads to the notion that a proposition is a type whose elements are proofs. This concept of a proposition as a type has an interesting history discribed in [Constable 80]; the idea was suggested by H. Curry, developed by W. Howard and systematically explored by P. Martin-Löf.

To make this idea precise, we first agree that atomic propositions such as $0 = 0$, $1 = 1$, etc. are atomic types. We can distinguish them from other atomic types such as the integers, N, the characters, Σ, etc., but they have the same logical character. Next we can define compound propositions in terms of the type constructors. This definition of compound propositions reduces logic in a sense to type theory, and it defines a computational or constructive logic as opposed to the conventional classical logic. However, we can interpret classical logic into this constructive logic. Such an economical and computational approach to logic is extremely attractive in a programming logic. This point is discussed at length in [Constable 80]. Moreover, as Martin-Löf also points out, it is natural to require that a logic of computation include at least all of constructive mathematics. Any earlier stopping point will be artificial.

The following table defines compound propositions in terms of the already defined type constructors.

compound proposition	type
$A \& B$	$A \times B$
$A \vee B$	$A + B$
$A \Rightarrow B$	$A \rightarrow B$
$\neg A$	$A \rightarrow \phi$
$\exists x \varepsilon T.\ A$	$\Sigma x \varepsilon T.\ A$
$\forall x \varepsilon T.\ A$	$\Pi x \varepsilon T.\ A$

In addition to these compound propositions, there is also for each defined type, say $N \times \Sigma$, a corresponding defined identity proposition $I_{N\times\Sigma}(x,y)$ which asserts that x is identical to y. In this case, the definition is $<n,x>$ equals $<n^1,x^1>$ iff $n = n^1, x = x^1$. We also write this proposition as $<n,x> =_{N\times\Sigma} <n^1,x^1>$. In general, for a defined type T, there is a defined <u>T-identity proposition</u>, $=_T$.

From these definitions and the rules for types, we can derive rules of logic. This is done in detail in [Constable 80], the result is a derived natural deduction system. A brief account of this derivation is presented in 2 below.

1.9 types as objects

To a mathematician familiar with classical set thoery, it seems natural to think of types as sets and to treat them as objects. Thus, sets of sets, sets of sets of sets, etc., are easily conceived. But types are not sets. As conceived here types are data types, to use computer jargon. They are classifications of objects which can be represented in the computer. From this viewpoint, it is not obvious that types can themselves be treated as objects. Indeed, many programming languages which deal with types, such as Algol 68, do not treat them as objects. The mathematician looking at types as sets would be hard pressed to simultaneously treat them as objects representable in a computer, because they can be infinite.

Central to our view of types is that they are objects. It is important to be able to compute with types (hence with propositions, and as we shall see in §4 , with programs). This can be done because we think of types intensionally. So, just as we can manipulate programs as data objects in programming languages like LISP, even though extensionally they are infinite sets, so in this system we can manipulate types as data objects, even though extensionally they are infinite.

If types are to be objects, then they must have a type. We might try to argue that there is only one universe V of types. Given A,B types, we would say $A \epsilon V$, $B \epsilon V$. But if V itself is a type, then we would have $V \epsilon V$. This situation leads to a paradox as Girard has shown ([Martin-Löf 73]). Martin-Löf suggests a hierarchy of universes to avoid the paradox: $V_1, V_2, V_3, \ldots$. V_1 is the type of all small types; so $N \varepsilon V_1$, $\Sigma \varepsilon V_1$, etc. Also, each V_i is closed under the type forming operations.

V_1 itself is a type, called a large type. It can be used with the type constructors to form other large types such as $V_1 \times V_1$, $V_1 \rightarrow V_1$, etc. These large types have very large type, that is $V_1 \varepsilon V_2$, $V_1 \times V_1 \varepsilon V_2$, ... This hierarchy is repeated indefinitely.

This approach to treating types as objects appears completely adequate for both constructive mathematics and programming. To illustrate the approach in programming, we will discuss generic procedures in the context in §4 . Parts of a constructive set theory have been developed on this basis in [Constable 80].

1.10 strong intensionality

Not only can we treat objects as types, but we can analyze the structure of a type. For example, given types $A \times (B + C)$ and $(A \times B) + (A \times C)$ we can say that the first is a product while the second is a sum. The types are not identical even though there are natural isomorphisms between them. To accomplish this analysis of types, there are predicates at each level V_i which recognize the result of various type constructors (in fact, they recognize functions with types in V_i as values). For example, there is a predicate $PROD_i$ which will recognize products of the form $\Pi x \epsilon T.\ A(x)$. There are also functions to decompose types; for instance, there is a function to select the index set T of the product $\Pi x \epsilon T.\ A(x)$, and there is a function to select the function part, $A(x)$, of the product.

2. PROOF RULES

2.1 sample rules

The rules for this logic are organized by type constructor. For each constructor there are rules for introducing types built by it and objects built by the corresponding object constructors, there are rules for decomposing types and objects, (sometimes called elimination rules), there are rules to define predicates and functions which analyze the structure of objects built by the constructor. We illustrate these rules for the product type $\Pi x \epsilon A.\ B(x)$, omitting some detail which can be found in [Constable 80].

(i) introduction of Π

Given a type A in V_a and a function B which for each x of type A produces a type in V_b , then $\Pi x \epsilon A.\ B(x)\ \epsilon\ V_{max\{a,b\}}$.

(ii) introduction of functions (elements of $\Pi x \epsilon A.\ B(x)$)

If $b(x) \epsilon B(x)$ for each $x \epsilon A$, then $\lambda x \epsilon A.\ b(x)\ \epsilon\ \Pi x \epsilon A.\ B(x)$. (This is not the actual rule used in FORMAL-PL/CV3; instead there are rules for introducing functions by composition, conditional, application, and recursion (if A is recursive) as well as rules for introducing identity and constant functions. With these rules we can define functions corresponding to lambda terms of the form $\lambda x \epsilon A.\ b(x)$ for a certain class of expressions $b(x)$ not made precise here.)

(iii) elimination of functions

If $a \epsilon A$, $f \epsilon \Pi x \epsilon A.\ B(x)$, then $f(a) \epsilon B(a)$.

(iv) decomposition of products

If $PROD_i(T)$, that is, T is a product of level i, then

$\text{Product Index} : V_i \rightarrow V_{\text{Product Index Level}(T)}$

$\text{Product Function} : V_i \rightarrow V_{\text{Product Function Level}(T)}$

$\Pi x \epsilon \text{Product Index}(T) .\ \text{Product Function}(T)(x) = T$

$\text{Product Index}(\Pi x \epsilon A.\ B(x)) = A$

$\text{Product Function}(\Pi x \epsilon A.\ B(x)) = B$

(We also need a Dependence function not displayed here.)

2.2 natural expression of logical rules

When the rules for types are applied to the special type, propositions, the result is rules for logic. We will examine some of the rules, writing them in the natural deduction style used in PL/CV [Constable, O'Donnell 78]. Part of this style is a conventional notation for the logical operators which is summarized in the table below.

type theory	logic literature	PL/CV
A × B	A & B	A & B
A + B	A ∨ B	A | B
A → B	A ⇒ B	A ⇒ B
ΠxεT. A(x)	∀xεT. A(x)	ALL x T. A
ΣxεT. A(x)	∃xεT. A(x)	SOME x T. A

Below the predicate logic rules are written; first on the left as they are derived in the type theory; then on the right in a natural notation, specifically that used in PL/CV. There are also comments concerning what must be written in a PL/CV proof since some expressions can be automatically inferred by a type coercion mechanism and therefore need not be written by the user.

Assume throughout that A, B, C, and A(x) are propositions, T a type and let square brackets denote optional phrases.

	TYPE THEORY	LOGIC
1. AND	introduction	$\frac{A,B}{A\&B}$
	Given a ε A, b ε B then <a,b> ε A & B	If A and B are proved previously, then A & B can be added to the proof with no explicit justification. We can automatically produce the pair <a,b> from a ε A and b ε B.
	elimination	$\frac{A\&B}{A}$ $\frac{A\&B}{B}$.
	Given <a,b> ε A & B we know that a ε A and b ε B.	If A & B appears in a proof, then A and B can be added to the proof with no explicit justification. We can automatically produce the proofs a,b from <a,b>.

TYPE THEORY	LOGIC
2. OR	
introduction	$\frac{A}{A \mid B}$
Given $a \in A$, then $inj^{A \to A+B}(a) \in A \vee B$.	Given A in a proof, $A \mid B$ can be added with no explicit justification since the proof a of A can be converted, by $inj^{A \to A+B}$, to a proof of $A \mid B$.
elimination	
Given $d \in A \vee B$ and given deduction $c_1(a)$ of C from $a \in A$ and $c_2(b)$ of C from $b \in B$, then if casel(d) then $c_1(d)$ else $c_2(d)$ fi is a deduction of C from d and c_1, c_2.	Given a proof of $A \mid B$ and deduction Σ_1 of C from A and Σ_2 of C from B, then the proof expression BY CASES, $A \mid B$, PROOF; [ATTAIN C;] CASE A; Σ_1; C CASE B; Σ_2; C QED denotes the conditional expression if casel(d) then $c_1(d)$ else $c_2(d)$ fi.

TYPE THEORY	LOGIC

3. IMPLICATION

introduction

TYPE THEORY:

Given an expression $b(x) \varepsilon B$ for $x \varepsilon A$, the function

procedure(x)
    x ε A
    b(x)
end

is a proof of $A \Rightarrow B$.

LOGIC:

Given a deduction Σ corresponding to b a proof of B from assumption A, then the proof expression

```
BY[⇒] INTRO[DUCTION],
PROOF; [ATTAIN B;]
   ASSUME A;
      Σ
QED
```

denotes the function f.

elimination

TYPE THEORY:

Given $a \varepsilon A$ and $d \varepsilon A \Rightarrow B$, then $d(a) \varepsilon B$.

LOGIC:

$$\frac{A, A \Rightarrow B}{B}$$

Given A and $A \Rightarrow B$, then B can be concluded with no explicit justification because we can find the proof of B automatically from the proofs of A and $A \Rightarrow B$.

TYPE THEORY	LOGIC
4. ALL	
introduction	
Given an expression $a(x) \in A(x)$ for $x \in T$, the function	Given a deduction $\Sigma(x)$ corresponding to $a(x)$, then
procedure	BY [ALL] INTRO[DUCTION,]
$x \in T$	PROOF; [ATTAIN ALL x T.A(x)]
$a(x)$	ARBITRARY $x \in T$;
end	$\Sigma(x)$
	QED
is a proof of $\forall\, x \in T.A(x)$	denotes the function which is the proof of ALL x T.A(x).
elimination	
Given $f \in \forall\, x \in T.A(x)$, then for $t \in T$, $f(t) \in A(t)$.	Given a proof of $\forall\, x \in T.A(x)$, then A(t) BY ALLEL, ALL x T.A(x), t; denotes the typing $f(t) \in A(t)$ and the proof of $A(t)$ is $(\forall\, x \in T.A(x))(t)$.

5. SOME

TYPE THEORY	LOGIC
introduction	
Given $t \in T$ and $a(t) \in A(t)$, then $\langle t,a(t)\rangle \in \exists x \in T.A(x)$	Given $t \in T$ and $\Sigma(t)$ a deduction of $A(t)$, then A(t); . . . SOME x T.A(x)BY INTRO, t; denotes the typing $\langle t,a(t)\rangle \in \exists x \in T.A(x)$. The ellipsis means that A(t) is proved previously in the deduction. The pair $\langle t,\Sigma(t)\rangle$ can be created from t and Σ.
elimination	
Given t,a(t) $\langle t,a(t)\rangle \in \exists x \in T.A(x)$, then $t \in T$, $a(t) \in A(t)$.	Given a deduction of SOME x T.A(t), then CHOOSE t T WHERE A(t) denotes the components t and a(t) of the pair $\langle t,a(t)\rangle$. The scope of this CHOOSE operation is always determined by PROOF, QED delimters.

2.3 defining theories

The user can write functions which generate (by some efficient algorithm perhaps) a proof of a proposition C from certain hypotheses H_i . In the simplest case the user can write new inference rules, which in logic have the form

$$\frac{H_1,\ldots,H_n}{C} \quad .$$

This means that if h_i are proofs of H_i , then from them we can build a proof c of C . In functional notation,

$$c = f(h_1,\ldots,h_n)$$

$$f: H_1\times \ldots \times H_n \to C \ .$$

More generally the user might write a complex procedure to generate proofs.

In general, we will allow syntax of the form

$$c \text{ BY } f(h_1,\ldots,h_n)$$

for any proof generating function f . This is the form used for the application of user defined proof rules. Such rules are guaranteed to be correct because the function f must have the appropriate type.

Another way that the user can define proof rules is to write a decision procedure to recognize proofs. This would have the form

$$f: H_1\times\ldots\times H_n \times C \to \text{Boolean}$$

where H_i, C are classes of propositions and where $f(H_1,\ldots,H_n,C)$

is true for $H_i \epsilon \mathcal{H}_i$, $C \epsilon \mathcal{C}$ provided there is a function g which builds a proof $g(h_1,\ldots,h_n)$ of c for proofs h_i of H_i .

The PL/CV3 syntax for such procedures is

$$C \text{ BY } f, H_1, \ldots, H_n .$$

In PL/CV2, there is a built in procedure which generates proofs of arithmetic statements for arithmetic hypotheses. For instance, $2 \cdot x < 2 \cdot y$ can be generated from $x<y$. This is written in PL/CV2 as `2*x<2*y BY ARITH, x<y;`

3. EXAMPLES

3.1 a PL/CV3 example

Here is an argument in PL/CV3 for Cantor's theorem that the class of functions $N \to N$ is not enumerable. The theorem actually establishes that the effectively computable functions are not effectively enumerable, but it can be given a classical interpretation. (For a discussion of the constructive view of this theorem, see [Constable 76].)

CANTOR'S_THEOREM

```
01      ¬SOME g N→(N→N). ALL f N→N.  SOME n N.(g(n)=f)
02      BY INTRO, PROOF;
03         CHOOSE g N→(N→N) WHERE L: ALL f N→N.  SOME n N.(g(n)=f);
04         FOR x N DEFINE d(x)=g(x)(x)+1;
05         /* Note dεN→N */
06         SOME n N.(g(n)=d) BY ALLEL, L, d;
07         CHOOSE nd N WHERE g(nd)=d;
08         d(nd)=g(nd)(nd)+1;
09         g(nd)(nd)=d(nd);
10         FALSE BY ARITH, d(nd)=d(nd)+1;
11      QED;
```

The line numbers would be generated by the Proof Checker. As explained in 2.2 the notation SOME x T , ALL x T is PL/CV for $\exists x\epsilon T$, $\forall x\epsilon T$. At line 02 there is a proof block opener, PROOF, and the phrase "BY INTRO" which tells the reader (including the Proof Checker) that the proof will proceed by introducing the main operator, which is the negation sign, $\neg$. A negation $\neg A$ is introduced by assuming A and deriving FALSE (at line 10); thus the reader knows that after line 02 there is an assumption that $\exists g\epsilon N\rightarrow(N\rightarrow N).\forall f\epsilon N\rightarrow N.\exists n\epsilon N.(g(n)=f)$. From this existential statement we know that we can choose such a g , as done at line 03. Then a function d is defined at line 04. The definition is justified because g(x)(x)+1 is a well defined integer expression. This is determined by the Type Checking part of the Proof Checker. Since 04 defines a function, named d , it is possible to substitute d for f in the universal statement labeled L , $\forall f \in N \to N . \exists n N . (g(n)=f)$. This results in line 06 which in turn justifies the choice of nd in line 07. The rest of the proof is simply applying definitions and the laws of arithmetic.

3.2 a PL/CV2 example

In PL/CV2 we can write proofs for the integer, boolean and character data types. A complete development of the Fundamental Theorem of Arithmetic appears in [Constable 79]. This piece of algorithmic number theory involves 1,477 lines of PL/CV2 text including comments. The proofs were all checked by the Proof Checker on an IBM 370/168 in about three minutes. Below is a sample of this theory (lines 114 to 154).

```
114     FOR (X,Y) FIXED DEFINE DIV(Y,X)=SOME Q FIXED.X=Q*Y;
115     /* DIV(Y,X) IS READ "Y DIVIDES X" AND IS EQUIVALENT TO      */
115     /* MOD(X,Y)=0.  WITH INFIX OPERATORS IN PLCV WE WOULD WRITE*/
115     /* 'Y DIVIDES X' TO MATCH THE MATHEMATICAL NOTATION Y|X.    */
115     /* WE ESTABLISH NEXT THE PRECISE RELATIONSHIP BETWEEN DIV   */
115     /* AND MOD.  */
115     DIV_MOD_EQUIVALENCE:
115       ALL (A,B) FIXED WHERE B^=0. ( DIV(B,A)<=>MOD(A,B)=0 ) BY INTRO,
```

```
115        PROOF:
116        ONLY_IF_PART: DIV(B,A) => MOD(A,B)=0 BY INTRO,
116          PROOF;
117          CHOOSE Q0 FIXED WHERE A=B*Q0;
118          A=B*Q0+0;
119          0<ABS(B) BY ALLEL,ABS_LEMMA_1,B;
120          0<=0<ABS(B);
121          MOD_FACT: ALL (Q,R) FIXED.(A=B*Q+R & 0<=R<ABS(B) => R=MOD(A,B)
121          BY FUNCTION,MOD(A,B);
122          0=MOD(A,B) BY ALLEL,MOD_FACT,Q0,0;
123          QED; /#END OF ONLY IF PART#/
124        IF_PART: MOD(A,B)=0 => DIV(B,A) BY INTRO,
124          PROOF;
125          SOME Q FIXED.(A=B*Q+MOD(A,B)) BY FUNCTION,MOD(A,B);
126          SOME Q FIXED.(A=B*Q+0);
127          DIV(B,A);
128          QED; /# END OF IF PART #/
129        QED; /# END OF DIV_MOD_EQUIVALENCE #/
130      */
*THEOREM
130      /*/
130      DIVISOR_SIZE:
130      /# FOR D>=0, A>0, IF D DIVIDES A THEN D<= A #/
130      ALL (D,A) FIXED WHERE D>=0 & A>0 . (DIV(D,A) => D<=A)
130       BY INTRO, INTRO,
```

```
130     PROOF;
131          CHOOSE M FIXED WHERE M*D=A;
132          M<=0 | D > 0 BY ARITH, D>=0;
133              D = 0 | D > 0 BY ARITH, D>=0;
134              D<=A BY CASES, D=0 | D>0,
134              PROOF;
135                CASE D=0;
136                D<=A BY ARITH,D=0,0<=A;
137                CASE D>0;
138                  D<=A BY CASES, M<=0   M>=1,
138                  PROOF;
139                  CASE M<=0;
140                       M*D<=0 BY ARITH, M<=0, *, D>=0;
141                       '0'B BY ARITH, 1 <= A = M*D <= 0;
142                  CASE M>=1;
143                       M*D>=D BY ARITH, M>=1, *, D>=0;
144                  QED;
145                QED;
146     QED;
147      */
*THEOREM
147     /*/
147     TRANSITIVITY_OF_DIVIDE:
147     /* TRANSITIVITY OF DIVIDES */
147     ALL (A, B, C) FIXED .(DIV(A,B) & DIV(B,C) => DIV(A, C))
147     BY INTRO, INTRO,
```

```
PROOF;
      CHOOSE M1 FIXED WHERE M1*A = B;
      CHOOSE M2 FIXED WHERE M2*B = C;
      M2*(M1*A) = C; /* BY SUBSTITUTION */
      (M2*M1)*A = C;
      DIV(A,C) BY INTRO, M2*M1;
QED;
*/
```

§3 PROGRAMS AS TYPES

3.1 INTRODUCTION

In the universe of types described so far we find those which define objects we recognize as computations of programs. It is natural to identify programs with such types, just as we identify propositions as types of proofs. In fact, computations and proofs are very similar, and we recognize that programs are in a very real sense just elaborations of propositions. They look in some cases like infinitary propositions. It is noteworthy that this point of view agrees with that of Erwin Engeler described in [Engeler 68] arrived at from different considerations. This viewpoint has become the starting point for <u>algorithmic logic</u> [Rasiowa 77, Banachowski, et.al., 77, Mirkowska 71] and its variants, <u>programming logic</u>, [Constable 77] and <u>dynamic logic</u> [Pratt 76, Harel 79]. We will explore this connection in detail in section 4.

Viewing programs as types is tantamount to giving them a mathematical semantics. The goal of the Scott-Strachey approach to programming language semantics [Stoy 77, Milne, Strachey 76] is precisely to give such a mathematical (as opposed to machine oriented) semantics, which has now become known as <u>denotational semantics</u>. To clarify our new semantics we will compare the denotational approach and the type approach for a very simple class of programs.

One of the main reasons for specifying a mathematical semantics of programs is to provide a means for justifying proof rules for reasoning about them. The type interpretation will permit us to derive rules for programs from our rules for types. We do this in section 4.

3.2 A SIMPLE PROGRAMMING LANGUAGE

We illustrate the type semantics on the class of while schemes. A denotational semantics for this language can be found in [Milne, Strachey 76]. In presenting the syntax we adopt the convention that a name with a capital letter, Stmt , denotes a grammatical class, and the name without capitals, stmt , denotes an element of the class. Terminal symbols, such as $\underline{\text{do}}$, are underlined.

Simple_stmt ::= $f_1 | f_2 | f_3$ ...

Conditional ::= $\underline{\text{if}}$ Bexp $\underline{\text{then}}$ Stmt $\underline{\text{else}}$ Stmt $\underline{\text{fi}}$

Loop ::= $\underline{\text{while}}$ Bexp $\underline{\text{do}}$ Stmt $\underline{\text{od}}$

Stmt ::= Simple_stmt|Conditional|Loop|Stmt;Stmt

Bexp ::= $bexp_1 | bexp_2 | bexp_3$ ...

In the first definition the exact nature of simple statements is left unspecified, but intuitively they correspond to assignments and procedure calls. When we want to make assignments explicit we will use

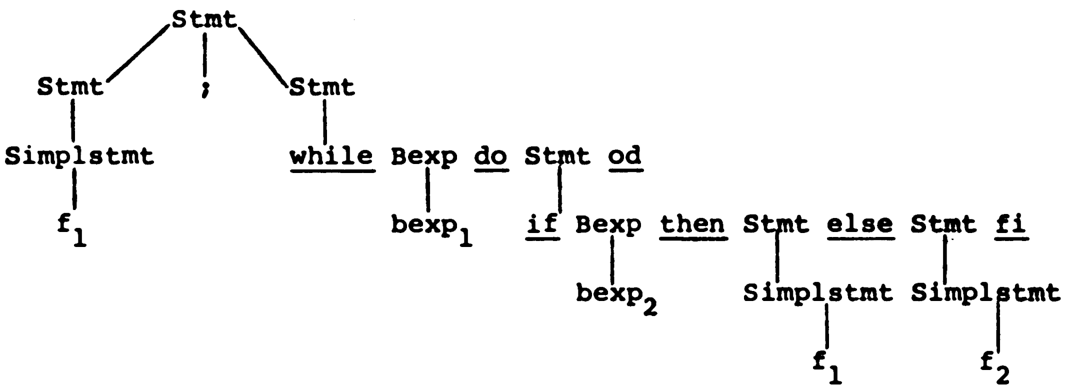
Assignment ::= Identifier := Expression

Identifier ::= $id_1 | id_2 | id_3$ ...

Expression ::= $exp_1 | exp_2 | exp_3$ ...

We also use $x, y, z, x_1, x_2, x_3 \ldots$ as identifiers.

Here is a sample while scheme and its parse tree.



f_1; while $bexp_1$ do if $bexp_2$ then f_1 else f_2 fi od

3.3 DENOTATIONAL SEMANTICS

Denotational semantics is built around the notion that the statements of a programming language describe actions of a processor on a state. The collection of all states is denoted S and an individual state, s . The meaning of a simple statement is a function from S to S . Let $\mathcal{M}[\![f_i]\!]$ denote the meaning of f_i , then

(i) $\mathcal{M}[\![f_i]\!] : S \to S$

The meaning of a boolean expression, $bexp_i$, is a map from state to boolean values, true , false . Thus,

(ii) $\mathcal{M}[\![bexp_i]\!] : S \to \{\underline{true}, \underline{false}\}$.

The meaning of compound statements is given as follows.

(iii) $\mathcal{M}[\![stmt_1 ; stmt_2]\!](s) = \mathcal{M}[\![stmt_2]\!](\mathcal{M}[\![stmt_1]\!](s))$

(iv) $\mathcal{M}\llbracket\underline{\text{if}}\ \text{bexp}\ \underline{\text{then}}\ \text{stmt}_1\ \underline{\text{else}}\ \text{stmt}_2\ \underline{\text{fi}}\rrbracket(s) =$

$$\begin{cases} \mathcal{M}\llbracket\text{stmt}_1\rrbracket(s) & \text{if } \mathcal{M}\llbracket\text{bexp}\rrbracket(s) = \underline{\text{true}} \\ \mathcal{M}\llbracket\text{stmt}_2\rrbracket(s) & \text{if } \mathcal{M}\llbracket\text{bexp}\rrbracket(s) = \underline{\text{false}} \end{cases}$$

(v) $\mathcal{M}\llbracket\underline{\text{while}}\ \text{bexp}\ \underline{\text{do}}\ \text{stmt}\ \underline{\text{od}}\rrbracket(s) =$

$$\begin{cases} \mathcal{M}\llbracket\underline{\text{while}}\ \text{bexp}\ \underline{\text{do}}\ \text{stmt}\ \underline{\text{od}}\rrbracket(\mathcal{M}[\text{stmt}](s) & \text{if } \mathcal{M}\llbracket\text{bexp}\rrbracket(s) = \underline{\text{tr}} \\ s & \text{if } \mathcal{M}\llbracket\text{bexp}\rrbracket(s) = \underline{\text{fa}} \end{cases}$$

When we wish to analyze simple statements further and discuss assignments we must refine the concept of state. We introduce a class of values, denoted Values. A state becomes a map, s:Identifiers $\rightarrow$ Values. An expression denotes a value. Thus,

(vi) $\mathcal{M}\llbracket\text{exp}\rrbracket : S \rightarrow \text{Values}$

(vii) $\mathcal{M}\llbracket x := \text{exp}\rrbracket(s) = s'$ where

$$\begin{cases} s'(\text{id}) = \mathcal{M}\llbracket\text{exp}\rrbracket(s) & \text{if } x = \text{id} \\ s'(\text{id}) = s(\text{id}) & \text{if } x \neq \text{id} \end{cases}$$

3.4 TYPE SEMANTICS

1. while schemes

To interpret while schemes as types we assume that simple statements f_i are types, intended to be types of computations.

A conditional statement, <u>if</u> bexp <u>then</u> $stmt_1$ <u>else</u> $stmt_2$ <u>fi</u> , is the type of a computation which is either a computation of bexp resulting in <u>true</u> followed by a computation of $stmt_1$ or a computation of bexp resulting in <u>false</u> followed by a computation of $stmt_2$. This can be written as a disjoint union of products

(i) $\{\langle bexp\rangle \mid bexp=\underline{true}\}\times stmt_1 + \{\langle bexp\rangle \mid bexp=\underline{false}\}\times stmt_2$.

We write "<bexp>" rather than "bexp" to denote the computation of bexp because conventional notation is that bexp denotes the result or value of the computation. It is also true that in some programming languages, like Algol 68, statements have values. But we shall adopt the convention that stmt denotes the type not the value.

The loop statement, <u>while</u> bexp <u>do</u> stmt <u>od</u> , denotes the recursive type defined by

(ii) $L= \{\langle bexp\rangle \mid bexp=\underline{false}\} + (\{\langle bexp\rangle \mid bexp=true\}\times stmt\times L)$.

Temporarily we let $stmt_1;stmt_2$ denote the type

(iii) $stmt_1 \times stmt_2$.

Notice that only finite computations are defined by these equations; thus if the loop <u>while</u> bexp <u>do</u> stmt <u>od</u> fails to terminate, then the type L is empty. If we want to discuss nonterminating computations such as <u>while</u> <u>true</u> <u>do</u> stmt <u>od</u> , then we talk about the type $L = stmt + (stmt \times L)$ which includes arbitrarily long computations.

2. parameters

Normally while schemes are parameterized by the elements of some input domain A . For $x\varepsilon A$, let $B(x)$ denote a while scheme type, then $\Pi x \epsilon A.\ B(x)$ denotes the class of executions of the program B on inputs from A . For the deterministic while schemes considered so far, any $f \epsilon \Pi x \epsilon A.\ B(x)$ is determined by executing $B(x)$ in the usual way, so there is a unique f in this type (just as there is a unique f in the type $\Pi x \epsilon N.\ B(x)$ where $B(x) = x^2$). The type $\Pi x \epsilon A.\ B(x)$ does not have an element unless $B(x)$ is nonempty (i.e., terminates) for each x in A . Thus A is the domain of the program B .

If we are interested in possibly nonterminating computations of loops on inputs from A , then we take as the type of
<u>while</u> bexp(x) <u>do</u> stmt(x) <u>od</u>

$$L(x) = \{\langle bexp(x)\rangle \mid bexp(x)=\underline{false}\} + \{\langle bexp(x)\rangle \mid bexp(x)=\underline{true}\} + \{\langle bexp(x)\rangle \mid bexp(x)=\underline{true}\} \times stmt(x) \times L(out_x(stmt(x))) \ .$$

The function out_x selects a value from the type $stmt(x)$ in ways to be discussed below (5).

Such an $L(x)$ cannot be empty as long as $bexp(x)$ is not empty.

3. while schemes with assignment

Consider a sequence of assignments

$$x_1 := exp_1;\ x_2 := exp_2;\ \ldots;\ x_n := exp_n$$

where each x_i is distinct and exp_i can depend on any x_j, $j<i$ for example,

$$x_1 := x+y;\ x_2 := x_1 * y;\ x_3 := x_1 + x_2 \ .$$

One straight forward meaning for this program is the sequence or n-tuple of values computed $exp_1, exp_2, \ldots, exp_n$. The names x_i

are used to specify the relationships among values. They also serve, incidentally, to name the components of the n-tuple.

If this is our view of the program segment, then we know precisely what type to ascribe to it, namely a dependent product with x_i as component selectors:

$$\underline{\text{prod}}(x_1{:}exp_1, x_2{:}exp_2, \ldots, x_n{:}exp_n) \ .$$

(Note, concurrent assignment such as $x,y:=exp_1,exp_2$ can be treated as assignment of a pair $x:=\langle exp_1,exp_2\rangle$, and x.1,x.2 can be used to select components.)

4. efficiency considerations, reusing variables

Our intention is to interpret every while scheme with assignments as a type by interpreting each block as a dependent product and interpreting conditionals and loops as in 1. A <u>block</u> is a maximal sequence of statements, $stmt_1;\ldots;stmt_n$ generated using the production Stmt::=Stmt;Stmt.

This simple view becomes clouded by the need to interpret a number of important conventions. Consider the program

L1:x:=x+1; L2:y:=x+1 .

In the component at L2 , the expression x+1 depends on the first component. As a type expression this could be written

<u>prod</u>(L1:x+1, L2:L1+1) .

But this expression makes it appear that three variables are necessary, L1, L2 and x; when in fact, x,y will suffice. Apart from these efficiency considerations, it is clear that all variables in a block of pure assignments could be uniquely named. The convention of reusing variables could be regarded as a notional convenience for control of memory allocation.

5. economy of description -- reusing variables

Another notational convention requires explanation. Consider the program

L1:$x:=x_0+y_0$; L2:<u>if</u> b <u>then</u> $x:=exp_1$ <u>else</u> $y:=exp_2$ <u>fi</u>; L3:z:=g(x,y) .

At L3 , the references x,y depend on the value of the conditional. If b is true, then x refers to L2's x while if b is false it refers to L1 . Likewise, y's reference at L3 depends on b . Using the case discriminating functions already in the theory, we can regard L3 as an abbreviation for

L3':g(<u>if</u> casel(L2) <u>then</u> L2.x <u>else</u> L1 <u>fi</u>,
<u>if</u> casel(L2) <u>then</u> y <u>else</u> L2.y <u>fi</u>) .

The abbreviation g(x,y) is used for two distinct reasons. First it displays "memory management" information. Second, it summarizes the conditional information by using the context to determine the reference of x,y .

Another notational convention which accomplishes the same purpose is used in PL/CV2. The meaning of x,y is brought out from the then and else blocks to the outer level of the conditional using an attain statement. We write

L2: <u>attain</u>$((b \Rightarrow x=exp_1 \& y=y_0) \& (\neg b \Rightarrow y=exp_2 \& x=x_0+y_0))$; <u>if</u> ... <u>fi</u> .

The proof rules for program types will be expressed in terms of the common abbreviations, making them easy to use. However, user defined rules for program types must refer to the unabbreviated form -- making them difficult to use.

6. feedback in iteration -- reusing variables

One of the most important notational conventions in procedural langauges involves reusing variables to indicate feedback relations in iterative loops. For example, consider

$$\underline{\text{while}}\ b(x)\ \underline{\text{do}}\ x:=x+y, y:=x+y\ \underline{\text{od}}\ .$$

The variables x,y play a dual role, both as parameters (<u>free variables</u>) and component selectors (<u>bound variables</u>). In their first occurrence on the right, they are bound references to previous components of the block, indicating a dependency. This is illustrated in the diagram below:

(x:=x+y;y:=x+y);(x:=x+y;y:=x+y);(x:=x+y;y:=x+y);...

This dependency is easily represented in the loop definition by specifying the appropriate parameter in the recursive call:

$$W(x_1,x_2) = (b(x_1)=\underline{\text{false}}) + (b(x_1)=\underline{\text{true}}) \times \text{body}(x_1,x_2)$$
$$\times\ W(\text{body}(x_1,x_2).x, \text{body}(x_1,x_2).y)$$

To express the value of the "variables" x,y upon termination of the loop, we need a simple way to refer to the last copy of the body in the sequence

$$(b=\text{true}) \times \text{body} \times (b=\text{true}) \times \text{body} \times \ldots\ .$$

The function $\text{last}(W(x_1,x_2))$ produces the last body executed. To pick values of the variables we write $\text{last}(W(x_1,x_2)).x$, $\text{last}(W(s_1,x_2)).y$. But as with the conditional, we use a simple convention of denoting these values by x and y in statements occurring beyond the loop, as in

L1: <u>while</u> b <u>do</u> x:=x+y;u:=x+y <u>od</u> ; L2:g(x,y) .

§4 DERIVING RULES FOR PROGRAMS

4.1 INTRODUCTION

In principle, because we are working in a very rich type theory, we could express programs as functions $f{:}A \to B$ and assertions about programs as ordinary higher type propositions. For example, a procedure $p(\bar{x},\bar{y})$ which modifies parameters $\bar{y}=y_1,\ldots,y_n$ could be viewed as producing a structure whose y_i component is denoted by $p(\bar{x},\bar{y}).y_i$. Assertions about p would have the ordinary form

$$A(p(\bar{x},\bar{y}).y_1,\ldots,p(\bar{x},\bar{y}).y_n) \ .$$

But experiehce has taught us that problems of scale demand a new notation. Some programs are too large to be treated as simple mathematical functions, they are more like sections of mathematical text. To understand them, we must convey information incrementally inside the program. This leads us to the notion of <u>a program with assertions</u> (sometimes called an <u>asserted program</u>[†]).

4.2 PROGRAMS WITH ASSERTIONS

The type interpretation of programs extends nicely to programs with assertions. Consider for example the segment

$$x>0;\ x{:=}x-1;\ x\geq 0 \ .$$

---

† We will sometimes use the term asserted program to refer to a program which has been asserted to halt.

This describes a certain triple, <proof, integer, proof> , and we view the segment as a product

$$\underline{\text{prod}}\ (L1{:}x>0,\ L2{:}x-1,\ L3{:}L2\geq 0)\ .$$

This view contrasts to that which considers a program with assertions as a notation for proofs.

So to add assertions to programs we simply allow propositions among the types allowed as components of products.

4.3 A PROOF RULE FOR LOOPS

Let us derive a rule for reasoning about loops from the rules for types. The computation defined by a loop is undertaken to attain some goal. The direct goal that loop execution accomplishes is building a computation of the loop type. But presumably the purpose of the computation is to attain a higher goal, usually one described by a proposition G .

One way that a loop computation attains a goal G is to get closer to G on each iteration by building more and more of the proof of G . Indeed, the loop computation can be regarded as an inductive construction of the proof of G . Consider the following while loop, $W(\bar{x})$, where the inputs $\bar{x}=x_1,\ldots,x_n$ are of type T , $\bar{x}\varepsilon T$.

$$W(\bar{x})=(\text{bexp}(\bar{x})=\underline{\text{false}})+((\text{bexp}(\bar{x})=\underline{\text{true}})\times\text{body}(\bar{x})\times W(\text{body}(\bar{x}).\bar{y}))\ .$$

We use the notation $\text{body}(\bar{x}).\bar{y}$ to denote the sequence of values, say $y_1,\ldots,y_m$, $m\leq n$ which arise from the body and are used as input values on iteration of the loop. For simplicity we assume that $m=n$ and all "input" parameters correspond to "outputs" from the loop.

Suppose the goal G can be expressed as

$$I(\mathrm{last}(W(\bar{x})).\bar{y}) \;\&\; \neg \mathrm{bexp}(\mathrm{last}(W(\bar{x})).\bar{y}) \;,$$

and suppose that the loop body maps a proof of $I(\bar{x})$ to a proof of $I(\mathrm{body}(\bar{x}).\bar{y})$. More specifically, assume that x_n is of type $I(x_1,\ldots,x_{n-1})$, that is x_n is a proof of the proposition $I(x_1,\ldots,x_{n-1})$; and y_n is of type $I(y_1,\ldots,y_{n-1})$. Then the loop computation builds a proof of the goal inductively. We can describe a $W(\bar{x})$ recursive function which defines the proof that follows.

$$\mathrm{proof}(w(x),x) = \begin{cases} x_n \times \langle \neg \mathrm{bexp}(\bar{x}) \rangle & \text{if } \mathrm{casel}(w(\bar{x})) \\ & \text{(i.e., if } \neg \mathrm{bexp}(\bar{x})) \\ \mathrm{proof}(w'(\bar{x}),\mathrm{body}(\bar{x}).\bar{y}) & \\ & \text{(i.e., if } \mathrm{bexp}(\bar{x})) \end{cases}$$

where x_n is the proof of $I(x_1,\ldots,x_{n-1})$ and where $w'(\bar{x}) \in W(\mathrm{body}(\bar{x}).y)$.

In the PL/CV2 system, the proposition I is called a <u>loop invariant</u> and the input of a proof of this type to the loop body is designated by the key word <u>assume</u>. The loop is written

<u>while</u>(bexp) <u>do</u> <u>assume</u> I; ...; I <u>od</u>

and the explanation of the loop rule combines ideas which in the type system described here are distinct. First, the loop rule describes the inductive construction of the proof by requiring that I be provable at the end of the loop body from the assumption of

I at the beginning. Second, the rule describes the type correctness of the entire loop body, that is, the claim that the loop construct is a legitimate recursive type definition.

4.4 LOOP TERMINATION

The PL/CV2 system also includes a rule of loop termination to show that the computation described inductively is nonempty. So far, we have not discussed methods of proving this in the type theory. In the case of unparameterized recursive type definitions, such as

$$T = N + T \times T$$

the type T is shown to be nonempty by constructing an element of it, e.g., $inj_{N \to T}(2) \varepsilon T$.

In the case of a recursive type definition $T(x)$ parameterized by N , or by any inductive type M , a proof that $T(x)$ is nonempty for $x \varepsilon M$ proceeds by M-induction. For loops this rule takes the form of defining a function

$$M \to W(m,\bar{x}) \quad \text{for} \quad m \varepsilon M$$

by M-recursion.

It is convenient to allow special forms to display such functions. The termination rule for loops in PL/CV is just such a special form. The N-induction is displayed right in the definition of the loop as follows

```
1.  ∃n∈N. T(n,x̄);
    while bexp
      do
2.      arbitrary n∈N where T(n,x̄);
3.      ¬T(0/n,x̄);
         .
         .
         .
4.       T(n-1/n,x̄)
      od
```

Line 1 asserts that integer n is in relation T to the parameters of the loop $\bar{x}$. Line 2 asserts that n is an arbitrary parameter to the loop satisfying T, and line 3 asserts that when the parameters $\bar{x}$ are such that bexp is true, then the parameter n is nonzero. Finally, line 4 shows that on the next copy of the body in the recursive type, a smaller integer parameter can be used.

This form of N-induction embedded in the loop definition may not be the most general and ideal form, but it illustrates a way of embedding induction in the type definition, and it is the way currently used in PL/CV3 for all primitive recursive types, i.e., those of the form $T = B+R(T)$ for R a type expression in T, B the base type.

§5 CONCLUSION

The simple examples of §4 illustrate how rules for reasoning about fairly complex program constructs, such as procedures with type parameters [Demers, Donahue 80] and loops which use multi-set orderings for termination, [Dershowitz, Manna 78], etc. can be derived from the rules for types. Moreover, the rules for combining programs (deterministic and nondeterministic) and propositions as in algorithmic logic, dynamic logic and programming logic can be derived from the rules for types. Since any modern high level programming language will have a rich type structure, as rich as that in Algol 68, ADA, or Russell, a logic for reasoning about such programs will have rules for types roughly comparable to the rules we need for the type theory outlined here. Thus, we have shown that a constructive theory of types such as the one outlined here is necessary and sufficient as a logic of programs. Comparison of our axiomatizations of elementary programming logics [Igaraski, London, Luckham 75, Constable, O'Donnell 78] illustrates the power of abstraction in formal theories because the same number of axioms (about 40) suffices to define both theories, yet the type theory is vastly richer than the elementary theory. Indeed, since the type theory is closely related to Martin-Löf's ITT [Martin-Löf 73], and since ITT appears adequate to formalize constructive mathematics, this theory appears adequate to formalize all computational mathematics. Since our constructive type theory was motivated largely by the needs of modern programming languages, these results suggest that theories adequate for high level programming will also be adequate for constructive mathematics. This observation supports Martin-Löf's belief that the search for logically ever more satisfactory high level programming languages can stop short of anything but a language in which constructive mathematics can be adequately expressed [Martin-Löf 79].

ACKNOWLEDGEMENTS

I would like to thank Daniel Zlatin for his assistance in explicating these ideas. Together we are working on the PL/CV3 version of the theory described here. I also want to thank Michele Fish for her careful preparation of the manuscript.

References

1. Banachowski, L., A. Kreczmar, G. Mirkowska, H. Rasiowa, and A. Salwicki, An Introduction to Algorithmic Logic, Banach Center Publications, Vol. 2, Warsaw, 1977.

2. Constable, R.L., M.J. O'Donnell, A Programming Logic, Winthrop, Cambridge, 1978.

3. Constable, Robert L., "FORMAL: a family of languages for communicating formal reasoning," Computer Science Department, Cornell University, 1980.

4. Constable, Robert L., "On the Theory of Programming Logics," Proceedings of the 9th Annual ACM Symposium on Theory of Computing, Boulder, Colorado, May, 1977, 269-285.

5. de Bruijn, N.G., "The mathematical language AUTOMATH, its usage and, some of its extensions," Symposium on Automatic Demonstration Lecture Notes in Math, Vol. 125, Springer-Verlag, New York, 1970, 29-61.

6. Demers, A.J., J. Donahue, "Parameters and Type Checking," Proc. of Seventh ACM Symp. on Principles of Programming Languages, January, 1980.

7. Demers, A.J., J. Donahue, "Revised Report on Russell," Dept. of Computer Science Tech. Report TR 79-389, Cornell University, September, 1979.

8. Derschowitz, N. and Zohar Manna, "Proving Termination with Multiset Orderings," Comm. ACM, Vol. 22, No. 8, 465-476.

9. Downey, P., R. Tarjan, R. Sethi, "Variations on the Common Subexpression Problem," to appear in JACM.

10. Engeler, E., Formal Languages: Automata and Structures, Markham, Chicago, 1968.

11. Gordon, M., R. Milner, C. Wadsworth, "Edinburgh LCF, A Mechanized Logic of Computation," Lecture Notes in Computer Science, Springer-Verlag, 1979.

12. Harel, D., "First-Order Dynamic Logic," Lecture Notes in Computer Science, 68, Springer-Verlag, New York, 1979.

13. Igarashi, S., R.L. London, D.C. Luckham, "Automatic Program Verification I: A Logical Basis and its Implementation," Acta Informatica 4, 1975, 145-182.

14. Kennedy, K., J.T. Schwartz, "An Introduction to the Set Theoretical Language SETL," Comp. & Math. with Applications, Vol. 1, 1975, 97-119 (Pergamon Press).

15. Kleene, S.C., Introduction to Metamathematics, D. Van Nostrand, Princeton, 1952, 550 pp.

16. Kozen, D.C., Complexity of Finitely Presented Algebras, Ph.D. Thesis, Cornell University, Computer Science Department, 1977.

17. Lewis, C.H., B.K. Rosen, "Recursively Defined Data Types, Part 1," ACM Symposium on Principles of Programming Languages, Boston, October, 1973, 125-138.

18. Manna, Z., A. Pneuli, "The Modal Logic of Programs," Proc. of the Sixth Int. Colloq. on Automata, Languages and Programs, Graz, Austria, July, 1979.

19. Manna, Z., Six Lectures on the Logic of Computer Programming, SIAM, 1980.

20. Martin-Löf, Per, "Constructive Mathematics and Computer Programming," 6th International Congress for Logic, Method and Phil. of Science, Hannover, Aug., 1979.

21. Martin-Löf, Per, "An Intuitionisitc Theory of Types: Predicative Part," Logic Colloquium '73, ed. H.E. Rose, J.C. Shepherdson, North-Holland, Amsterdam, 1975, 73-118.

22. Milne, R., C. Strachey, "A Theory of Programming Language Semantics," Halsted Press, New York, 1976.

23. Mirkowska, G., "On Formalized Systems of Algorithmic Logic," Bull. Acad. Pol. Sci., Ser. Math. Astrong. Phys., 18 (1971) 421-428.

24. Oppen, D.C., "Reasoning about recursively defined data structures," Conf. Record of 5th Annual ACM Symposium on Principles of Programming Languages, ACM, New York, January, 1978, 151-157.

25. Pratt, V.R., "Semantical Considerations on Floyd-Hoare Logic," Proc. of 17th Annual Symp. on Foundations of Computer Science, Houston, October, 1976, 109-121.

26. Rasiowa, H., Algorithmic Logic, ICS PAS Reports 281, Institute of Computer Science, Polish Academy of Science, Warsaw, Poland, 1977.

27. Scott, Dana, "Data Types as Lattices," SIAM Journal on Computing, 5,3, September, 1976.

28. Scott, Dana, "Constructive Validity," Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125, Springer-Verlag, 1970, 237-275.

29. Scott, Dana, C. Strachey, "Toward a Mathematical Semantics for Computer Languages," Proc. of the Symposium on Computers and Automata, Polytech Institute of Brooklyn, New York, 1971, 19-46.

30. Shamir, A., W.W. Wadge, "Data Types as Objects," Automata, Languages, and Programming, 4th Colloquium, Lecture Notes in Computer Science No. 52, Turku, 1977, 465-479.

31. Solomon, M.H., Theoretical Issues in the Implementation of Programming Languages, Ph.D. Thesis, Cornell University, May, 1977.

32. Stoy, J.E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, the MIT Press, Cambridge, 1977, 414 pp.

33. Stenlund, S., Combinators, λ-terms, and Proof-Theory, D. Reidel, Dordrecht, 1972, 183 pp.

34. Troelstra, A.S., "Metamathematical Investigations of Intuitionistic Arithmetic and Analysis," Lecture Notes in Mathematics, 344, Springer-Verlag, New York, 1973.

35. van Wijngaarden, A.B.J., et.al., "Revised Report on the Algorithmic Language ALGO 68," Supplement to ALGO BULLETIN, Univ. of Alberta, 1974.

36. Constable, R.L., "A Constructive Theory of Recursive Functions," Computer Science Technical Report, TR 73-186, Cornell University, 1973, 66 p.

37. Constable, R.L., "The Fundamental Theorem of Arithmetic in PL/CV2," Computer Science Technical Report, TR 80-424, 1980.