8189 COPY 2

The Varieties of Programming Language

by

Christopher Strachey

Technical Monograph PRG-10

March, 1973.

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford.

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford. OX2 6PE.

ABSTRACT

This paper suggests an analysis of the domains used in programming languages. It identifies some of the characteristic domains and shows that programming languages vary widely in their definition of these domains.

This monograph is a revised and slightly expanded version of a paper given at the International Computing Symposium at Venice 12-14 April, 1972. (see reference [10]).

PREFACE

(With apologies to Professor William James, Miss Stella Gibbons and the late Herr Baedeker.)

**In my belief that a large acquaintance with particulars often makes us wiser than the mere possession of abstract formulas, however deep, I have ended this paper with some concrete examples, and I have chosen these among the extreme designs of programming languages. To some readers I may consequently seem, by the time they reach the end of the paper, to offer a caricature of the subject. Such convulsions of linguistic purity, they will say, are not sane. It is my belief, however, that there is much of value to be learnt from the study of extreme examples, not least, perhaps, that our view of sanity is rather easily influenced by our environment; and this, in the case of programming languages, is only too often narrowly confined to a single machine. My ambition in this and other related papers, mostly so far unwritten, is to develop an understanding of the mathematical ideals of programming languages and to combine them with other principles of common sense which serve as correctives of exaggeration, allowing the individual reader to draw as moderate conclusions as he will.

CONTENTS

THE VARIETIES OF PROGRAMMING LANGUAGE

0. INTRODUCTION

There are so many programming languages in existence that it is a hopeless task to attempt to learn them all. Moreover many programming languages are very badly described; in some cases the syntax, or rather *most* of the syntax, is clearly and concisely defined, but the vitally important question of the semantics is almost always dealt with inadequately.

Part of the reason for this is that there is no generally accepted formalism in which to describe the semantics; there is nothing for semantics corresponding to BNF for syntax. BNF is far from adequate to describe the whole syntax of any programming language, but with a little goodwill and a few informal extensions here and there, it is enough to be of considerable help in describing a large part of the syntax of many languages. Moreover, and this is one of its chief advantages, it is very widely understood and used by programming language designers and implementers.

When we come to the semantics the situation is not nearly so satisfactory. Not only is there no generally accepted notation, there is very little agreement even about the use of words. The trouble seems to be that programming language designers often have a rather parochial outlook and appear not to be aware of the range of semantic possibilities for programming languages. As a consequence they never explain explicitly some of the most important features of a programming language and the decisions among these, though rarely mentioned (and frequently I suspect made unconsciously), have a very important effect on the general flavour of the language. The main purpose of this paper is to discuss some features of this range of semantic possibilities in the hope that it may make it easier to classify the varieties of programming language.

One of the greatest advantages of using a high level programming language is that it allows us to think about abstract mathematical objects, such as integers, instead of their rather arbitrary representation by bit-patterns inside the machine. When we write programs we can now think of *variables* instead of *locations* (or *addresses*) and *functions* instead of *subroutines*. When we write a statement such as

$$x := Sin(y+3)$$

in Algol 60, what we have in mind is the mathematical functions sine and addition. It is true that our machines can only provide an *approximation* to these functions but the discrepancies are generally small and we usually start by ignoring them. It is only after we have devised a program which would be correct if the functions used were the exact mathematical ones that we can start investigating the errors caused by the finite nature of our computer.

This is the "mathematical" approach to writing programs; we are chiefly interested in the *values* of the expressions and not in the steps by which they are obtained. The alternative, earlier approach, which might be called "operational", involves specifying in detail the sequence of steps by which the result can be obtained. While the ability to do this is also an important facet of computing, it should be regarded as a means to an end; the important thing is to compute the correct quantity. It is generally much easier to prove that one particular program provides an approximation to a mathematically exact function than it is to prove the approximate equivalence of two programs directly. As there are usually several possible ways of implementing any particular function, it is obviously more satisfactory to specify the ideal mathematical function as the first step and then, as a second to consider the implementation and the approximation it introduces. All this is widely appreciated by numerical analysts and programmers and it accounts, at least in part, for the popularity of high level languages.

When it comes to the description of programming languages themselves, however, the situation is quite different. Most of the work on syntax and some on semantics has been at the level of symbol manipulation - that is to say it has been concerned with the representatio (generally on paper)'rather than with the mathematical objects represented. The unsatisfactory nature of our understanding of programming languages is shown up by the fact that although the subject is clearly a branch of mathematics, we still have virtually no theorems of general application and remarkably few specialised results.

A second purpose of this paper is to advocate a more conventionally mathematical approach to the problem of describing a programming language and defining its semantics, and, indeed, to the problems of computation generally.

1. MATHEMATICAL BASIS

In our search for a mathematical approach to semantics we shall make great use of functions of various sorts. Many of these will be of "higher type" - i.e. will have other functions as their arguments or results. The word "functional" is sometimes used for functions which operate on functions, but we prefer to use the single word "function" for all types.

1.1 *Functions*

In order to specify a function mathematically we need to give it a *domain* (the set in which its arguments lie) and *range* (the set of its function values) as well as its *graph* (the set of ordered pairs of arguments and function values). The domain and range specify the *functionality* of the function - i.e. the set of which it is a member. The graph, which is often given by an expression or algorithm, identifies the particular member of this set.

The functionality of a function is often taken for granted or glossed over when it is defined. This may be unobjectionable as the functionality can sometimes be deduced unambiguously from the expression for its graph. There are, however, cases in which a more rigorous investigation shows up difficulties and confusions of considerable importance. We shall therefore look rather carefully at the domain and range of the more important functions which occur in the interpretation of a programming language. Before considering specific examples we need to discuss the general features of domains and ranges. (As these are similar we shall use the word *domain* for both domains and ranges.)

1.2 *Domains*

There are two main classes of domain: *elementary* and *compound*. The elementary domains correspond to the familiar simple types in programming languages: their properties are in general mathematical in nature and independent of the design choices of a programming language. Some common elementary domains are:

T	Truth values (Booleans)
N	Integers
R	Reals
Q	Character strings (Quotations)

These can all be considered as "data types" to be manipulated by the programming language and their properties are the usual mathematical ones. The various expressions in the programming language may have values which lie in one or other of these domains. The domain Q is generally confined to those character strings which are manipulated by the program; the text of the program itself is not one of these and so is not regarded as lying in Q.

Compound domains are constructed from other domains, either elementary or compound, by one of three methods. If D_0 and D_1 are any two domains (not necessarily different) we write

$D_0 + D_1$ for their *sum*

$D_0 \times D_1$ for their *product*

An element of $D_0 + D_1$ is *either* an element of D_0 *or* an element of D_1 (but not both).

An element of $D_0 \times D_1$ is an ordered pair whose first component is an element of D_0 and whose second component is an element of D_1.

Sums and products of domains can be extended to more than two components without any serious difficulty and we shall write expressions such as $D_0 + D_1 + D_2$ and $D_0 + D_1 + [D_0 \times D_1]$ in a rather informal manner to represent such domains.

There are two notational abbreviations for sums and products which are sometimes convenient. We write

D^n for $D \times D \times D \times \ldots \times D$ (n factors)

and D^* for $D^1 + D^2 + D^3 + \ldots$

Examples

$N + R$ is the domain of numbers which may be either integers or reals. The arithmetic operators in programming languages often use this domain.

$R \times R$ is the domain of ordered pairs of reals. A complex number, for example, can be represented by an element in this domain.

R^6 is the domain of sixtuples all of whose components are reals. A real vector of dimension 6 might be an element of this domain.

R^* is the domain of all real vectors of any dimension.

The third method of combining the two domains D_0 and D_1 is to form those functions whose argument is in D_0 and whose result is in D_1; we shall write $D_0 \to D_1$ to represent this domain. It will appear in the next section that we are not interested in *all* the set-theoretic functions from D_0 to D_1 - only in those which are in some sense "reasonable". But nor are we only interested in *total* functions - it may well be that for some values of its arguments our functions "fail to converge" - i.e. are undefined. We shall have more to say about functional domains in the next section. The general topic of domains for programming languages is discussed further in several papers cited in the references [1,3,5].

1.3 *Reflexive Domains*

When we come to examine the domains required by programming languages later in the paper we shall often want to define a domain *self-referentially*. For example in a list-processing language we might want to discuss the domain of single level lists of atoms. Following LISP we could define a list as an ordered pair, the first component of which was always an atom, while the second was another list. Thus if A is the domain of atoms and D is the domain of single level lists, we should have an equation

$$D = A \times D$$

If we wanted to allow lists as well as atoms as the first component (so that our domain was of list-structures, not merely single level lists) the defining equation would become only very little more complicated.

$$D' = [A + D'] \times D'$$

These equations are reminiscent of the recursive structure definitions in some programming languages.

Another example is the type-free λ-calculus of Church and Curry. If we allow atoms as well as λ-expressions every object must be either an atom or a function (as every λ-expression is considered to be a function). This leads to the defining equation.

$$D = A + [D \to D]$$

This looks much like our previous definitions but it conceals a serious difficulty. *If we take* $D \to D$ *to be the set of all set-theoretic functions from* D *to itself, there is no solution to the equation.* In fact there are always more elements of $D \to D$ than there are of D so that the equation (which should be interpreted as "up to an isomorphism" only) cannot be satisfied.

If D has k members, the full set $D \to D$ has k^k members and, as Cantor's theorem shows, this is always greater than k provided k is greater than one.

At first sight this looks like a fatal flaw and it certainly demonstrates vividly the mathematical danger in failing to prove the existence of the objects we wish to discuss. We cannot now feel happy with any of our domains defined by a self-referential equation (we shall refer to these as *reflexive domains*) until we have proved their existence. It was the impossibility of doing this for definitions involving $D \to D$ that prevented the construction of set-theoretic models of the λ-calculus and forced it to remain a purely formal theory.

Fortunately, however, in 1969 Dana Scott discovered a solution to this problem. As we indicated in the last section the difficulty of cardinality is avoided by restricting the domain $D_0 \to D_1$ to include only some of the set-theoretic functions, though the restriction appears quite naturally as a consequence of the theory. In outline Scott argues as follows:

It is reasonable to adjoin to every domain a partial ordering based intuitively on a *degree of approximation* or *information content*. For many elementary domains this partial ordering is of a rather trivial kind, but it is sufficient to turn the domains into complete lattices. The partial ordering for a compound domain can be derived from those of its components.

If a function is to be well behaved, it should be monotonic, i.e. preserve the partial ordering - it should not be possible to get a better defined (more accurate) result by giving it a worse defined argument. If a function is to be computable, it should be possible to obtain any degree of approximation to its result by giving a sufficient, but still finite, amount of information about its arguments. This means that the function should preserve limits in some way. These two conditions lead to the idea that we should be concerned only with *continuous* functions (the term is, of course, precisely defined) and this is the restriction imposed on the construction $D_0 \to D_1$. The exact mathematical nature of continuous functions is discussed elsewhere; it is sufficient to say here that they include all the ordinary and reasonable sorts of function - and, indeed, all those which are computable - and exclude only those which have mathematically pathological properties.

Making use of these ideas Scott was then able to construct, by a method which is reminiscent of that of Dedekind's cuts for constructing real numbers, a reflexive domain which satisfied the equation $D = D \to D$, thus producing the first set-theoretic model for the λ-calculus.

In a further extremely elegant piece of work he proved the existence of a universal domain U which satisfies the equation

$$U = A + [U + U] + [U \times U] + [U \to U]$$

where A is any domain. The domain U proves to be extremely rich in sub-domains and Scott was able to show that these include all the reflexive domains which can be defined by a self-referential equation using A (or its components) and the domain constructing operators +, ×, and →.

This is not the place to go into the details of this work and the interested reader is referred to the papers by Scott[1,2,3]. We can procede with our analysis of the characteristic functions of programming languages secure in the knowledge that all the reflexive domains we require, no matter how complicated, do have a mathematical existence.

2. CHARACTERISTIC DOMAINS IN PROGRAMMING LANGUAGES

2.1 *Denotations*

Programming languages follow the example of mathematics generally in allowing names chosen by the user to stand for or *denote* certain objects. The relationship between the name and the thing it denotes is, of course, a function in the mathematical sense; we shall call it the *environment* and reserve the Greek letter ρ to stand for an individual environment. The functional nature of the environment, which we shall write as Id → D, varies widely from one programming language to another and is well worth closer study.

The domain (i.e. the set of arguments of ρ), which we wrote as Id, is the set of *names* or *identifiers* in the programming language. In the sense of §1.2 above, Id is an elementary domain, and it is also the only domain we shall encounter whose members are elements of the text of a program, and are therefore parts of the programming language itself and not the objects manipulated by it. Id is generally defined by the syntax rules of the programming language. It is a very simple domain whose only basic property is the relation of equality - it is always possible to say if two names are the same or not, but names in general have no internal structure.

The only remarkable thing about Id is the number of different words which have been used to describe its members. The fact that they have been called "names", "identifiers" and "variables" in different languages would not matter so much if these same words had not been also used with quite different meanings. I have preferred to use the word "name" or sometimes "identifier" for a member of Id as I think this accords best with the non-technical use of the word, but the reader should be warned that both Algol 60 and Algol 68 use the word "name" in quite different senses. Algol 60 uses the term "call by name" where "call by substitution" would be more appropriate; Algol 68 uses the word "name" in an equally incorrect and even more misleading manner to mean approximately what is generally meant by the word "address".

The range of ρ - i.e. those things which can be given names - will be written as D. (In the earlier parts of this paper D has been used as a general symbol for any domain. In what follows it will be reserved for the domain of *denotations*). In many languages D is a large compound domain with many component parts. It must include anything which can be passed to a procedure as a parameter (as inside the procedure this will be denoted by the corresponding formal parameter) as well as the objects declared as local or global variables. Thus in Algol 60, D must include procedures (and type-procedures), labels, arrays and strings.

The domain D does not, however, include integers, reals or booleans. The reason for this is that we want to preserve the static nature of ρ. In ordinary mathematics, the thing denoted by a name remains constant inside the lexicographical scope of the name; it is determined only by context and not by history. In programming languages this is also true of some names - for example procedures in Algol 60; once declared they keep the same meaning (denotation) throughout their scope. On the other hand, for names declared in Algol 60 as **real**, **integer** or **boolean**, it is possihle to change the value associated with the name by an assignment command. For names such as these, the associated integer, real or boolean value can only be obtained dynamically and depends on the current state of the store of the machine. In spite of this, however, the address in the store associated with the name remains constant - it is only the contents of this address which are altered by the assignment command. It is therefore appropriate to say that the name denotes a *location* which remains fixed, and that the ordinary value associated with the name is the content of this location. The location is sometimes known as the *L-value* of the name and its content is called the *R-value*. The concepts of an L-value, which is a location, and the corresponding R-value which is its content, can be extended to cover expressions (for example array elements) as well as names.

We therefore need an elementary domain of locations; which we shall call L, and it must be a component part of D for all languages which have an assignment command.

2.2 *Stored Values*

The state of the machine, and in particular the state of its store, determines the contents of the locations. We shall use the Greek letter σ to stand for a machine state and S to stand for the set of all possible machine states. We shall not discuss the nature of the domain S exhaustively - it seems probable that it may vary from language to language - but it must always contain at least enough information to give the contents of all the locations in use, it must therefore include a component with functionality $L \rightarrow V$ where V is the domain of all *stored values* - i.e. those quantities which can be the value of the right hand side of an assignment command.

The two functions ρ and σ together with their associated domains D and V go a long way to characterising a programming language. There is a fundamental difference between these two functions which is the source of many of the confusions and difficulties both about programming languages and also about operating systems. This is that while the environment ρ behaves in a typically "mathematical" way - several environments can exist at the same point in a program, and on leaving one environment it is often possible to go back to a previous one - the machine state σ which includes the contents function for the store, behaves in a typically "operational" way. The state transformation produced by obeying a command is essentially irreversible and it is, by the nature of the computers we use, impossible to have more than one version of σ available at any one time. It is this contrast between the static, nesting, permanent environment, ρ, and the dynamic irreversibly changing machine state, σ, which makes programming languages so much more complicated than conventional mathematics in which the assignment statement, and hence the need for σ, is absent.

2.3 *The Assignment Command*

We can now give a model for an abstract store and explain the meaning of the assignment command in terms of it. The model is deliberately simplified and the explanation informal. Most existing programming languages need a more complicated model and a more formalised description of the assignment command is necessary before much more detailed work on semantics can be carried out. Many of these developments have been, or are in the process of being, worked out [4,6] but for the present paper it seems better to avoid as much detail as possible, and to give only the main outlines.

The simple model of the store contains the following domains

L	Locations	$\alpha, \alpha' \in L$
V	Storable Values	$\beta \in V$
$S = [L \to V]$	Stores	$\sigma, \sigma' \in S$

We postulate the following basic functions on these domains. In each case we first give the functionality (i.e. range and domain of the basic function) and then an expression which defines it.

(i) $\mathit{Contents} : L \to [S \to V]$

$\mathit{Contents}(\alpha)(\sigma) = \sigma(\alpha)$

(ii) $\mathit{Update} : [L \times V] \to [S \to S]$

If $\sigma' = \mathit{Update}(\alpha,\beta)(\sigma)$

$\mathit{Contents}(\alpha)(\sigma') = \beta$

and $\mathit{Contents}(\alpha')(\sigma') = \mathit{Contents}(\alpha')(\sigma)$ if $\alpha' \neq \alpha$

Thus the effect of updating a location α in a store σ with a value β is to produce a new store σ' which yields the contents β for the location α but is everywhere else identical with σ. This, of course, is exactly what we expect a simple update operation to do. A point to notice is that the partially applied update function $\mathit{Update}(\alpha,\beta)$ is of type $[S \to S]$- i.e. a function that transforms (alters) the store.

Before we can deal with the assignment command we need to introduce functions which yield the values of expressions. Since expressions (which include names as a special case) in general have both L-values and R-values we need two such functions, which we shall write as $\mathcal{L}$ and $\mathcal{R}$. These functions operate on expressions in the programming language and their results clearly depend on the environment, ρ, to provide a denotation for the names in the expression; it is also fairly obvious that their results may depend on the state of the store, σ, as well. When we consider what results they should yield, we must remember that there is a possibility that the evaluation of an expression may have a side effect - i.e. it may alter the store as well as producing a result. This implies that the results of our evaluation functions should be pairs consisting of a value and a possibly altered store.

To express these ideas in symbols we need two new domains

Exp	Expressions in the Programming Language	$\varepsilon_0, \varepsilon_1 \in \mathrm{Exp}$
$\mathrm{Env} = [\mathrm{Id} \to D]$	Environments	$\rho \in \mathrm{Env}$

Then we have the basic functions

$\mathcal{L} : \mathrm{Exp} \to [\mathrm{Env} \to [S \to [L \times S]]]$

and $\mathcal{R} : \mathrm{Exp} \to [\mathrm{Env} \to [S \to [V \times S]]]$

The detailed definitions of $\mathcal{L}$ and $\mathcal{R}$ form part of the semantic description of the programming language, and we shall not consider them further here.

We can now consider the effect of a general assignment command of the form

$$\varepsilon_0 := \varepsilon_1$$

(Note that the left side of this is an expression, ε_0, although most programming languages limit rather severely the sorts of expression that may be used here.)

The operation takes place in three steps

1. Find the L-value of ε_0
2. Find the R-value of ε_1
3. Do the updating.

(Note: we have assumed a left-to-right order for the evaluations.)

If we are obeying this command in an environment ρ with an initial store σ_0, these three steps can be written symbolically as

1. $\mathcal{L}(\varepsilon_0)(\rho)(\sigma_0) = \langle\alpha,\sigma_1\rangle$

α is the L-value of ε_0, σ_1 is the store which may have been altered while finding α; if there are no side effects, $\sigma_1 = \sigma_0$.

2. $\mathcal{R}(\varepsilon_1)(\rho)(\sigma_1) = \langle\beta,\sigma_2\rangle$

β is the R-value of ε_1. Note the use of σ_1 in place of σ_0; it is this that expresses the left-to-right order of evaluation.

3. $\mathit{Update}(\alpha,\beta)(\sigma_2) = \sigma_3$

Then the effect of the whole command is to change σ_0 into σ_3.

We can now introduce another semantic function $\mathcal{C}$ which gives the meaning of commands. The functionality of $\mathcal{C}$ will be

$$\mathcal{C} : \mathrm{Cmd} \to [\mathrm{Env} \to [S \to S]]$$

where Cmd is the domain of commands in the programming language. In terms of $\mathcal{C}$ we can write

$$\mathcal{C}(\varepsilon_0 := \varepsilon_1)(\rho) = \theta$$

where $\theta \in S \to S$ and, for the example above,

$$\theta(\sigma_0) = \sigma_3.$$

3. TWO SPECIMEN LANGUAGES

In order to make the ideas discussed above more concrete we give below a discussion of two programming languages in terms of their domains. The first, Algol 60, is probably familiar to most readers; the second, PAL, is unlikely to be known by many. For both languages we start by listing (and if necessary discussing) the elementary domains; we then define and discuss various derived compound domains which occur naturally in the description of the language and finally give the composition of the characteristic domains D and V.

3.1 *Algol 60*.

a. Elementary Domains

T	Booleans (truth values)
N	Integers
R	Reals
Q	String (quotations)
J	Labels (jump points)
L	Locations
S	Stores (machines states)

T, N, and R have their ordinary mathematical properties. Algol 60 has no basic operations on Q, but strings may be passed as parameters. We treat J and S as elementary domains because we do not want to investigate their structure. We note that S at least includes $L \to V$.

b. Derived Domains

Expression values

$$E = D + V$$

E must contain D because a name by itself is a simple form of expression.

Procedures

$$P = [D^* \to [S \to S]] \\ +[D^* \to [S \to [V \times S]]]$$

The parameters must lie in D as they are denoted by formal parameters inside the body of the procedure; their number is unspecified so that the parameter list is a member of

$D^* = D^0 + D^1 + D^2 \ldots$. The body of an ordinary (non-type) procedure is a command - i.e. it transforms the state of the machine and so is in $[S \to S]$. A type procedure produces a result which, perhaps by chance, in Algol 60 lies in V (not in E) but the procedure itself may also have a side effect and alter the store. Thus its functionality must be $[S \to [V \times S]]$.

Arrays

The elements of an array can be assigned to and must therefore denote locations .

$$A1 = L + L^2 + L^3 + \ldots \qquad \text{(vectors)}$$
$$= L^*$$
$$A2 = A1 + A1^2 + \ldots \qquad \text{(matrices)}$$
$$= A1^* = L^{**}$$
$$A3 = A2^* = L^{***} \qquad \text{(3-arrays)}$$

- - - - - - - - - -

$$A = L^* + L^{**} + L^{***} + \ldots . \qquad \text{(all arrays)}$$
$$= L^{*^*}$$

Calls by Name

$$W = S \to [E \times S]$$

Formal parameters called by name are rather like type-procedures (functions) with no parameters; they produce a value and may have a side-effect and so alter the store. The value they produce, however, is not confined to V but may be anywhere in E.

c. Characteristic Domains

Denotations

$D = L$	(booleans, integers, reals)
$+ P$	(procedures, type-procedures)
$+ L^{*^*}$	(arrays)
$+ W$	(calls by name)
$+ W^*$	(switches)
$+ Q$	(strings)
$+ J$	(labels)

where $P = [D^* \to [S \to S]] + [D^* \to [S \to [V \times S]]]$
and $W = S \to [E \times S]$

Stored values

$V = T + N + R$

Note that V is rather small and D very large.

3.2 *PAL*

PAL is a language developed by Evans and Wozencraft [7] at MIT for teaching purposes. It is an unusually "clean" language but difficult to implement efficiently. It resembles Gedanken of Reynolds [8] and Euler of Wirth [9] in the fact that its type checking and coercion, if any, are done dynamically at run-time and not at compile time.

a. Elementary Domains

These are the same as for Algol 60 viz: T,N,R,Q,J,L and S. The jumps in PAL are considerably more powerful than those in Algol 60, so that J in PAL is different from J in Algol 60; PAL also has some basic operators on Q.

b. Derived Domains

Expression values

$E = D + V$

This domain is hardly needed in PAL.

Procedures

$P = L \to [S \to [L \times S]]$

There is only one sort of procedure (or function) in PAL. This takes a single location (L-value) as an argument and produces a single location as a result, also perhaps altering the state of the machine as a side-effect. The effect of several arguments can be obtained by handing over a single list of arguments (a tuple as defined below); a pure procedure, which yields no useful result and is used, like a command, merely to alter the machine state, is given a dummy result.

Tuples

These are the only structural values in PAL; they take the place of arrays in Algol 60. A tuple is a vector of locations and is therefore a member of L^*.

c. Characteristic Domains

Denotations

$D = L$

All names can be assigned to, and so denote locations.

Stored values

$V = T + N + R$	(booleans, integers, reals)
$+ Q + J$	(strings, labels)
$+ L^*$	(tuples)
$+ P$	(procedures)
$+ \{dummy\}$	

All the values in PAL (except a single location) can be stored and so are part of V. Note that L is not itself a member of L^* - in that a 1-tuple is distinguishable from a location. In fact a 1-tuple is an ordinary R-value and can be assigned or stored.

Note that in contrast to Algol 60, D in PAL is very small and V very large.

4. CONCLUSION

The differences between the domain structure of Algol 60 and PAL are very striking. They lie, moreover, at a rather deep level and do not depend in any way on the syntax or even the range of basic semantic operations in the language. They are in some sense *structural*. It is clear that there are many important features of a programming language which cannot be revealed in any analysis as general as this; there are also some further structural features which are not made evident by a study of the domains. (An example of this is the different way in which Algol 60 and PAL deal with type checking and coercion.) In spite of this inevitable incompleteness, I think it would be well worth the effort of any language designer to start with a consideration of the domain structure.

The general idea of investigating the domain structure of a programming language grew from a collaboration between the author and Dana Scott which started in the autumn of 1969. Our main objective was to produce a mathematical theory of the semantics of programming languages. A general outline of this work is given in Scott [1]; Scott and Strachey [4] gives an introduction to the theory of the mathematical semantics based on these ideas. Other papers in print [2,3] and in preparation [5,6] give further details. Much still remains to be done before we have a reasonably complete theory and we hope to continue our work along these lines.

REFERENCES

[1]* Dana Scott, *An Outline of a Mathematical Theory of Computation*, Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems (1970), pp.169-176.

[2]* Dana Scott, *The Lattice of Flow Diagrams*, Springer Lecture notes in Mathematics, vol. 188, (1971), pp. 311-366.

[3]* Dana Scott, *Continuous Lattices*, Proc. Dalhousie Conference, Springer Lecture Notes (in press).

[4]* Dana Scott and Christopher Strachey, *Toward a Mathematical Semantics for Computer Languages*, Proc. Symposium on Computers and Automata. Microwave Inst. Symposia Series 21, Polytechnic Institute of Brooklyn.

[5]* Dana Scott and Christopher Strachey, *Data Types as Lattices*, (in preparation).

[6]* Christopher Strachey, *An Abstract Model for Storage* (in preparation).

[7] A. Evans, Jr., *PAL - a Language for Teaching Programming Linguistics*. Proc. ACM 23rd National Con., Brandon/ Systems, Princeton, N.J.

[8] J. C. Reynolds, *Gedanken - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept*, Comm. ACM 13, (1970), pp. 308-319.

[9] N. Wirth and H. Weber, *EULER: A Generalisation of Algol, and its formal definition*. Comm. ACM 9. (1966), pp. 13-24 and 89-99.

[10] Christopher Strachey, *Varieties of Programming Language*, Proc. International Computing Symposium, Cini Foundation Venice (1972) pp. 222-233.

** William James, *The Varieties of Religious Experience*, (Preface), Longmans & Co., London; Cambridge, Mass. (1902)

Stella Gibbons, *Cold Comfort Farm*, (Preface), Longmans & Co., London. (1932)

* Also published as a Programming Research Group Technical Monograph.