P. Lucas

Formal Semantics of Programming Languages: VDL

The history of ideas that led to the first formalization of the syntax and semantics of PL/I is sketched. The definition method and notation are known as the Vienna Definition Language (VDL). The paper examines the relationship between VDL and both denotational semantics and the axiomatic approach to programming language definition.

1. Introduction

Software, commercial and scientific application programs in particular, constitute formidable investments by industry. High level languages play an important role in the protection of these investments by preserving the validity and meaning of programs across multifarious hardware, operating systems, and implementations, as well as over time, as the underlying hardware and systems evolve.

To remain stable both in form and meaning, a programming language needs a rigorously precise and implementation-independent definition. Such definitions do not emerge casually, as committees struggle to standardize a language. This has been painfully clear ever since FORTRAN and ALGOL 60 appeared on the scene.

The first significant contribution towards rigorous and formal language definition was made by John Backus [1] for the purpose of defining the syntax of ALGOL 60. The method and notation, known as BNF (Backus Naur Form), or minor variants thereof, have been used for virtually all programming languages since ALGOL 60. It enjoys general consensus. The method profoundly influenced compiler construction and stimulated numerous theoretical studies in computer science.

The success of formal syntax definitions invited similar attempts at the formalization of the semantic aspects of programming languages, *i.e.*, the definition of the meaning of programs rather than their form. The problem turned out to be obstinate. In spite of considerable progress, no satisfactory solution exists, at least none that enjoys general acceptance.

This state of affairs is not particularly surprising. The formalization of programming languages inherits a frame of reference from linguistics and formal logic [2, 3]; in spite of its distinct origin and purpose, the analysis of programming languages shares some of the difficulties with the analysis of natural languages.

However, the past two decades have seen considerable progress. Language constructs that were only understood on an intuitive, pragmatic basis, if at all, can now be defined and analyzed in precise mathematical terms.

The present paper is a retrospective contemplation of a piece of work that contributed to this relative success: the first complete formalization of the semantics of a commercially available and generally used programming language, *viz.*, PL/I.

The methodology and the actual definition of PL/I were developed during the years 1964 to 1969 by the IBM Laboratory Vienna under the management of H. Zemanek. The definition method and the related notation are known collectively as VDL (for Vienna Definition Language). The term "Vienna Definition Language," to my knowledge, was first used by J. Lee in [4]. The acronym initially used for the PL/I formalization was "ULD" for Universal Language Document.

2. Historical sketch of VDL

• *The motivation and start*

The efforts related to formal semantics at the Vienna Laboratory were triggered by a preceding experience with an implementation of ALGOL 60.

Unlike users of a language, who might only need a partial knowledge of the language, an implementer needs a complete understanding of the language to be implemented. At the time (1960), this was a rather demanding prerequisite. The combination of blocks, procedures as arguments, general goto statements, and recursion was especially difficult to master. It was hoped that a formal model could serve as the basis for a systematic design and justification of execution environments and compiler algorithms.

With this motivation in mind, preliminary investigations started in 1964.

The state of the art at that time was summarized by an IFIP Working Conference on "Formal Language Description Languages" [5], held in Baden bei Wien in September 1964. "Attendance was limited by invitation to recognized experts in one or more of the various disciplines of linguistics, logic, mathematics, philosophy, and programming whose frontiers converge around the subject of the meeting. The resulting group—51 individuals from 12 nations—was ideal in size, breadth of experience, and commitment to the enterprise." (The quotation is from the Preface to the Working Conference Proceedings.) Members of the IBM Vienna Laboratory, involved in the preparation of the conference, had the opportunity to become acquainted with the subject and the leading scientists.

• *The PL/I definition*

Work on the formal definition of PL/I started in September 1965.

The first version of the PL/I definition was completed in December 1966 [6]; two further versions [7] and [8] had been produced by June 1968 and June 1969, incorporating changes and extensions to the language. This paper refers mostly to the second version, which includes the axiomatic definition of storage, lacking in the first version.

Close cooperation with the IBM Hursley Laboratory, then responsible for the PL/I language and its implementation, had been established to ensure accuracy of the formalized language content and feedback concerning problems uncovered by the formalization. Members of the IBM Hursley Laboratory also contributed to the formal definition and its methodology.

The ANSI PL/I Standard [9] is based on the VDL methodology, though not identical to the earlier VDL formalization. After some exposition of the VDL method, we contrast the VDL version with the ANSI Standard for PL/I.

There has been some influence from the formalization exercise on PL/I during the phase when the language received its final shape. The most visible practical influence is seen in the use of the methodology in the ANSI Standard. Considerable academic research is traceable to the work of the Vienna Laboratory in the computer science literature.

The intended practical role of a formal definition, of both syntax and semantics alike, is threefold: first, to provide an authoritative reference document from which, possibly less formal, user manuals can be derived, and also as an arbiter concerning subtle questions of form and meaning; second, to provide a basis from which implementations can be derived systematically, if not formally or even mechanically, thus rendering portability of programs as an uncompromised reality. A third role for a formal definition is that it provides a basis for reasoning about programs, for program proofs.

These objectives imply that the single, primary, authoritatively binding reference document be the formalized one and that the formalization be produced by the same body that designs the language; furthermore, the second objective asks for the formal document to be available before implementation design proceeds.

The VDL definition of PL/I falls short on both counts. It was certainly too late to be used by implementers.

VDL definitions of languages other than PL/I have been produced: *e.g.*, ALGOL 60 [10] and BASIC [11]. The exercise of formalizing ALGOL 60 resulted in a document very much smaller in size than the PL/I definition, thus demonstrating that the size of the PL/I definition is not necessarily due to the particular definition method.

• *The period after VDL*

Following the formalization of PL/I, various projects at the Vienna Laboratory attempted to improve the definition method and attacked some major open problems.

Some progress had been achieved towards methods for verifying the consistency of formal source language semantics and its implementation. The first stumbling attempts to use VDL for that purpose are found in [12]. The published trace of this activity is found in [13–18].

A significant improvement in the metalanguage and the mathematical apparatus has been worked out and is known as META-IV. The new metalanguage together with an adaptation of the methodology (called VDM, for Vienna Development Method [19]), strongly influenced by the work of Scott and Strachey on denotational semantics, was successfully applied to a subset of PL/I [20]. The improvement achieved results in proofs of correctness for implementations that are shorter, more lucid, and thus more convincing than earlier attempts.

META-IV has recently been used for formalizations of CHILL [21] and Ada [22], at the Technical University of Denmark.

It became apparent that the techniques proposed in the context of formal semantics of programming languages and the related implementation verification were equally applicable to program design, in particular to data abstraction and stepwise concretization with related correctness proofs. An early example of this turn in the development can be found in [23], which to my knowledge is the first publication showing the axioms defining the "abstract data type" stack and the relation of this device to its implementation.

Since 1976, the methodology that has its roots in VDL and META-IV has been pursued patiently and diligently outside the Vienna Laboratory, manifest mainly in the publications of Dines Bjorner, *e.g.*, [24], and Cliff Jones [25, 26].

What has been outlined is the history of one school of thought, VDL, and the subsequent developments triggered by VDL; this is the purpose of the paper and is elaborated in more detail in the following sections.

Other significant developments have occurred, and the picture would be incomplete without attempting to position the methodology contemplated in this paper relative to the currently relevant scientific context; Section 8 attends to this duty.

3. The VDL method

Central to main line programming languages, including PL/1, is a category of imperative sentences ("command" and "executable statement" are synonyms). How can the meaning of an imperative sentence be explained? Intuitively, the intent of such sentences is to effect change. The command "Paint this wall green," faithfully executed by an obedient painter, will turn a possibly white wall into a green one.

The most direct explicate of the meaning of an imperative sentence is a function over a set of potential states, mapping given "current" states into successor states, thus indicating the change of state intended to occur when the command is executed. Referring to the quoted example sentence above, the universe of states would be the relations between some set of walls (that can be referred to or pointed to) and a set of colors and patterns that walls can be given.

The philosophy thus sketched is further supported by the following observation. Machine languages are invariably defined, in their respective reference documents, by first specifying the possible states of the machine, *i.e.*, memory structure and contents, registers, instruction counter, etc. Each instruction in the repertoire of the machine is then, in turn, defined by the effect of its execution on the state of the machine. The style of definition is usually semiformal, *i.e.*, a mixture of plain English and some formal technical notation. A formal mathematical model has been used by C. C. Elgot and A. Robinson in [27] for the study of certain general properties of machine languages; that paper was instrumental in the early formation of the VDL methodology.

The high level languages in practical use to date are abstractions and extrapolations of machine languages; historically, one observes an evolution from von Neumann type machines, to symbolic asssembly languages, to high level programming languages. One can question the wisdom and utility of this development but not the historical fact and state of affairs. High level languages falling in this vein of development are called von Neumann languages.

The methods for defining von Neumann languages are primarily distinguished by the formal explicate chosen for the set of states and the formal explicate chosen to mirror state transitions.

The domain of states (explicated by a set of mathematical objects) and state transitions (functions from states to states) together are called an abstract machine.

The use of abstract machines for the purpose of defining programming language semantics was first proposed by John McCarthy [28]. This paper by John McCarthy cited above also introduces the notion of abstract syntax and the proof principle for verifying implementations. Peter Landin also applied an abstract machine in his formalization of ALGOL 60 language concepts [29].

PL/I is significantly richer and more complex than the stylized example languages that had been discussed prior

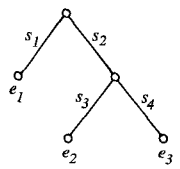

Figure 1 Composite object.

to VDL. The carefully designed formal explicates for the various concepts of PL/I (and similar languages) together with a consistent formal apparatus and metalanguage are the contributions of VDL.

Definitions in VDL are given in terms of a universal domain of objects. In the PL/I definition, there are two exceptions to this general rule: Concrete syntax is specified in a variant of BNF, and the storage component of the abstract machine is defined implicitly by axioms. The domain of objects is partitioned into two broad classes: atomic objects and composite objects. A supply of atomic objects is assumed as given and conveniently divided into subclasses. Composite objects are trees constructed recursively from atomic objects, composite objects, and a set of selectors. Roughly speaking, composite objects can be viewed as trees with branches labeled by selectors (unique at each level) and elementary objects at the leaf nodes. Figure 1 depicts a composite object with s_1, s_2, s_3, s_4 as selectors and e_1, e_2, and e_3 as elementary objects.

The domain of composite objects includes the null object Ω. Selectors, viewed as functions, can be applied to objects and yield the respective component, or Ω (the empty object), if such a component is not present; thus selectors are total functions.

There is a combined replacement/construction/deletion operator called μ. Let x be a composite object and y be an object (elementary or composite). The term $\mu(x;\langle s:y\rangle)$ yields x with the s component being replaced by y; if x has no s component, such a component is added; replacement by Ω amounts to a deletion of the respective component. The domain of objects is closed under the μ operation.

Conventional mathematical notation is used to define functions and predicates on the domain of objects; some additional notation is provided for concise specification of subdomains.

Such is the simple and uniform basis of VDL definitions; this simplicity comes at a price: maps, lists, stacks, etc., are all represented in terms of this one universal domain, thus exhibiting some unnecessary representational detail. The issue is discussed in more detail in [23]. META-IV employs a greater variety of primitive notions, such as maps, lists, tuples, etc. In either method, the respective primitive notions are used to formalize both, the syntactic constructs, *i.e.*, programs and parts of programs, as well as the semantic constructs, *i.e.*, the state of the abstract machine and its parts.

VDL definitions in general, and the PL/I definition in particular, follow a general method and plan; the order in which the parts of a definition are discussed below is not the order in which these have been or should be worked out, but is the order in which the parts of a definition are presented.

As usual there is a concrete, context free syntax, which is presented with a variant of BNF. However, it is unwise to relate the semantics of a complex programming language directly to its concrete syntax. Many secondary notational issues can be separated from the essential structure of such languages, such as: punctuation, conventions permitting omission of parentheses, default attributes (PL/I), and freedom of ordering with no semantic impact. The essential structure of a language is isolated by defining an abstract syntax (due to McCarthy [28]); it gives the semantically essential grammatical constructs and a canonical form for the language. An algebraic characterization of this form and its use in semantic definitions is to be found in the work of the ADJ group [30]. More detail is provided in the next section.

An abstract syntax is specified by equations analogous to a context free grammar (not identical, since the abstract syntax defines objects, *i.e.*, trees, rather than character strings).

The link between the concrete and abstract syntax is defined as a map, called a translator, from concrete programs to abstract programs.

The central part of the definition and focus of this paper is the specification of the abstract machine explicating the semantics of the language. A machine is specified in two parts: (1) a domain of states Σ, using exactly the same definition technique as was employed for the abstract syntax, *i.e.*, equations analogous to a context free grammar; and (2) a state transition function Λ, specifying computations for given initial states and programs. PL/I and most other practical languages include nondeterministic programs. Language features that may lead to nondeterministic programs include tasking and certain grammatical constructions where the sequence of elaboration of subphrases is left unspecified by definition.

The state transition function Λ is, therefore, a function mapping states into sets of possible successor states. A computation is thus a sequence of states:

$$\sigma_0, \sigma_1, \cdots \sigma_i, \sigma_{i+1}, \cdots$$

$$\sigma_{i+1} \in \Lambda(\sigma_i)$$

Both the abstract syntax of programs and the domain of states, given the definition method indicated, are too large, in the sense that the abstract syntax contains meaningless programs and the domain of states contains some that cannot occur given a particular state transition function.

As an aid, the definition of abstract programs and states is usually complemented by so-called context conditions that narrow both sets. For example, the restriction that the parameter list in a procedure declaration must not contain two occurrences of the same identifier could be stated as a context condition.

In summary we obtain the following structure of a VDL definition:

1. Concrete syntax,
2. Abstract syntax,
3. Translator,
4. Abstract machine.

The intent of such definitions is to serve as a basis for proving general properties of languages and programs, rather than to determine the effect of a specific program when applied to specific data. Of particular interest are theorems asserting the correctness of proposed implementation techniques, theorems useful in program verification (see Section 8), and also theorems related to the equivalence of given constructs (*e.g.*, can automatic variables in PL/I always be replaced by controlled variables and suitable explicit allocate and free statements?).

As mentioned in Section 2, the ANSI PL/I Standard uses a similar method of definition; the structure of the standard document is as outlined above. The detailed construction of the domain of states differs from the Vienna definition and is closer to the corresponding data structures needed in implementations. The definition of state transitions employs stylized English rather than a formal notation. The treatment of temporary results associated with complex state transitions is improved as compared to the Vienna definition. The definition of the state transitions in the ANSI Standard would probably be more compact than the VDL style definitions, if the stylized English were replaced by a formal notation.

4. Abstract syntax

The state transition function Λ is defined in terms of auxiliary functions; each syntactic category of the abstract syntax is usually related to exactly one such auxiliary function; thus the structure of the semantic definition parallels the syntactic structure. The key idea is that the meaning of a complex structure is defined in terms of the meanings of its constituent parts.

The punctuation marks and particles (mostly key words) play no role after the phrase structure and syntactic categories have been determined and are therefore absent from the abstract syntax.

The practical role of abstract syntax is, therefore, to minimize the number of grammatical categories and replace punctuation by an explicit structural definition, with the added advantage that the semantic definition becomes independent of semantically insignificant notational variations.

Let v be a variable name and e an expression; consider the following notational variants of the assignment statement:

$v = e$

$v := e$

$e \rightarrow v$

An abstract syntax would subsume all three variants under a definition:

assignment ::= ⟨*source*: *expr*, *target*: *var*⟩

where *source* and *target* are selectors for the respective essential parts of an assignment statement, one of syntactic category *expr* (abbreviation for expression), the other of syntactic category *var* (abbreviation for variable).

The concrete syntax of most programming languages includes the treatment of operator precedence by introducing auxiliary syntactic categories, one for each group of operators with the same precedence (the categories *primary*, *factor*, *term*, etc., in ALGOL 60 exist precisely for that purpose). Precedence rules are a notational convention that need not enter the essential structure of the language; the auxiliary categories can be eliminated in the abstract syntax:

expr ::= *var* | ⟨*rd1*:*expr*, *op*:*infix-op*, *rd2*:*expr*⟩ | ···

Expressions, *expr*, are defined to be either variables or composite objects consisting of three parts. Two of the parts are expressions obtained by applying the selectors *rd1* and *rd2* (short for operand one and operand two). The third part is an *infix-op* (infix operator) obtained by applying the selector *op*.

Note that the above abstract syntax definition would be hopelessly ambiguous if interpreted as a concrete syntax rule.

5. States

In the design of the domain of states for the abstract machine, the primary concern is simplicity, as the purpose is definition rather than implementation. In view of the volume and complexity of the VDL formalization of PL/I this statement may sound frivolous. The following few examples, starting with some rather simple language concepts, demonstrate how the state of the defining abstract machine is related to language concepts and how it has to be refined with each new idea introduced into the language.

The example published by J. McCarthy in [31] formalized a language (Micro ALGOL) involving simple variables, expressions, assignment statements, and conditional goto statements. Programs are sequences of such statements. States are simply maps from variables to their (current) values. Given a particular map, *state*, the value of a variable, *id*, may be determined via this map as indicated below; ID is the set of potential variables and VAL the set of potential values.

$$id \xrightarrow[state]{} val \qquad state: \mathrm{ID} \rightarrow \mathrm{VAL}$$

One special variable acts as the statement counter; its value points to the next statement to be executed.

If the language is enriched by permitting program variables to "share storage locations," the state of the abstract machine will have to reflect the fact that two distinct variable names may in fact denote "the same" variable, *i.e.*, an assignment to one variable also changes the value of the other. One method to reflect such "sharing" patterns is to introduce an indirect step. In the case of variables an auxiliary domain of names called locations (LOC) is introduced. A state consists now of two maps: a map from identifiers to locations, called environment (*env*), and a map from locations to values called storage (*stg*).

The following diagram shows how variables are connected to values in the new (provisional) design of the state:

$$v \xrightarrow[env]{} loc \xrightarrow[stg]{} value$$

A variable name denotes a location (map *env*) whose contents is a value (map *stg*).

Taking the full block structure and procedure invocation into account, the state of the abstract machine has to be further revised. Upon each block or procedure activation a new environment has to be formed including the locally defined names; upon exit the environment preceding the activation has to be re-established. The state must be extended by a stack component, keeping the environments of the suspended activations. The discussion of denotational semantics in Section 8 re-examines this step.

In addition to variable names, procedure names in the context of calls and as arguments have to be dealt with. Following the rules of block-structured languages, global names occurring in procedure bodies are bound in the environment of the block or procedure activation in which they are declared. In VDL definitions, procedure names denote pairs, $\langle body, env \rangle$, where the first component is the procedure body proper and the parameter list; *env* is a copy of the appropriate environment that binds the global names.

A closer look at the construction of the environment that needs to be paired with procedure bodies upon block entry reveals a difficulty. Since a procedure can contain global references to local names of the containing block, in particular recursive references to itself, the environment that needs to be constructed becomes an infinite object. The following example illustrates the point:

```
. . .
begin
    proc P(· · ·);
        · · · P(· · ·) · · ·      } body
    end
        . . .
end
```

Suppose the **begin** block is to be executed in environment *env*; the environment, *env'*, that has to be constructed upon entry into the block is given by

$$env' = env + \langle P, \langle body, env' \rangle \rangle$$

where maps are viewed as sets of pairs and *map1* + *map2* means: restrict *map1* by eliminating all elements from its domain that are in the domain of *map2*, and then form the union with *map2*.

The various state constructions, especially the above issue, have been discussed in depth by J. Reynolds [32].

It is only since the fundamental contributions of Dana Scott on denotational semantics [33–35] that such recursive domain equations are understood, *i.e.*, under which conditions such an equation makes sense and what domain is defined if it does.

One can easily see that *env'*, according to the preceding equation, is an infinite object by writing *env'* explicitly for the previous example:

$$env' = \{\langle P, \langle body, \{\langle P, \langle body, \cdots \rangle\rangle, \cdots\}\rangle\rangle, \cdots\}$$

The VDL definition avoided infinite objects in the state of the abstract machine. To avoid infinite objects as procedure denotations, another indirect step is employed; environments become maps from identifiers to unique names, N. Then what identifiers denote is indirectly associated by an additional part of the state, the denotation part, *den*, mapping unique names to denotations. For reasons of uniformity, this extra indirect step applies to all names, including variable names.

The following diagrams illustrate the relation between names and their denotations for the latest revision of the state.

Variables:

$$id \xrightarrow[env]{} n \xrightarrow[den]{} loc \xrightarrow[stg]{} value$$

Procedures:

$$P \xrightarrow[env]{} n \xrightarrow[den]{} \langle body, env \rangle$$

With the revision of the state one can easily see that *env'* (see preceding example) is no longer an infinite object:

$$env' = \{\langle P, n \rangle, \cdots\}$$

$$den = \{\langle n, \langle body, env' \rangle\rangle, \cdots\}$$

The environment *env'* contains a finite number of elements (one for each name declared in the begin block), and each of the elements is finite.

Since *env'* is finite, the copy of *env'* occurring in *den* does not make *den* infinite.

To capture all of PL/I's concepts, further revisions of the state become necessary. This is not the place to repeat all the details; these can be found in the literature. However, the treatment of the storage component in the PL/I definition introduced a new methodological aspect which is discussed in the next section.

The instruction counter used in the Micro ALGOL example does not yet have an analogous component in the revised state. Section 7 attends to this issue.

6. Implicit definitions, storage

The definition method related to VDL is usually categorized as operational or constructive, in contrast to axiomatic and so-called denotational methods (see Section 8). This is somewhat inaccurate; several parts of the formalization of PL/I are defined by nonconstructive means. The so-called storage component of the state of the PL/I machine is an example in point.

It was argued in Section 5 that a simple map from identifiers to values is insufficient to model the state of program variables for block structured languages permitting call by reference. The notion of "location" has been introduced to remedy the situation. The content function, a simple map from locations to values, served as the model of storage.

The relationships that may occur among PL/I variables are richer than those which can be explicated by this simple model. An initial attempt to produce a constructive model that would define precisely the properties of the PL/I language, *i.e.*, those which must hold independent of implementation, and no more, failed.

The storage model that was chosen is rather modern; in contemporary terminology it would be called an "abstract type." More precisely, certain domains are postulated together with functions and predicates on these domains; the properties of the domains, functions, and predicates are given by axioms.

The definition of the storage properties in axiomatic form appeared first in [7]; an introduction to the methodology is contained in [36]; a revised definition suitable for specialization to PL/I as well as ALGOL 68 was published by H. Bekic and K. Walk in [37] for the purpose of comparing the two languages.

The features of PL/I that have to be reflected by the axioms are: explicit allocation and freeing of storage, allocation and freeing within areas, pointers and their relation to based variables (*e.g.*, left to right equivalence), offsets, cells (similar to union types in ALGOL 68, this property has been deleted from PL/I), and alignment properties.

This is not the place to reprint the storage definition; the small example below should suffice to indicate the style of the definition.

The relevant domains are: *STG*, a set of storage objects (intuitively the set of all possible snapshots of storage); *P*, a set of locations; *VR*, a set of value representations. Roughly speaking, a location can be viewed as an address together with size information. The size of a location determines which values "fit" into it; in the case of PL/I, size not only relates to the number of bits required but also to alignment constraints. A location, *p*, is viewed as a function that can be applied to a given storage object, *stg*,

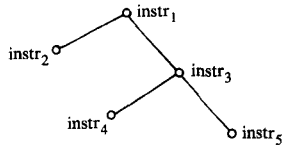

Figure 2 Control tree.

to yield that part of the storage object that corresponds to the location, $p(stg)$. A binary relation $is\text{-}indep(p1,p2)$ is postulated over the domain P of locations. Again intuitively, two locations are independent if they do not overlap or contain each other; obviously this relation must be symmetric.

$$is\text{-}indep(p1,p2) \supset is\text{-}indep(p2,p1)$$

An elementary assignment function is postulated,

$$el\text{-}ass: P \times VR \times STG \rightarrow STG,$$

which assigns a value representation to a location in a given storage object and yields the modified storage object. One of the properties of the elementary assignment function is stated by the axiom

$$is\text{-}indep(p1,p2) \supset p1(stg) = p1(el\text{-}ass(p2,vr,stg));$$

The axiom states that assignment to one location does not change the content of independent locations. Conversely, no definite property of the content of any location that is not independent of the assigned-to location can be derived on the basis of the above axiom. A program taking advantage of the storage mapping of a particular implementation, especially as regards the precise manner in which locations may overlap, is not implementation independent and thus not generally portable.

The full formalization in [7] of the storage properties requires a few additional domains and functions; about 40 axioms constrain those functions to reflect the related properties of PL/I.

7. State transitions, the meaning of statements

The method for specifying the state transition function Λ, mentioned earlier, may seem strange; motivation is required. The features of many programming languages including PL/I that make it difficult to define the state transition function in a straightforward manner are the combination of a nested phrase structure with general goto statements and nondeterminacy (unspecified sequencing).

Given a composite grammatical construct, *e.g.*, a conditional statement or iteration statement, one would like to compose the state transition of the compound from the state transitions associated with the parts. A statement, primitive or composite, could then be said to denote a function, *viz.*, the state transition function to be applied to the state when the statement is executed. Goto statements cut across this pleasant correspondence between syntactic and semantic structure such that this simple plan is difficult to follow. The issue is further discussed in connection with denotational semantics in Section 8.

Nondeterminacy also constitutes a barrier to the simple plan. The overall state transition effected by a compound phrase can be viewed as being composed of multiple atomic transitions. Unspecified sequencing means that the atomic transitions of two distinct phrases are merged (interleaved) arbitrarily. As a consequence, the overall state transition of the individual phrases does not provide enough information to determine the combined effect.

The question whether the concepts mentioned should be in programming languages at all is important but beyond the scope of this paper and had not been the problem VDL was intended to solve.

To define the state transitions and cope with the above indicated problems, VDL definitions include a control component in the state of the abstract machine. The control component can be viewed as a tree, called a control tree (see Fig. 2), whose nodes are associated with instructions. The construction can be represented in terms of the universal domain of objects; the details can be found in [36].

By convention, the instructions at the leaf nodes are candidates for immediate execution, and the choice is arbitrary. The execution of an atomic instruction is performed by effecting the state transition defined by the instruction and removing the instruction from the control component. Nonatomic instructions are like macros; they are replaced by a control tree. VDL provides notation for defining both kinds of instructions. Upon each block or procedure activation, the control component is stacked. The interpretation of a goto statement across block boundaries removes the appropriate number of stack levels. Instructions may, besides effecting a state transition, produce values. The control trees include a mechanism that permits the result of an instruction to be inserted in argument positions of predecessor nodes.

As an example, let $eval\text{-}expr(t,e)$ be the instruction that evaluates an expression t in environment e. If suitably defined, the instruction expands into the control tree shown in Fig. 3.

The dashed lines indicate data flow from the leaves to the root. For example, if *eval-expr*(*s-op-1*(*t*), *e*) is executed, the resulting value is placed into the first argument position of *eval-infix-expr*.

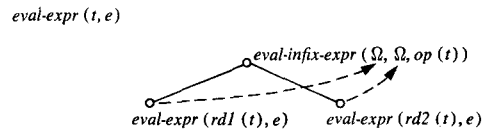

Figure 3 Example control tree.

8. The modern context, survey and relationships

It is not the intent of this paper to give a complete overview of the subject and the results that have been achieved. A brief historical review can be found in [38], some material from which appears here; the reader interested in the origins of the subject may wish to consult the *Proceedings of the IFIP Working Conference on Formal Language Description Languages* in 1964 [5].

The emphasis of this paper is on the applied rather than the theoretical aspects of the theme. Two approaches in particular are currently of practical relevance: the axiomatic approach and denotational semantics. It is instructive to examine the relationship between these two approaches and VDL.

- *Denotational semantics*

Denotational semantics has its roots in the early work of P. Landin [29, 39] and C. Strachey [40] with the λ-calculus [41] as the formal basis. Fundamental mathematical results due to D. Scott [33–35] rendered a firm basis upon which the plan of denotational semantics can be carried out. The textbook by J. Stoy [42] provides an admirable exposition of denotational semantics.

The formal semantics of a language, denotational style, associates with each phrase of the language a mathematical object; the phrase is said to denote that object. Conversely, this object is called the denotation of the phrase. The fundamental tenet of denotational semantics is that the denotation of a composite phrase is a function of the denotations of the immediate subphrases. A class of mathematical objects is associated with each grammatical category, and a function on those objects is associated with each grammatical construction; the function combines the mathematical objects associated with the components of the construction to obtain the mathematical object (denotation) associated with the phrase. There is a correspondence between the syntactic structure and the semantic structure of a language in the algebraic sense of the term "structure." This is elaborated in [30] where the denotation is required to be a homomorphism from the algebra of trees of the abstract syntax to an algebra of denotations.

Given a particular language, the problem is, of course, to find mathematical objects suitable for explication in the most direct and simple manner.

For the purpose of contrasting the methods, the following examples step through the same language concepts discussed in connection with VDL in Section 5. The term $\mathcal{I}[ph]$ refers to the denotation of the phrase *ph*. To indicate the various phrases to be discussed, an ALGOL-like and otherwise obvious notation is used, *e.g.*, $v := expr$ is an assignment statement composed from some expression *expr* and some variable v.

To define McCarthy's Micro ALGOL in denotational style one would first define a domain of states as before:

$$STG = ID \rightarrow VAL$$

The class of denotations to be attributed to statements, *st*, are functions from states to states:

$$\mathcal{I}[st]: STG \rightarrow STG$$

Expressions (without side effects) have functions from states to values as denotations:

$$\mathcal{I}[expr]: STG \rightarrow VAL$$

Examples for the definition of composite constructs are

$\mathcal{I}[st1;st2] = \mathcal{I}[st2] \circ \mathcal{I}[st1]$ where $\circ$ denotes functional composition

$$\mathcal{I}[v{:=}expr] = \lambda\, stg\ (assign(stg, v, \mathcal{I}[expr](stg)))$$

where $assign(stg, v, val) = stg'$

$$stg'(x) = \begin{cases} val \text{ for } x = v \\ stg(x) \text{ for } x \neq v \end{cases}$$

Goto's can be formalized using so-called continuations. The incorporation of goto's would require a revision of the denotations designed so far, leading to a structure that is not as straightforward as the above system of denotations. The technical details can be found in [43]; an in-depth discussion of the issue is presented in [44].

Ignoring the goto problem, the next revision incorporates block structure and procedures. This revision introduces the domains of locations, *LOC*, environments, *ENV*, and storage objects, *STG*, for the same reasons and purposes as in the VDL style.

$ENV = ID \rightarrow DEN$

$STG = LOC \rightarrow VAL$

DEN is a domain of mathematical objects comprising all those objects that identifiers can denote, including locations (*LOC*) denoted by variables. Denotations of procedure names are discussed shortly.

The major classes of denotations for composite constructs are

$\mathcal{M}[expr]$: $ENV \rightarrow (STG \rightarrow STG \times VAL)$

$\mathcal{M}[st]$: $ENV \rightarrow (STG \rightarrow STG)$.

Assuming expressions with side effects, given an environment, expressions determine a storage to storage transition and a value; statements determine a transition without yielding a value. The use of the term "transition" is convenient but somewhat misleading, since a definition, denotational style, does not need the notion of computation, *i.e.*, a sequence of states that an abstract machine assumes. For this reason there is no need to explicitly define a domain of environment stacks, as in VDL. However, by rather trivial considerations one may reconstruct the stack mechanism of the operational definition, if one so desires [16].

The most significant distinction between VDL and denotational semantics must be seen in the treatment of procedure denotations. For the present purpose it is sufficient to discuss parameterless procedures. Procedures simply denote transition functions from storage to storage. The environment, establishing what identifiers denote, would relate procedure names, p, to functions:

$env(p)$: $STG \rightarrow STG$

To see the implications for the definition, it is necessary to examine how such environments are constructed. Assuming a begin block with one local procedure declaration, the meaning of the block is defined by

$\mathcal{M}$[**begin proc** P; *body* **end**; *st* **end**] =

$\lambda\, env.$(**let** $env' = env + \langle P, (\mathcal{M}[body])(env')\rangle$,

$\mathcal{M}[st](env')$)

The crucial point of this definition is the equation defining the environment *env'* (**let** clause in the above formula), that is, the environment in which the statement, *st*, of the **begin** block must be performed.

The equation is of the form $env' = F(env')$, *i.e.*, solutions are fixed points of F. The environment, *env'*, that is wanted and that is said to be defined by the above equation is the "least" fixed point of F, least with respect to a universal domain and a partial order defined on that domain. The theory of denotational semantics constructs such a domain, the ordering relation and the conditions under which such a least fixed point exists, thus mathematically justifying the use of such equations.

The equation is similar to the one that was avoided in VDL, as discussed before, by introducing an additional set of auxiliary names and an indirect step in the referential structure of the state. Denotational semantics relates the state transition denoted by a procedure identifier directly via the environment; the VDL environment relates the identifier to a unique name, the unique name is then in turn (by the *den* map) related to the pair $\langle body, env\rangle$; by examining the definition of Λ for the case of procedure call statements one finds the associated state transition. Thus denotational semantics is more direct, a fact that is bound to be reflected in simpler correctness proofs.

Two aspects were mentioned in the introductory remarks on VDL that tend to distort the simple structural correspondence of the syntactic and the semantic domain: the presence of goto's and indeterminate sequencing. As mentioned earlier, goto's can be covered, though at a cost. Indeterminate sequencing has not found a satisfactory solution so far. There are proposed formalizations; see, *e.g.*, [45] and [46]. The problem is not just one of adequately formalizing a language construct, otherwise understood and accepted; it is unclear how to compose asynchronous processes in an orderly manner, with simple verification conditions, *i.e.*, at issue is not a definition of what exists but what is needed.

A definition for ALGOL 60 has been given in terms of denotational semantics in [47]. Ignoring notational differences, the PL/I subset definition [20] is of this kind. The language design group of Ada is preparing a denotational definition of Ada.

- *Axiomatic approach*

Research on axiom systems and proof theory suitable as a basis for program verification was initiated by R. Floyd [48] with a simple flowchart language as the object language. The results were refined and extended to high level languages by C. A. R. Hoare [49]. The subject has been most actively pursued including automatic program verification (see, *e.g.*, J. King [50]).

The axiomatic approach establishes a proof theory, a set of axioms and rules of inference, for proving properties of programs. Propositions about programs, or parts of programs, following [49], take the form

$p1\ \{st\}\ p2$

where $p1$ and $p2$ are propositions referring to program variables and st is a program statement. The intuitive meaning of the proposition is: if $p1$ is true immediately before the execution of st and the execution of st terminates, then $p2$ is true immediately after the execution of st. The propositions $p1$ and $p2$ are called the pre- and post-condition, respectively.

The axiom for assignment and the rule of inference for the compound statement "$st1;st2$," assuming the simplest language level, *i.e.*, expressions without side effects and no block structure, are given below.

$$p^{v}_{expr}\{v:=expr\}\,p$$

where p^{v}_{expr} is p with all occurrences of v replaced by $expr$

$$\frac{\textbf{if } p1\{st1\}p2 \textbf{ and } p2\{st2\}p3}{\textbf{then } p1\{st1;st2\}p3}$$

The set of propositions that are derivable for a given program, using the axioms and rules of inference, define the meaning of the program. It is in this sense that the proof theory defines the semantics of the language.

Pre- and post-conditions can be interpreted as propositions about the state of the computation before and after execution of the related program statement (VDL interpretation); with respect to a denotational definition, these propositions are about objects of the semantic domains. The type of propositions p in denotational terms has to be

$$\mathcal{I}[p]\colon STG \rightarrow BOOL.$$

The meaning of the new propositional form can then be understood as

$$p1\{st\}p2 \sim \mathcal{I}[p1]\,(stg) \supset \mathcal{I}[p2](\mathcal{I}[st](stg))$$

for all stg for which $\mathcal{I}[st]$ is defined.

Thus, the axioms are viewed as theorems in the framework of denotational semantics. This intimate relationship has been previously observed in [51], [16], and [42].

The close study of proposed axioms along these lines pays. For example, it is easily discovered [16] that the simple assignment axiom does not carry over to the next level of language, which includes block structure and procedures with call by reference; as soon as variables can denote the same location, the simple substitution of the target variable by the literal right-hand side of the assignment is no longer valid in general.

An axiomatic definition for PASCAL has been published in [52].

9. Challenges

This last section addresses some problems concerning the practical use of formal semantics.

There are two areas where the results and methodologies of formal semantics could have a major impact: language design and implementation.

The traditional pattern of language design invariably results in an informal reference document, which may possibly later be supplemented by a formal document. This pattern is outdated and needs to be changed. Formal semantics provides an intellectual tool most usefully applied in the process of creating a language, a process whose end result is a formal definition from which tutorials and other secondary literature are derived. In this setting, knowledge of the major principles and results in formal semantics appears as a prerequisite and indispensable tool of the language designer. The challenge is mostly in the educational sector of computer science.

An important motivation for the formalization of the semantics of programming languages was systematic implementation design. More precisely, the correctness of a proposed implementation design is argued with reference to a formal definition of the language to be implemented. In addition, and possibly of greater importance, a formal definition can be used in the design process to establish the range of alternative realizations. Some evidence for this claim is found in [13]. However, monolithic formal language definitions (*e.g.*, those cited in this paper) are not serving this purpose well.

Existing and proposed programming languages have much in common. Thus designers can draw from common technical knowledge and experience. It appears inappropriate to view a systematic implementation design as a series of successive transformations of the formal definition of the entire source language, with correctness proofs for each step. A better strategy is to consider parts of languages, isolatable concepts, and their related implementation techniques (*e.g.*, block structure and the related stack management).

From an engineering point of view, we need to establish a repertoire of algorithms and data structures (abstract types, in modern terminology), each associated with precisely stated assumptions about the source language. Formal syntax and semantics provide the frame of reference for stating these conditions. The burden of proof is on the inventors of new algorithms; the implementer needs to keep in evidence the conditions under which these algorithms work.

One observes that other engineering disciplines have reference books of the indicated kind; software engineering has not yet reached such a state of maturity. It appears that with respect to this latter topic the challenge is for scholarly research rather than education.

Acknowledgments

Many referees offered useful comments. I am especially grateful for the detailed suggestions and improvements suggested by D. Bjorner, C. B. Jones, J. Lee, J. Thatcher, and H. Zemanek.

References

1. J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proceedings of the International Conference on Information Processing, UNESCO, Paris 1959*, Oldenburg, Munich and Butterworths, London, 1960, pp. 125-132.
2. H. Zemanek, "Semiotics and Programming Languages," *Commun. ACM* **9**, 139-143 (1966).
3. H. Zemanek, "Formalization—History, Present and Future," *Programming Methodology: Lecture Notes in Computer Science*, Volume 23, Springer-Verlag New York, Inc., New York, 1975, pp. 477-501.
4. J. Lee and W. Delmore, "The Vienna Definition Language, a Generalization of the Instruction Definitions," *Preprints of the SIGPLAN Symposium on Programming Language Definition*, San Francisco, August 1969.
5. *Proceedings of the IFIP Working Conference on Formal Language Description Languages*, T. B. Steel, Jr., Ed., North-Holland Publishing Company, Amsterdam, 1966.
6. PL/I Definition Group, "Formal Definition of PL/I," Version 1, *Technical Report TR.25.071*, IBM Laboratory Vienna, 1968.
7. (a) M. Fleck and E. Neuhold, "Formal Definition of the PL/I Compile-Time Facilities," Version 2, *Technical Report TR.25.080*, IBM Laboratory Vienna, 1968. (b) K. Walk, K. Alber, K. Bandat, H. Bekic, G. Chroust, V. Kudielka, P. Oliva, and G. Zeisel, "Abstract Syntax and Interpretation of PL/I," Version 2, *Technical Report TR.25.082*, IBM Laboratory Vienna, 1968. (c) P. Lucas, K. Alber, K. Bandat, H. Bekic, P. Oliva, K. Walk, and G. Ziesel, "Informal Introduction to the Abstract Syntax and Interpretation of PL/I," Version 2, *Technical Report TR.25.083*, IBM Laboratory Vienna, 1968. (d) K. Alber, P. Oliva, and G. Urschler, "Concrete Syntax of PL/I," Version 2, *Technical Report TR.25.084*, IBM Laboratory Vienna, 1968. (e) K. Alber and P. Oliva, "Translation of PL/I into Abstract Text," Version 2, *Technical Report TR.25.086*, IBM Laboratory Vienna, 1968. (f) P. Lucas, P. Lauer, and H. Stiegleitner, "Method and Notation for the Formal Definition of Programming Languages," Version 2, *Technical Report TR.25.087*, IBM Laboratory Vienna, 1968.
8. (a) M. Fleck, "Formal Definition of the PL/I Compile-Time Facilities," Version 3, *Technical Report TR.25.095*, IBM Laboratory Vienna, 1969. (b) G. Urschler, "Concrete Syntax of PL/I," Version 3, *Technical Report TR.25.097*, IBM Laboratory Vienna, 1969. (c) K. Walk, K. Alber, M. Fleck, H. Goldman, P. Lauer, E. Moser, P. Oliva, H. Stiegleitner, and G. Zeisel, "Abstract Syntax and Interpretation of PL/I," Version 3, *Technical Report TR.25.098*, IBM Laboratory Vienna, 1969. (d) K. Alber, H. Goldman, P. Lauer, P. Lucas, P. Oliva, H. Stiegleitner, and K. Walk, "Informal Introduction to the Abstract Syntax and Interpretation of PL/I," Version 3, *Technical Report TR.25.099*, IBM Laboratory Vienna, 1969.
9. *Programming Language PL/I*, ANSI X3.53, American National Standards Institute, New York, 1976.
10. P. E. Lauer, "Formal Definition of ALGOL 60," *Technical Report TR.25.088*, IBM Laboratory Vienna, 1968.
11. J. Lee, "The Formal Definition of the Basic Language," *Computer J.* **15**, 32-41 (1972).
12. P. Lucas, "Two Constructive Realizations of the Block Concept and Their Equivalence," *Technical Report TR.25.085*, IBM Laboratory Vienna, 1968.
13. W. Henhapl and C. B. Jones, "The Block Structure Concept and some Possible Implementations with Proofs of Equivalence," *Technical Report TR.25.104*, IBM Laboratory Vienna, 1970.
14. W. Henhapl and C. B. Jones, "A Runtime Mechanism for Referencing Variables," *Info. Process. Lett.* **1**, 14-16 (1971).
15. C. B. Jones and P. Lucas, "Proving Correctness of Implementation Techniques," *Symposium on Semantics of Algorithmic Languages: Lecture Notes in Mathematics*, Vol. 188, E. Engeler, Ed., 1971, pp. 178-211.
16. P. Lucas, "On Program Correctness and the Stepwise Development of Implementations," *Proceedings Convegno di Informatica Teorica*, University of Pisa, Pisa, Italy, 1973, pp. 219-251.
17. C. B. Jones, "Yet Another Proof of the Block Concept," *Laboratory Note No. LN25.3.075*, IBM Laboratory Vienna, 1975.
18. C. B. Jones, "The Vienna Development Method: Examples of Compiler Development," *Le Point sur la Compilation*, M. Amirchaby and D. Neel, Eds., Institut de Recherche d'Informatique et d'Automatique, 1978.
19. "The Vienna Development Method: the Meta-Language," D. Bjorner and C. B. Jones, Eds., *Lecture Notes in Computer Science*, Vol. 61, Springer-Verlag New York, Inc., New York, 1978.
20. H. Bekic, D. Bjorner, W. Henhapl, C. B. Jones, and P. Lucas, "A Formal Definition of a PL/I Subset," *Technical Report TR.25.139*, IBM Laboratory Vienna, 1974.
21. *The Specification of CHILL* and supplement, *The Formal Definition of CHILL*, C.C.I.T.T. (International Telegraph and Telephone Consultative Committee), Recommendation Z200, Geneva, Switzerland, 1980.
22. "Towards a Formal Description of Ada," D. Bjorner, Ed., *Lecture Notes in Computer Science*, Vol. 98, Springer-Verlag New York, Inc., New York, 1980.
23. P. Lucas, "On the Semantics of Programming Languages and Software Devices," in *Formal Semantics of Programming Languages: Courant Computer Science Symposium*, Randall Rustin, Ed., New York University, 1970.
24. "Abstract Software Specification," D. Bjorner, Ed., *Lecture Notes in Computer Science*, Vol. 86, Springer-Verlag New York, Inc., New York, 1980.
25. C. B. Jones, "Constructing a Theory of a Data Structure as an Aid to Program Development," *Acta Informatica* **11**, 119-137 (1979).
26. C. B. Jones, "Software Development: A Rigorous Approach," *Prentice-Hall International Series in Computer Science*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.
27. C. C. Elgot and A. Robinson, "Random-Access Stored-Program Machines: An Approach to Programming Languages," *J. ACM* **11**, 365-399 (1964).
28. J. McCarthy, "Towards a Mathematical Science of Computation," in *Information Processing 1962*, C. M. Popplewell, Ed., North-Holland Publishing Company, Amsterdam, 1963.
29. P. J. Landin, "The Mechanical Evaluation of Expressions," *BSC Computer J.* **6**, 308-320, 1964.
30. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Initial Algebra Semantics and Continuous Algebras," *J. ACM* **24**, 68-95 (1977).
31. J. McCarthy, "A Formal Description of a Subset of ALGOL," *Proceedings of the IFIP Working Conference on Formal Language Description Languages*, T. B. Steel, Jr.,

Ed., North-Holland Publishing Company, Amsterdam, 1966.
32. J. C. Reynolds, "Definitional Interpreters for High-Order Programming Languages," *Proceedings of the 25th ACM National Conference*, 1972, pp. 717–740.
33. D. Scott, "Outline of a Mathematical Theory of Computation," *PRG-2*, Oxford University Programming Research Group, Oxford, England, 1970.
34. D. Scott, "Mathematical Concepts in Programming Language Semantics," *AFIPS Conf. Proc., Spring Jt. Comput. Conf.* **40,** 225–234 (1972).
35. D. Scott, "Data Types as Lattices," *SIAM J. Computing* **5,** 522–587 (1976).
36. P. Lucas and K. Walk, "On the Formal Description of PL/I," *Annual Review in Automatic Progamming* **6,** Pergamon Press, Inc., Elmsford, NY, 1969.
37. H. Bekic and K. Walk, "Formalization of Storage Properties," *Symposium on Semantics of Algorithmic Languages: Lecture Notes in Mathematics*, E. Engeler, Ed., Vol. 188, Springer-Verlag New York, Inc., New York, 1971.
38. P. Lucas, "On the Formalization of Programming Languages: Early History and Main Approaches," *The Vienna Development Method: the Meta-Language:* D. Bjorner and C. B. Jones, Eds., *Lecture Notes in Computer Science*, Vol. 61, Springer-Verlag New York, Inc., New York, 1978, pp. 1–23.
39. P. J. Landin, "A Correspondence between ALGOL 60 and Church's Lambda-Notation," (2 parts), *Commun. ACM* **8,** 89–101 and 158–165 (1965).
40. C. Strachey, "Towards a Formal Semantics," *Proceedings of the IFIP Working Conference on Formal Language Description Languages*, T. B. Steel, Jr., Ed., North-Holland Publishing Company, Amsterdam, 1966, pp. 198–220.
41. A. Church, "The Calculi of Lambda-Conversion," *Annals of Mathematical Studies*, No. 6, Princeton University Press, Princeton, NJ, 1941.
42. J. Stoy, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, MA, 1977.
43. C. Strachey and C. Wadsworth, "Continuations: A Mathematical Semantics which can Deal with Full Jumps," *PRG-11*, Oxford University Programming Research Group, Oxford, England, 1974.
44. C. B. Jones, "Denotational Semantics of GOTO: An Exit Formulation and Its Relation to Continuations," *Vienna Development Method: the Meta-Language*, D. Bjorner and C. B. Jones, Eds., *Lecture Notes in Computer Science*, Vol. 61, Springer-Verlag New York, Inc., New York, 1978, pp. 278–304.
45. H. Bekic, "Towards a Mathematical Theory of Processes," *Technical Report Tr.25.125*, IBM Laboratory Vienna, 1971.
46. G. D. Plotkin, "A Powerdomain Construction," *SIAM J. Computing* **5,** 452–487 (1976).
47. P. D. Mosses, "The Mathematical Semantics of ALGOL 60," *PRG-12*, Oxford University Programming Research Group, Oxford, England, 1974.
48. R. W. Floyd, "Assigning Meaning to Programs," *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, J. Schwartz, Ed., Vol. 19, American Mathematical Society, Providence, RI, 1967, pp. 19–32.
49. C. A. R. Hoare, "The Axiomatic Basis of Computer Programming," *Commun. ACM* **12,** 576–583 (1969).
50. J. C. King, "A New Approach to Program Testing," *Programming Methodology: Lecture Notes in Computer Science*, Vol. 23, Springer-Verlag New York, Inc., New York, 1975, pp. 278–290.
51. Z. Manna and J. Vuillemin, "Fixed-Point Approach to the Theory of Computation," *Report AIM 164*, Stanford University Computer Science Department and Department of Artificial Intelligence, Stanford, CA.
52. C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *ACTA Informatica* **2,** 335–355 (1973).

Received September 14, 1980; revised February 25, 1981

The author is located at the IBM Research Division laboratory, 5600 Cottle Road, San Jose California 95193.