ON THE FORMAL DEFINITION OF ADA

V.Donzeau-Gouge, G.Kahn, B. Lang
IRIA-Laboria, Rocquencourt FRANCE
B. Krieg-Brueckner
CII-HB

ABSTRACT: This report presents the essential design decisions that were made when drafting the Formal Definition of the programming language Ada, commissioned by the U.S(Department of Defense. The goals, structure and conventions of the document are outlined. This paper constitutes an introduction to reading the Formal Definition.

KEYWORDS: Programming Languages, semantics, programming environment.

1 Introduction

As a part of the DOD-1 language effort, the Steelman Report has required a formal definition (1H). This requirement was both innovative and far-sighted.

Purposes

It may seem at first that devising the Formal Definition of a programming language is essentially an academic exercise. In fact, the formal definition is called to play an important role in several aspects related to the acceptance of the language by a large community of users. A formal definition can serve:

(i) As a standard for the language, that is as a means to answer unambiguously all questions that a programmer or an implementor may raise about the meaning of a construct of the language. The formal definition should serve as a reference document for the validation of implementations and as a guideline for implementors. It will permit to unify the user interface across implementations (e.g. error messages) and the interface between processors manipulating programs (e.g. mechanical aids for normalization and documentation of Ada programs).

(ii) As a reference document for justifying the validity of optimizations and other program transformations. The only valid optimizations will be those that do not alter the meaning of a program.

(iii)As a reference document for proving properties of programs written in the language. In particular, it will allow the derivation of inference rules that can be used conveniently when proving properties of programs.

(iv) As an input for a compiler-generator when the technology becomes available. The Formal Definition of Ada is specified with enough precision to be processed, except for some straightforward notational transformations, by the experimental system SIS [Mosses].

Furthermore, the concurrent development of Ada and its formal definition have already resulted in further major benefits:

- Difficulties in early drafts of the Reference Manual (such as lack of clarity, ambiguities, omissions or inconsistencies) have been uncovered very early.

- Feedback was established to strive for economy of concepts in the Ada language.

These benefits are essentially independent of the particular method of definition that has been selected.

Requirements

When designing the formal definition of a language like Ada, there are two major requirements to keep in mind:

(i) The definition must be complete. If the definition is not complete its usefulness as a reference will be seriously diminished. This completeness can only be achieved by using a mathematically well-founded definitional method.

As of the Spring of 1979, however, the State of the Art in formal semantics does not allow us to offer a mathematically meaningful semantics for all issues concerning tasking. This is a very serious gap in our theoretical understanding of programs. Research in Semantics of parallelism is extremely active [Kahn] but the conclusion does not seem very near. No attempt has been made to give a dynamic semantics for task synchronization in Ada, while it is hoped that all other aspects of the language are satisfactorily covered.

In all matters relating to concurrency, the readers will have to do with the textual description of the dynamic semantics that is provided, pending a scientific breakthrough.

(ii) The Formal Definition of Ada is meant to be used in an industrial environment. Therefore extreme care must be given to notations. Considerations of compactness and mathematical elegance that are of prime importance in a scientific environment become less central in an engineering environment. A great deal of effort should be spent on the style of the definition and its intuitive content, to make it accessible to the intended readership: implementors of compilers, standardization committees, educated Ada programmers. Naturally, such an attempt should preserve the mathematical rigor of the definition, and should be seen merely as the development of a convenient notation.

The formal definition given here is akin to a large program. Special attention has been given to several key issues:

- The structure of the description reflects the underlying semantic concepts of the language.

- The choice of identifiers stays as close as possible to the terminology of the Reference Manual.

- The style of the description is homogeneous and uniform conventions are used throughout.

Method

There are three widely accepted methods of formally defining the semantics of a programming language.

(a) Operational Semantics

In this method, best exemplified by the Vienna Definition Method, the semantics is modelled by the behavior of an abstract machine. This has a practical appeal but also presents several problems:

(i) The mechanism of the abstract machine tends to overspecify the language since all details of machine-state transitions must be given.

(ii) It is not immediately obvious that the language has been well defined. One must rely on a proof that any execution terminates with a unique answer.

(iii) The theory of operational semantics is, in fact, rather difficult and not well-understood. Using an operational semantics to validate optimizations or to prove properties of programs is intricate because we are not well-equipped to reason logically about the behavior of a complex machine.

Recent advances in operational semantics should make this approach more suitable in the future. [Hennessy-Plotkin, Huet-Levy]

(b) Axiomatic Definition

This method is very popular because it is directed towards proving properties of programs. Its deficiencies, however, render it unsuitable for the definition of a language like Ada:

(i) First, giving some properties of language constructs cannot constitute a definition, unless some proof of completeness can be given.

(ii) An axiomatic definition is not adapted to a use by implementors since many details about the dynamic semantics cannot be formalized adequately.

(iii) No complete axiomatic definition of a large programming language has ever been carried out successfully, to date. Treatment of exceptions, for example, does not fit well in this formalism. Research in this area is active however. [Luckham-Polak]

(c) Denotational Semantics

We have elected to present a formal definition of Ada using denotational semantics. There are several reasons for choosing this method:

(i) It allows the definition of the language to any desired level of detail.

(ii) The method has been used (with success) on a number of languages with characteristics similar to those of Ada: Pascal, Algol 60, CLU, etc. [Tennent, Mosses, Scheifler]

(iii)The mathematics underlying this method have been extensively investigated. The method is based on very strong theoretical foundations.

(iv) It is well-suited to proving the validity of program transformations and proving properties of programs. [Milner]

A potential objection to the use of this method is the arcane style of presentation traditionally favored by its talented advocates. [Milne-Strachey] We hope to have overcome this difficulty.

Summary of Denotational Semantics

It is not the place here to make a comprehensive presentation of the method pioneered by Strachey and Scott. The reader is referred to the existing textbooks on the subject.[Stoy, Gordon]. Here, we shall just outline very quickly the essential ideas of the method.

In denotational semantics, one wishes to associate to every program an abstract mathematical object called its meaning. Usually, the meaning of a program is some functional object, say a function from inputs to outputs. The mapping that specifies how one associates a meaning to every program in Ada is called the denotational semantics of Ada. To properly define the denotational semantics of a language, one must first define a semantic universe, where meanings are to be found. Then one describes how to associate a meaning to every atomic component of a program and, for every construct of the language, how to derive the meaning of a compound fragment of program from the meaning of its subparts. Hence, denotational semantics is nothing but a rather large, recursive definition of a function from syntactic objects - programs - to semantic objects - input-output functions.

Defining the semantics of a language in this way naturally leads to assigning a meaning not only to complete programs but also to program fragments, a very useful mathematical property known as referential transparency. The recursive structure of the syntactic objects is well captured by the abstract syntax of Ada. Section 2 is devoted to a detailed presentation of the abstract syntax of Ada, that is of the tree form of Ada programs.

There is a wide body of literature discussing the mathematical nature of the semantic domains that need to be used. At first, it is not necessary to understand in depth the mathematical theory of these domains in order to follow the semantic description of Ada. In fact, denotational semantics uses a very small number of concepts. We shall describe, in general terms, three keys ideas that pervade the whole definition.

Ada is an imperative language. Understanding it requires some notion of a store. Programs use the store and update it as they are executed. Now if we wish to describe the store as abstractly as possible, that is without assuming any particular implementation, all we need to know is that it defines a mapping

STORE: LOCATIONS ---> VALUES

If s is a store and l is a location the expression s(l) will then denote the value stored at location l. To update the store, we will assume the existence of a function UPDATE that, given a store s, a location l and a value v returns a new store s' = UPDATE(s,l,v) that differs from s only by the fact that s'(l) = v. Typically, it is the purpose of an assignment statement to modify the store.

Another feature of Ada is its name structure. This structure allows a given identifier to refer to different objects, depending on where it occurs in a program. To model this phenomenon abstractly, we will assume the existence of a mapping:

ENVIRONMENT: IDENTIFIERS ---> DENOTATIONS

Here again, by merely saying that an environment is such a mapping, we want to avoid describing any particular implementation of this concept. The primary purpose of declarations is to modify the environment. In Ada, however, there are many other ways to alter the environment.

As a third example, let us consider the problem of describing the control mechanism of Ada. At first it would not seem too easy to describe it in a referentially transparent manner. If the meaning of an assignment is some transformation of the store, the meaning of a sequence of assignments should be the composition of these transformations. But what if we wish to give meaning to a **goto** statement or an **exit** statement? How can we describe the raising of an exception, either explicitly or during the evaluation of an expression.

A very general technique allows us to deal with this kind of problem in denotational semantics. Intuitively, the idea here is to give to the semantic functions an extra parameter that specifies "what-to-do-next". This parameter is called a continuation. The meaning of a program fragment is in general also a continuation. Typically, the meaning of an assignment statement with continuation c is obtained by prefixing c with a store to store transformation. In fact, Ada has a sophisticated exception mechanism, implying the use of a whole exception environment associating a continuation to each exception handler.

Continuations are not very easy to understand at first. The Static Semantics, where it is specified what checks need to be performed "at compile time" on Ada programs, does not use any continuations, so that it is possible to become thoroughly familiar with the Formal Definition's approach before having to tackle this concept.

Style of the Definition

Given that the first objective of the Formal Definition is to serve as a reference document for implementors, a great deal of attention was given to the choice of the meta-language, i.e. the language in which Ada is to be formally described. The typographical conventions of the Oxford School, with their intensive use of Greek letters and diacritical signs, are not ideally suited to an audience of programmers and engineers. The notation developped in [Mosses], (which is used as input for his system SIS) is a much better candidate already. Mosses' notation is elegant, machine readable, convenient to use for anybody familiar with applicative programming and efficient in its treatment of abstract syntax. We have tried to go even further towards usual programming convention in using a narrow (applicative) subset of Ada itself as a meta-language.

A minor extension was needed in order to allow procedures as arguments and results. Italics, boldface, upper and lower case are used systematically to avoid confusion between language and metalanguage. Identifiers in distinct fonts are considered to be distinct. It is hoped that the increased understandability of the Formal Definition will compensate for a definite loss of elegance.

In keeping with the goal of minimizing the number of new notations, we have attempted to stay close to the terminology of the Reference Manual, refraining from introducing new names unless they were absolutely necessary. Furthermore, rather than presenting

the Formal Definition as a completely separate document, we have followed the structure of the Reference Manual. The equations of the Formal Definition intend to make more explicit the English text in the Reference Manual. They are folded in the Reference Manual, so to speak. Experience with the Formal Definition will show whether this is the right approach.

As a final remark, let us indicate that we make extensive use of the abstraction facility of Ada. It may seem unfortunate that we could not avoid using one of the seemingly more advanced features of the language. But in fact, all we really need is a way to specify a collection of related functions together with their types. This concept is very familiar in mathematics as an algebra. Similarly, the use of the generic facility corresponds directly to the notion of a polymorphic function (or functional) in mathematics. In fact, all (value returning) procedures defined in the document are functions in the mathematical sense. The sublanguage of Ada that is used is purely applicative and the only "side effects" involve the construction of new objects.

2 Abstract Representation of Programs

In this section, we present a standard way of representing programs. It is to be used not only to define the semantics of the Ada language but also as a standard interface between all processors manipulating Ada programs. Programs are represented as trees, called Abstract Syntax Trees. These trees are defined with the help of the Ada's encapsulation facility, so as not to preclude subsequent efficient implementation.

2.1 Motivations

Since the meaning of programs will be defined recursively on their structure, it is necessary to specify with great precision what this structure is before developing the Formal Definition per se. On the other hand, quite apart from the Formal Definition, there is considerable interest in standardizing the representation of programs. This standard representation will play a crucial part in the harmonious development of the programming support, a collection of issues addressed in Pebbleman. Typical tools that are to benefit from such a definition are: syntax-oriented editors, interpreters and compilers, documentation and normalization aids, program analyzers, optimizers, verification tools.

2.2 Requirements

We now list some requirements that an abstract representation must satisfy to be effective as a standard:

(a) It must be possible to implement it efficiently on a variety of machines.

(b) It must reflect the structure of programs. For example, it must be easy to recognize and isolate program fragments such as statements, procedures, declarations, expressions, identifiers, etc...

(c) It must be easy to manipulate and modify.

(d) It must include all meaningful information contained in the original program text. In particular it must be possible to restore the program text from the representation, up to minor standardizations.

(e) It should not be cluttered with irrelevant information.

(f) It must have a simple and usable mathematical definition since it will be a foundation for the Formal Definition.

(g) Finally, as a matter of course, it must allow the representation of any legal Ada program.

Requirements (b) and (c) rule out the textual representation of programs. It is easy to see that many processors would need a "parser" as a mandatory front end. It would also be a mistake to use a parse tree as usually produced by a parser: such trees depend on the parsing method used and are cluttered with irrelevant details (Requirement (e)).

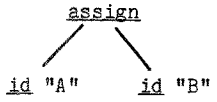
Common intermediate languages designed for optimization fail requirements (b) through (d). Using abstract syntax, a method put forward in the early sixties is very natural, simple and meets requirements (a) through (g).

2.3 Abstract Syntax Trees

The essential idea underlying abstract syntax is the treatment of programs and program fragments as trees. For example, the assignment

A := B

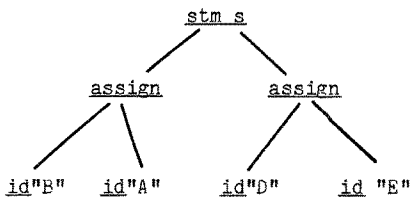
will be (pictorially) represented by the tree t.



Each node in the tree is labeled by a construct. In our notation, the construct labeling the top node of the tree t is denoted by KIND(t). Here KIND(t) = assign. The subtree representing the left-hand-side of the assignment is denoted by SON(1,t) and the subtree denoting the right-hand-side by SON(2,t). The whole Ada language is defined using 126 constructs. Most constructs label trees with a fixed number of sons. These constructs are said to be of fixed arity. To represent lists, it is necessary to use nodes that may have an arbitrary number of sons. For example the fragment

```
B := A;
D := E;
```

is represented as



The construct assign is binary while stm s is a list construct. The node labeled stm s could have an arbitrary number of sons.

Notations: All Ada constructs have been written underscored. List constructs have names ending in _s, like stm s, exp s, or decl s.

Not all trees labeled with constructs are abstract syntax trees. A grammar imposes a restriction on the strings of terminal symbols that are sentences of the language it defines. The Ada abstract syntax is similarly defined by a tree grammar. This grammar specifies precisely which trees are Ada trees. Let us define a sort to be a set of constructs. The abstract syntax of Ada is specified with the help of 57 sorts. If the root of a tree t is a construct belonging to sort s, we say that t is of sort s. The entire abstract syntax of Ada is completely specified by giving, for each construct, its arity as well as the sort of each son.

Note that list constructs are homogeneous: all constituents of a list must be of the same sort.

Notations.

(a) Sorts are written underscored and capitalized (e.g. COND). When a sort is a singleton sort (i.e. it contains a single construct), it has the same name as its member, but capitalized. Furthermore, since list constructs are characterized by the common sort of their constituents, their name always reflects that sort. As an example, a node labeled stm s has subtrees of sort STM, a node labeled decl s has subtrees of sort DECL.

(b) A notation similar to BNF has been used to specify the sorts. When writing for example:

COND ::= EXP | condition

we mean that COND is the union of sort EXP and the singleton set {condition}. Since sorts and constructs are distinguished typographically, the symbol | is used without ambiguity. For each construct, a sequence of sorts is given. For example the specification

if -> CONDITIONAL_S STM_S

means that the first son of an if construct is of sort CONDITIONAL_S and the second son is of sort STM_S.

Formally,

```
SORT_OF_SON(if,1) = CONDITIONAL_S
SORT_OF_SON(if,2) = STM_S
```

In the case of list constructs, the fact that all constituents belong to the same set is emphasized by the use of three dots as in

```
stm_s -> STM ...
```

A skeleton for the Abstract Syntax of Ada is shown below, encapsulated in an Ada package as described in section 2.4 .

```
package ADA_SYNTAX is
   type CONSTRUCT is (

         -- nullary constructs
         and          , and_then   , catenate , etcetera...

         -- unary
         abort        , access     , address  , etcetera...

         -- binary constructs
         alternative  , array      , assign   , etcetera...

         -- ternary constructs
         accept       , binary_op  , block    , etcetera...

         -- arbitrary constructs
         alternative_s , bounds_s   , choice_s , etcetera...   )

   type ARITIES is (nullary, unary, binary, ternary, arbitrary);
   function ARITY (construct: CONSTRUCT) return ARITIES;

   type SORT is set_of (CONSTRUCT);
      -- We assume a generic package set has been defined
      -- which provides sets and union of sets

      ALTERNATIVE, ALTERNATIVE_S, BINARY_OP, etcetera...
                                          : constant SORT;

   function SORT_OF_SON(construct: CONSTRUCT; n: INTEGER := 0) return SORT;
      -- the expression SORT_OF_SON(construct, n) denotes the sort of the
      -- n-th argument of "construct", if it is of fixed arity.  In the case
      -- of a list construct, it denotes the common sort of each son.

private

   -- The sorts are  sets described in a table included here.

   -- The structure of each Ada construct is gien in a table included next.


end ADA_SYNTAX;
```

```
package ADA_TREES is
   use ADA_SYNTAX;
   type TREE is private;
   -- Tree constructors

   procedure MAKE(construct: CONSTRUCT; s: STRING)        return TREE;
   procedure MAKE(construct: CONSTRUCT; t: TREE)          return TREE;
   procedure MAKE(construct: CONSTRUCT; t1, t2: TREE)     return TREE;
   procedure MAKE(construct: CONSTRUCT; t1, t2, t3: TREE) return TREE;

   -- Tree selectors

   procedure KIND    (t: TREE)              return CONSTRUCT;
   procedure SON     (n: INTEGER, t: TREE)  return TREE;
   procedure TOKEN   (t: TREE)              return STRING;

   -- Handling of list constructs

   procedure HEAD    (l: TREE)              return TREE;
   procedure TAIL    (l: TREE)              return TREE;
   procedure PRE     (t: TREE, l: TREE)     return TREE;
   procedure EMPTY   (construct: CONSTRUCT) return TREE;
   procedure IS_EMPTY(l: TREE)              return BOOLEAN;


private
   -- Description of the implementation of type TREE
end ADA_TREES;
```

2.4 Encapsulation of the Abstract Syntax

To be certain that the abstract syntax of Ada can be used as a standard for the representation of Ada programs, we could define it as a Ada data structure. This would not however leave enough room for efficient implementation and would involve unnecessary and harmful overspecification. Instead, we have chosen to specify only the visible part of Ada packages that provide the abstract syntax of Ada and the tools for the manipulation of Ada trees. Notice that the procedure MAKE is overloaded. This avoids creating one procedure name per construct. This overloading will be resolved on the basis of the number of arguments handed to it in any call. The procedure MAKE must be programmed using the KIND and SORT_OF_SON procedures provided in the package ADA_SYNTAX, to check that it is not asked to build unlawful Ada trees. Similarly, the constructor procedure EMPTY checks that its argument is a construct of arbitrary arity.

Most processors will find the selector function SON perfectly adequate. For the Formal Definition, where readability is of prime importance, we have assumed the existence of a third package, ADA_SELECTORS. This package allows to refer to subtrees by name rather than by position. A simple convention for the names of the selectors has been followed in the Formal Definition: for each sort, a selector function is defined that is named after the sort. Assume now, for example, that "statement" is a tree with a root labeled if. Instead of writing:

```
SON(1,statement)
```

we may write

```
CONDITIONAL_S(statement)
```

In cases like the binary construct pair that has more than one son of the same sort, numbering is used. Thus EXP1(pair) and EXP2(pair) return the first and second son of the tree pair, respectively, as both are of sort EXP.

3 Structure and Notations

In the English language description of the semantics of Ada given in the Reference Manual, one can distinguish three kinds of concerns:

(i) Some features of the language are provided to shorten the text of programs or to increase their readability. These features are best explained as combinations

of other possibilities of Ada.

(ii) A number of specifications are intended to delineate the class of legal programs, within the class of syntactically correct ones. Considerations such as the need to declare every identifier before using it, coherence in the use of types and resolution of ambiguity in the use of overloading, are in this category.

(iii)The rest of the informal definition concerns the behavior of programs during execution.

The Formal Definition is structured in a manner that reflects these quite distinct concerns.

3.1 Normalization

One part of the Formal Definition specifies transformations of the abstract syntax tree that do not require any type information. These transformations are performed to eliminate the use of some notational conveniences or to check simple syntactic constraints. They are defined by functions mapping TREE's to TREE's and regrouped in an Appendix of the Formal Definition. Whenever these functions are sufficiently simple (i.e. involve no context), the text includes their description as a simple rewriting rule.

Example:

```
[ if CONDITIONAL_S else STM_S end if; ] ->
[ if CONDITIONAL_S elsif true then STM_S end if; ]
```

The kind of constraints dealt with by normalizations must require only little contextual information, in particular no information about types. For example, when the Manual states:

> "Within the sequence of statements of a subprogram or module body, different labels must have different identifiers."

this check is one of those performed in this normalization phase.

3.2 Static Semantics

The next part of the Formal Definition is concerned with what is usually called type-checking. A type checker is presented as a mapping from abstract syntax trees to an extended abstract syntax tree, rather than as a mapping returning true or false. This is intended to mimic the concepts of "compile time" checks as opposed to "run time" checks. Type-checked programs contain all type information needed at run time, and only that type information. In this way dynamic semantics will not need to carry a static environment.

More specifically, the Static Semantics of Ada has to deal with the following tasks:

It must check that the declarations are valid, i.e. there is no repeated declaration of the same designator in the same scope. It must check that all designators are declared.

2. It must check that all designators are used in a manner that is consistent with their type.

3. It must carry out the evaluation of static expressions where required.

4. All information on types of designators must be used to generate an extended abstract syntax tree. This includes:

   4.1 Detecting and eliminating all overloading

   4.2 Reordering actual parameters in subprogram calls. Remember that Ada uses both positional and named parameters in subprogram calls. Once it has been processed, a subprogram call will list all its parameters in named parameter associations.

   4.3 Normalizing aggregates as lists of named component associations.

   4.4 Resolving ambiguities between indexed component, qualified expression and subprogram call.

5. Exception names are made unique within a program

6. The dot notation is systematically used to access identifiers visible through a use list.

Furthermore, the Static Semantics is given additional structure in collecting together in separate packages:

- a package that abstracts away the structure of the static environment, where information regarding the type of designators is recorded. The external behavior of this "abstract machine" is defined by a collection of functions that

  - build or select type denotations

  - declare or access designators

- a package that collects together auxiliary functions used:

  - to solve overloading

  - to check for side effects of functions and value returning procedures

3.3 Dynamic Semantics

The language Ada insists that a large number of verifications should be done "at compile time". It should come to no surprise that the precise description of the

type-checking of Ada should form a significant part of its formal definition. In contrast, the dynamic semantics of Ada, if you will remember that tasking is not dealt with in our Formal Definition is rather more conventional.

The Static Semantics is described as a transformation performed on abstract syntax trees. The Dynamic Semantics corresponds more to the customary notion of interpretation. The meaning of each construct is defined recursively on type checked abstract syntax trees. Information about the identifiers in the program (e.g. the value of a constant, the constraints associated with a subtype) is recorded in the dynamic environment.

The functions used in Dynamic Semantics are partitioned into three groups, following to the terminology of the Reference Manual:

(a) Those defining the elaboration of declarations (Prefix ELAB).

(b) Those defining the evaluation of expressions (Prefix EVAL).

(c) Those defining the execution of statements (Prefix EXEC).

The dynamic semantics is parameterized by:

- an abstract machine that provides a model of storage allocation
- a set of definitions which characterize the restrictions of a concrete machine (minimum and maximum value for integers, etc).

3.4 Treatment of Errors

Some errors may be discovered during normalization and during the evaluation of the Static Semantics. They are reported by inserting a special construct in the abstract syntax tree, at the lowest meaningful level. The Dynamic Semantics is only defined on trees which do not contain such errors. In this way, the place and reason for an error are defined precisely. Since the errors now are part of the formal definition, an opportunity is given to standardize error messages. (Note that it seems more difficult to standardize syntax error diagnostics, because the discovery of syntax errors occurs at different moments with different parsing strategies and it may be unwise to constrain Ada parsers to use a specific parsing technique). Furthermore, since error messages are located in the program tree, the location of the errors will be completely unambiguous, . This facility will be most useful when using sophisticated program editors that know the structure of Ada such as the MENTOR system.[Donzeau-Gouge]

Errors occurring during the execution of a program raise the appropriate exceptions, as prescribed by the semantics of Ada.

REFERENCES

[Donzeau-Gouge] V.Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, J-J. Levy "A structure oriented program editor", Proceedings of the International Computing Symposium, North-Holland Publishing Company, 1975.

[Gordon] M. J. . Gordon, Descriptive Techniques for Denotational Semantics, Springer Verlag, 1979.

[Hennessy-Plotkin] M. Hennessy, G. D. Plotkin, "Full abstraction for a simple parallel programming language", MFCS Proceedings, Sept. 1979

[Huet-Levy] G. Huet, J-J. Levy, "Call-by-need computations in non-ambiguous linear term rewriting systems", Rapport IRIA-Laboria no. 359, August 1979.

[Kahn] G. Kahn (Ed.) Semantics of concurrent computations, Lecture Notes Volume 70, Springer Verlag 1979.

[Luckham-Polak] D. C. Luckham, W. Polak, "Ada Exception Handling: An Axiomatic Approach", Stanford University Artificial Intelligence Laboratory, August 1979.

[Milne-Strachey] R. Milne, C. Strachey, A Theory of Programming Language Semantics, Chapman and Hall, 1976.

[Milner] M. Gordon, R. Milner, C. Wadsworth, "Edinburgh LCF", Computer Science Department, University of Edinburgh, 1978.

[Mosses] SIS- Semantics Implementation System, Reference Manual and User Guide, DAIMI MD-30, University of Aarhus, August 1979.

[Mosses] P. osses, "The Mathematical Semantics of Algol 60", Technical Monograph PRG-12, Oxford University Programming Research Group, January 1974.

[Scheifler] R. W. Scheifler, "A Denotational Semantics of CLU", Master's Thesis, MIT Laboratory For Computer Science, May 1978.

[Stoy] J. Stoy, Denotational Semantics: The Scott-Strachey approach to Programming Language Theory, MIT Press, 1977.

[Tennent] R. D Tennent "A denotational definition of the programming language PASCAL", Programming Research Group Memorandum, Oxford University, April 1978.