INFORMATION PROCESSING 83, R.E.A. Mason (ed.)
Elsevier Science Publishers B.V. (North-Holland)

NAIVE DENOTATIONAL SEMANTICS

Andrzej BLIKLE and Andrzej TARLECKI
Institute of Computer Sciences
Polish Academy of Sciences,
PKiN P.O. Box 22, 00-901 Warsaw, Poland

Invited Paper

The sophisticated mathematical framework of denotational semantics (Scott's reflexive domains and continuations) is not only discouraging for many practitioners but also leads to several technical problems in applications. In this paper we investigate the possibility of developing denotational semantics where semantic domains are just sets and where states rather than continuations are transformed by the program's components. We show that a full mechanism of goto's may be described without continuations and that some procedural mechanisms (e.g. static binding with a hierarchy of procedural parameters - like in PASCAL) do not require reflexive domains. We show further, that if we relax the denotational principle and adopt a copy-rule semantics of procedures, then any procedural mechanism can be described in our framework.

Motto "... the metalanguage of a formal definition must not become a language known to only the priests of the cult"

M.Marcotty et al [18]

1. INTRODUCTION

The Scott-Strachey-Wadsworth [25],[28], [31] denotational semantics has been developed in the early 1970's as a mathematical method of writing rigorous, well-structured and machine-independent definitions of programming languages. The necessity of such definitions was especially felt by people involved in designing and implementing programming languages, writing user-oriented descriptions of programming languages proving program correctness and teaching the theory of programming languages.

Even though today's denotational semantics can be essentially used in all the areas mentioned above, the response from its potential users has been relatively weak. Practically no programming language of the 1970's or 1980's has been designed using denotational semantics. Some languages were given denotational definitions later - (e.g. ALGOL [22],[14], PASCAL [1], CHILL [5] or ADA [8] - but these definitions are still rather experimental and written for a narrow audience of "the priests of the cult". In the program-correctness investigations, nearly all the research has been based on a traditional logic-oriented semantics rather than on Scott's reflexive domains and Strachey-Wadsworth continuations (cf. de Bakker [2] and the references there). Finally, denotational semantics as a framework where to teach the theory of programming languages has been practically forgotten. So far we have texts devoted either to denotational semantics itself, such as Milne and Strachey [20], Stoy [29] or Gordon [13], or to teaching programming language theory at an intuitive level where denotational semantics only helps the author in an elegant presentation of the material but is not shown to the reader, e.g. Tennent [32].

The discrepancy between the potential advantages of denotational semantics and the relatively weak response from the community of practitioners may be only partially explained by the usual psychological barrier associated to a formalization of software engineering. The major factor resposible for this situation seems to be a rather difficult mathematical model of denotational semantics and in particular the technique of continuations and Scott's theory of domains. Especially the latter introduces a great deal of mathematical sophistication frequently discouraging even for professional mathematicians.

In order to make denotational semantics closer to practitioners some authors try to teach it (like Gordon [3]) or to use it (like Bjørner and Jones [6]) in an only half-formal way. They do everything formally, except that Scott's domains are regarded as sets. Since domains - especially the reflexive domains - are rather far from being sets, such an attitude is hardly acceptable. It invites the user of denotational semantics to relax the requirement of formality whenever he wants to do so. This, of course, may lead to inconsistencies. For instance in the Vienna Development Method (VDM) [6], the technical version of denotational semantics, one may write a set of domain equations where operations non-expressible in the Scott model (like $\xrightarrow[m]{}$) appear together with operations which do require that model (like $\rightarrow$). Of course, such equations have no mathematical model at all !

Being aware of the problems which his approach causes to practitioners, D. Scott recently suggested two other models of domains: [26] and [27]. These new models, although mathematically very elegant, are not simpler than the former. In general it seems rather evident that any model which captures the concept of a reflexive domain, i.e. which provides the solvability of the domain equation $D = D \to D$, must involve complex mathematical constructions.

The goal of the present paper is to formalize the non-formal treatment of denotational semantics and to investigate which programming mechanisms can be described without reflexive domains and continuations. We show that procedures with static binding and with a hierarchy of procedural parameters (no procedure can take itself as a parameter whether directly or indirectly) can be described within our framework in keeping the usual denotational style. For instance, we can describe FORTRAN, PASCAL or MODULA but we cannot describe in that way ALGOL 60, LISP and APL. We further show, that if we allow a so called copy-rule semantics where a procedure identifier denotes a procedure text rather than a state-transforming function, then we can describe all known procedure mechanisms hence also these of ALGOL 60, LISP and APL. Of course, it may be argued whether copy rule is "denotational enough". We refrain from commenting on that point, but we wish to mention that copy rule does not violate the structurality of language definition and that it provides an adequate framework for procedure-oriented Hoare logic (cf. Olderog [23]). As to the mechanisms of jumps we show that full goto's can be easily handled without continuations.

The denotational techniques exploited in this paper - like the model of goto's and the semantics of procedures - are well known from the literature and/or folklore. We do not pretend to have originality at this point. All we want to convey to the reader is that a good part (if not all) of the half-formal applied denotational semantics may be formalized without tears (and continuations) in an elementary set theory. In particular, all these VDM domain constructors which are not expressible in the Scott model have simple definitions in our framework.

Mathematically our approach is based on an elementary theory of fixed-point equations in cpo's (Sec. 2). In that a general framework we define some problem-oriented tools: the cpo's of syntactic and semantic domains (Sec. 3) and the cpo of functions (Sec. 4). This allows us to introduce a general model of a direct (i.e. non-continuational) denotational semantics (Sec. 5), to show how to handle goto's in that model (Sec. 6) and how to describe procedures with static binding and a hierarchy of parameters (Sec. 7). The copy-rule semantics is briefly discussed at the end (Sec. 8). The paper has been written for readers who need not be familiar with denotational semantics but who should have some general acquaintance with the mathematical methods used in the theory of programming languages.

2. CPO'S AND FIXED POINTS

In this section we recall a few well-known concepts and facts mainly in order to fix our notation.

A partially ordered set $(A, \sqsubseteq)$ is called a cpo (chain-complete partially ordered set) if it contains the least element $\bot_A$ (or simply $\bot$ if A is understood) and if any increasing enumerable sequence $a_1 \sqsubseteq a_2 \sqsubseteq \sqsubseteq \ldots$ (called a chain) in A has a least upper bound $\bigsqcup_{n=1}^{\infty} a_n$ in A. If A and B are cpo's, then a total function $f : A \to B$ is called monotone, if $a_1 \sqsubseteq a_2$ implies $f(a_1) \sqsubseteq f(a_2)$ and is called continuous, if for any chain $a_1, a_2, \ldots$ in A also $f(a_1), f(a_2), \ldots$ is a chain (in B) and $f(\bigsqcup_{n=1}^{\infty} a_n) = \bigsqcup_{n=1}^{\infty} f(a_n)$.

If $f : A \to A$ is continuous, then there exists the least a such that $f(a) = a$. It is called the least fixed point of f, is denoted by $Y.f$ and satisfies the equation $Y.f = \bigsqcup_{n=1}^{\infty} f^n(\bot)$, where $f^1(\bot) = f(\bot)$ and $f^{n+1}(\bot) = f(f^n(\bot))$.

A cpo is called a set-theoretic cpo if its elements are sets and if they are ordered by inclusion. In our approach we use essentially two classes of basic cpo's:
1. The set-theoretic cpo's of domains
2. The cpo's of functions

Cpo's may be combined together by four operations which we define below. Let $(A, \bot_A)$ and $(B, \bot_B)$ be cpo's.

Disjoint union $(A \cup B, \sqsubseteq)$, where A and B must be (made) disjoint and where we glue $\bot_A$ with $\bot_B$ into $\bot$ and for $c, d \in A \cup B$ we set $c \sqsubseteq d$ if either $c, d \in A$ and $c \sqsubseteq_A d$ or $c, d \in B$ and $c \sqsubseteq_B d$ or $c = \bot$

Cartesian product $(A \times B, \sqsubseteq)$, where $(a_1, b_1) \sqsubseteq (a_2, b_2)$ iff $a_1 \sqsubseteq_A a_2$ and $b_1 \sqsubseteq_B b_2$

Total functions $(A \to B, \sqsubseteq)$, where $f \sqsubseteq g$ if $f(a) \sqsubseteq_B g(a)$ for all $a \in A$.

Continuous functions $(A \xrightarrow[c]{} B, \sqsubseteq)$ with the same ordering as above.

3. THE CPO'S OF DOMAINS

Everywhere in this paper we use the words domain and set as synonyms. For technical reasons we distinguish between syntactic domains such as Identifier, Expression, Command etc. and semantic domains such as Boolean, Integer, Location, State etc. Since in both cases domains may be defined by fixed-point equations, the appropriate cpo's must be established. Both are set-theoretic cpo's.

The cpo of syntactic domains is fairly obvious. We take the alphabet A of the programming language which is being defined and the family $Lan(A) = \{L \mid L \subseteq A^*\}$ of all formal languages over A. This family is a cpo with inclusion. The usual operations in Lan(A) are the union $L_1 | L_2$ and the concatenation $L_1 \frown L_2$ of languages. These operations are continuous and allow for the fixed-point definitions of all (and only) context-free languages (see Blikle [9] for details). For convenience we can add L^*, L^+, $L_1 \cap L_2$ which are also continuous and $L_1 - L_2$ which is continuous in L_1 but is not in L_2.

The cpo of semantic domains is constructed essentially in a similar way although now we provide more operations on domains and the argument about the existence of the appropriate family of domains is more subtle. We start by establishing a family $\mathbb{B}$ of basic domains which - analogously to the alphabet A - depends on the programming language which is being defined. Typically $\mathbb{B}$ will contain such domains as Identifier, Location, Boolean, Integer, Real, etc. The only general requirement about $\mathbb{B}$ is that it contains at least one infinitely enumerable set. Now, by analogy to syntactic domains we take as our cpo of semantic domains the set of "all sets over $\mathbb{B}$". This idea is formalized by taking an arbitrary universal set which contains $\mathbb{B}$. A set is called universal (Cohn [11]) if it has the following four properties:

1) If $A \in U$, then $A \subseteq U$,
2) If $A \in U$, then $2^A \in U$,
3) If $A, B \in U$, then $\{A,B\} \in U$
4) If $I \in U$ and $F_i \in U$ for any $i \in I$, then $\bigcup_{i \in I} F_i \in U$.

By an axiom of set theory, every set is an element (hence also a subset) of some universal set. It is also easy to prove (Cohn [11]) that every universal set is closed under all usual set-theoretic operations such as $A | B$ (union); $A \cap B$, $A \times B$, 2^A, $A \to B$, etc. Additionally, if a universal set contains an infinitely enumerable element, then it is also a set-theoretic cpo.

The concept of a univeral set allows for the construction of a cpo of domains over any family of basic domains. Such a cpo is closed under all set-theoretic operations used in the denotational semantics. Below we give a typical list of such operations. Many of them are known from META-IV, the definitional metalanguage of VDM (Bjørner and Jones [6]).

1) $A | B$ - union
2) $A_1 \times \ldots \times A_n$; $n \geq 1$ - Cartesian product (non-associative)
3) $A^* = \bigcup_{n=0}^{\infty} A^n$ - star-operation
4) $A \overset{\sim}{\underset{m}{\to}} B$ - the set of all mappings from A to B, i.e. the set of all partial functions $f : A \overset{\sim}{\to} B$ such that f(a) is defined for only a finite number of arguments a.
5) A-finset - the set of all finite subsets of A
6) $A \underset{pm}{\to} B | e$ - the set of pseudo-mappings from A to B, i.e. the set of all total functions $f : A \to B | \{e\}$ such that $e \notin B$ and $f(a) \neq e$ for only a finite number of arguments.
7) $A - B$ - difference
8) A^ω - the set of all finite and infinite sequences over A
9) $A \overset{\sim}{\to} B$ - the set of all partial functions from A to B
10) $A \to B$ - the set of all total functions from A to B
11) $A \underset{c}{\to} B$ - the set of all continuous functions from A to B provided that A and B are cpo's.

The last operation requires some comments. Formally this is a four argument partial operation which takes two sets A and B and two binary relations R_1 and R_2 and if (A,R_1) and (B,R_2) are cpo's, then it returns the set of all continuous functions from (A,R_1) to (B,R_2). Otherwise the operation is undefined. Normally the orderings are clear from the context and therefore we write simply $A \underset{c}{\to} B$. This will be seen later in examples.

Similarly to the case of syntactic domains, also here not all domain operations must be continuous. And in fact not all of them are:

(i) 1)-5) are continuous in all arguments
(ii) 6) is continuous in B and 7) is continuous in A
(iii) 8)-11) are non-continuous in all arguments.

According to these properties, 1)-5) can be used in arbitrary recursive definitions of domains, 6) and 7) only if A

resp. B are constant and 8)-11) can be used only in a non-fixed-point framework. For instance the following fixed-point set of equations is legal:

$$\begin{aligned}
&\text{Data} = \text{Bool} \mid \text{Int}\\
&\text{File} = \text{Data} \mid [\text{Data} \times \text{File}]\\
&\text{Record} = \text{Ide} \xrightarrow[pm]{} [\text{Data} \mid \text{Record}] \mid \text{UNBOUND}\\
&\text{Value} = \text{Data} \mid \text{Record} \mid \text{File}\\
&\text{State} = \text{Ide} \xrightarrow[pm]{} \text{Value} \mid \text{UNUSED}
\end{aligned}$$

but if we introduce the domain of procedures and accordingly modify the concept of a state:

$$\begin{aligned}
&\text{State} = \text{Ide} \xrightarrow[pm]{} [\text{Value} \mid \text{Proc}] \mid \text{UNUSED}\\
&\text{Proc} = \text{State} \tilde{\to} \text{State}
\end{aligned}$$

then our set of equations becomes illegal (unsolvable) due to the recursive use of $\tilde{\to}$ between State and Proc.

The recursive use of $\tilde{\to}$ or $\to$ in domain equations will be referred to as heavy recursion. Heavy recursion is forbidden in our framework but possible in the Scott approach. In the latter case, however, the operations 4),5),6), 7),9) and 10) become non-expressible (cf. the remarks of J.E.Stoy [30] on the formal semantics of META-IV).

In the sequel domain equations will be written in the form

$$d : D = F(D_1, \ldots, D_n)$$

where the prefix "d:" means that a typical element of D is denoted by d, possibly with indices. We also allow curried-function's domains to have polymorphic types, e.g. $A \to [B \tilde{\to} [C \xrightarrow[m]{} D]]$ which we usually write without parentheses as $A \to B \tilde{\to} C \xrightarrow[m]{} D$. We assume the following hierarchy of priorities of domain operations: $\times, \mid, \xrightarrow[\alpha]{}$ and then all the others with equal priorities.

In the context of functional domains we use ":" for "$\in$", e.g. we write $f : A \xrightarrow[m]{} B$ to say that f is a mapping from A to B.

4. THE CPO'S OF FUNCTIONS

In our approach we use three types of cpo's of functions:

1. the cpo's of partial functions $(A \tilde{\to} B, \subseteq)$ with the set-theoretic ordering by inclusion; we call this ordering horizontal.
2. the cpo's of total functions $(A \to B, \sqsubseteq)$ with the argumentwise ordering defined in Sec. 2 where B must be a cpo; we call this ordering vertical.
3. the cpo's of continuous functions $(A \xrightarrow[c]{} B, \sqsubseteq)$ with vertical ordering where A and B must be cpo's.

In the remainder of this section we define basic operations on functions. First we set some notational conventions. We write f.x for f(x) and $f.x_1.x_2.\ldots.x_n$ for $(\ldots((f.x_1).x_2)\ldots).x_n$ if f is curried. We write f.x = ! and f.x = ? to say that f is defined, resp. is not defined, in x. We write $b \to c,d$ for if b then c else d. Of course b must be a boolean value tt (true) or ff (false). We write

$$\begin{aligned}
&b_1 \to c_1,\\
&\ldots\\
&b_n \to c_n,\\
&\text{TRUE} \to c_{n+1}
\end{aligned}$$

for $b_1 \to c_1, (b_2 \to \ldots \to c_{n-1}, (b_n \to c_n, c_{n+1})\ldots)$. TRUE may be equivalently replaced by not b_n. We also use standard λ-notation, e.g. $\lambda x.x+a$ denotes - for every fixed a - a function of x "add a to x" and $\lambda a.\lambda x.x+a$ denotes a function of a which for any argument a returns a function of x. For instance $(\lambda a.\lambda x.x+a).1 = \lambda x.x+1$. Below we define four operations on functions.

1) Sequential composition. If $f : A \tilde{\to} B$ and $g : B \tilde{\to} C$, then $f \circ g : A \tilde{\to} C$ where

$$(f \circ g).a = (f.a = ?) \to ? \ , \ g.(f.a)$$

Notice that in $f \circ g$ we first evaluate f and then g.

2) Conditional composition. If $p : A \tilde{\to} B$ and $f,g : A \tilde{\to} C$, then IF p THEN f ELSE g FI $: A \tilde{\to} C$ where

$$\begin{aligned}
&\text{IF p THEN f ELSE g FI} \ . \ a =\\
&\ p.a = tt \to f.a,\\
&\ p.a = ff \to g.a,\\
&\ \text{TRUE} \quad \to ?
\end{aligned}$$

In applications tt and ff normally belong to B but we do not require this in definition. If A = C and g.a = a for all $a \in A$, then we use the shorthand IF p THEN f FI.

3) Overwriting. If $f,g : A \tilde{\to} B$ then $f[g] : A \tilde{\to} B$, where

$$f[g].a = (g.a = !) \to g.a \ , \ f.a$$

4) Restriction. If $f : A \tilde{\to} B$ and $C \subseteq A$ then $(f \mid C) : A \tilde{\to} B$, where

$$(f \mid C).a = (a \in C) \to f.a \ , \ ?$$

As is easy to prove, 1) and 2) are continuous in all arguments whereas 3) and 4) only in f. Moreover 1) and 3) are associative.

Since mappings and total functions (between sets) are regarded as particular cases of partial functions all four operations remain applicable in these cases too. For pseudomappings we apply essentially only 3) and 4) where we assume that the undefinedness element e replaces ?.

Our operations may be applied to total functions from a set A into a cpo B. In that case "?" should be replaced by the appropriate least element in all four definitions. The new operations have analogous continuity and associativity properties as in the former case. Notice that continuity is related now to the vertical ordering.

Operation 1) preserves the continuity of its arguments and therefore may be used in the cpo of continuous functions. Operation 2) has the same property whenever B is a flat domain. Operations 3) and 4) do not preserve continuity.

The overwriting operator can be generalized - in an obvious way - to the case where $f : A \tilde{\to} B$, $g : C \tilde{\to} D$ and $f[g] : A|C \tilde{\to} B|D$. This gives rise to a new domain operator. For $Fun_1 \subseteq A \tilde{\to} B$ and $Fun_2 \subseteq C \tilde{\to} D$ we set

$$Fun_1 \; ! \; Fun_2 = \{f_1[f_2] \mid f_i \in Fun_i\}$$

Since this operator is not applicable to all domains we cannot talk about its continuity. However, we get a continuous operation if we combine ! with continuous functional operations, e.g. as in $(A \tilde{\to}_m B) ! (C \tilde{\to}_m D)$.

To complete this section we introduce a special notation for mappings and pseudomappings. If $a_1,\ldots,a_n \in A$, $b_1,\ldots,b_n \in B$ and $a_i \neq a_j$ for $i \neq j$, then $[b_1/a_1,\ldots,b_n/a_n] : A \tilde{\to}_m B$ where

$$[b_1/a_1,\ldots,b_n/a_n].a = \begin{array}{l} a = a_1 \to b_1, \\ \ldots \\ a = a_n \to b_n, \\ \text{TRUE} \to ? \end{array}$$

For pseudomappings we use the same notation with the obvious alteration of the meaning. The undefinedness element e is not explicit in the notation but is usually understood from the context.

5. DIRECT SEMANTICS; A GENERAL MODEL

In this and in the three subsequent sections we show how to construct a direct (i.e. non-continuation) semantics in our framework. This section gives some general preliminaries; in the following sections we discuss jumps and procedures.

We distinguish five basic syntactic domains:

```
x    : Ide  - identifiers
exp  : Exp  - expressions
dec  : Dec  - declarations
com  : Com  - commands
prog : Prog - programs
```

For the sake of generality we do not fix these domains. We only assume that every program is a block, that blocks are commands of the form begin dec com end and that blocks may be nested. We assume further that declarations may appear only at the beginning of a block and that commands are closed under if and while. So far, we introduce neither goto's nor procedures. We discuss these mechanisms in the next three sections.

Our semantic domains are the following:

```
d   : Data      = Bool|Int|Real|File ...
l   : Loc       = {l1,l2,...}
                              (locations)
v   : Value     = Data|Loc
env : Env       = Ide -pm-> Value|UNBOUND
                              (environments)
      Store-lo  = Loc -pm-> Data|UNUSED
                              (location stores)
      Store-io  = {IN,OUT} -> Data*
                              (input/output stores)

sto : Store     = Store-lo ! Store-io
sta : State     = Env × Store
```

The domains of data and locations do not need comments. Values (denotable values) are the only objects which can be bound to identifiers in the environment. Later they will also contain procedures. Environments are pseudomappings which bind values to identifiers (UNBOUND is the undefinedness element). Stores are pseudomappings which bind data to locations and the strings of data (the input file and the output file) to special reserved locations IN and OUT. We assume that IN and OUT do not belong to Loc. Each state consists of an environment and a store.

Now, we define four semantic functions:

```
E : Exp  -> State ~> Data.
C : Com  -> State ~> State
D : Dec  -> State ~> State
P : Prog -> Data* ~> Data*
```

Their types reflect our state-to-state ideology which - in our opinion - benefits especially in the program-correctness considerations. Of course commands modify only the store and declarations modify either the environment or the environment and the store.

Below we give a few examples of semantic clauses. Some of them will be referred to in the sequel.

$$C[com_1;com_2] = C[com_1] \circ C[com_2]$$

C[if exp then com_1 else com_2 fi] =
IF E[exp] THEN $C[com_1]$ ELSE $C[com_2]$ FI

C[while exp do com od] =
IF E[exp]
THEN C[com] ∘ C[while exp do com od] FI

C[begin dec com end].(env,sto) =
(env,[D[dec] ∘ C[com].(env,sto)]$_2$)
where $[(x_1,x_2)]_i = x_i$ for $i = 1,2$.

P[begin dec com end].d-string =
[C[begin dec com end].(in-env,
in-sto)]$_2$.OUT

where

in-env = [ ]
in-sto = [d-string/IN, ( )/OUT]

The first three rules are self explanatory. The denotation of a block passes the input environment unmodified and then calculates the output store in three steps:
1. the input state is modified by the declaration which creates the local environment of the block,
2. the modified state is passed to the command of the block which modifies its store,
3. the store of the resulting state is passed to the output.

The denotation of a program given an input data string first creates the input state. The input environment is empty, i.e. associates UNBOUND to all identifiers. The input store stores the input file in IN, the empty file in OUT and UNUSED in all other locations. The input state is executed by the block of the program and upon termination the content of OUT in the output store is returned.

The reader has probably noticed that in the informal comments to denotational definitions we have adopted a simplified wording. For instance, we say that a state is "modified by a command" rather than "transformed by the denotation of a command". We use this convention throughout the paper since it is rather intuitive and its formal meaning is normally clarified by a denotational definition.

6. JUMPS: BRANCHINGS, EXITS AND GOTO'S

In this section we discuss three successively more general mechanisms of jumps and show how to incorporate them into our direct denotational semantics. The first mechanism corresponds to jumps that do not interrupt the execution of structured commands. Such jumps are called branchings and are typical for low-level programming languages. The second mechanism corresponds to jumps which do interrupt the normal execution of structured commands but which always shift the control forwards in the program and allow only exiting (not entering) structured commands. Such jumps are called exits and correspond to escapes, returns, exception raisings, ect. The third mechanism describes arbitrary jumps - forwards and backwards, in and out - and is referred to as goto's.

For simplicity we describe all the three mechanisms within the standard syntax of goto's limiting in each case the contextual rules of putting labels into commands. We assume that with every $i \in Num = \{1,2,...\}$ and $com \in Com$, both goto i and i : com are commands.

In the case of goto's used as branchings we can use a modification of a simple model described by A.Mazurkiewicz [19]. In that model every body of a block is regarded (or rewritten) as a sequence of labeled commands of the form

$1 : com_1 ;$ if exp_1 then goto t.1 else goto e.1 fi ;
...
$n : com_n ;$ if exp_n then goto t.n else goto e.n fi ;
n+1 : stop

where com_i's contain neither goto's nor (visible) labeled commands and where t and e are functions which map $\{1,...,n\}$ into $\{1,...,n+1\}$. Now, with every label i we associate a function T_i : State $\tilde{\to}$ State which describes the transition of states between i and n+1. This is, of course, a prototype of a continuation associated to i in the standard denotational semantics. It is easy to define T_i's operationally and then it is equally easy to prove (Blikle [10]), that the (n+1)-tuple $(T_1,...,T_{n+1})$ is the least solution of the following fixed-point set of equations:

$$T_1 = C[com_1] \circ \underline{IF}\ E[exp_1]\ \underline{THEN}\ T_{t.1}\ \underline{ELSE}\ T_{e.1}\ \underline{FI}$$
...
$$T_n = C[com_n] \circ \underline{IF}\ E[exp_n]\ \underline{THEN}\ T_{t.n}\ \underline{ELSE}\ T_{e.n}\ \underline{FI}$$
$$T_{n+1} = I$$

where I is the identity function over State. Since T_1 is the state transition between 1 and n+1, this set of equations may be regarded as a denotational definition of our block.

The second mechanism - the exits - corresponds to the case where goto's may appear within structured commands but where two contextual restrictions must be observed.:

1. goto i alway precedes i : com,
2. if i : com appears inside a structured command, then so does goto i.

In other words, exits shift the control forwards and allow exiting, but not entering, structured commands.

The mechanism of exits is described by splitting the set of states into executing states and searching states. If an executing state enters a non-goto command, then it is transformed in the usual way. If it enters goto i, then it is marked by i and becomes a searching (for i) state . All commands which follow, and which are not labeled by i are transparent for that state. The (first) command labeled by i removes the tag i from the state and trans-

forms the state in an ordinary way.

In order to describe our model formally, we slightly modify the concept of the store by setting:

```
Store-ju = {JUMP} → Num|NIL
Store    = Store-lo ! Store-io ! Store-ju
```

In other words stores now have one more special reserved location JUMP which binds either a label or a special element NIL. A state with NIL in JUMP is called an executing state. A state with i in JUMP, is called a searching state (searching for i). In order to dynamically distinguish between executing and searching states we introduce two predicates on states:

```
executing : State → Bool
searching : State → Bool
```

defined in an obvious way. Now, we extend and modify our former semantics:

```
C[goto i].(env,sto) =                                   (1)
 sto.JUMP = NIL → (env,sto[i/JUMP]),
 TRUE           → (env,sto)

C[i : com].(env,sto) =                                  (2)
 sto.JUMP = i   → C[com].(env,sto[NIL/JUMP])
 sto.JUMP ≠ NIL → (env,sto)
 TRUE           → C[com].(env,sto)

C[com1 ; com2] = C[com1] ∘ C[com2]                      (3)
```

For all other cases, i.e. if com is either a primitive command (I/O, assignment, etc.) or a structured command (while, block, procedure call, etc.) we set

C[com] = IF executing THEN as before FI

The last mechanism which we describe in this section corresponds to jumps which may lead forwards and backwards in the program and which may be used to exit arbitrary structured commands and to enter (jump into) while and if. Formally we could extend this mechanism to jumping into blocks, but since this has little practical sense we omit that case.

The third model is a modification of the second. We take the same domain of states and we keep the clauses (1), (2) and (3) unchanged. We modify the semantics of if, while and block.

```
C[if exp then com1 else com2 fi] =
 IF executing
 THEN
 IF E[exp] THEN C[com1] ELSE C[com2] FI
 ELSE
 C[com1] ∘ IF searching THEN C[com2] FI
FI
```

In the executing mode we evaluate exp and then proceed either to the execution of com_1 or to the execution of com_2 according to the usual rule. In the searching mode we do not evaluate exp but first search in com_1 and then, if we still remain in the searching mode, we search in com_2.

```
C[while exp do com od] =
 IF executing
 THEN
 IF E[exp]
 THEN C[com] ∘ C[while exp do com od]FI
 ELSE
 C[com] ∘ IF executing
          THEN C[while exp do com od]FI
FI
```

In the executing mode we proceed as usual. In the searching mode we enter directly the body com and then, if we find the appropriate label, we remain in the loop.

So far our semantics handles only jumps forwards. In order to allow jumps backwards we have to modify the semantics of blocks. Informally speaking, a searching state which arrives at the end of a block must be returned to the beginning of the block if it contains a label which appears in that block. This requires a recursion in the semantics of blocks and the introduction of a new function. Let

J : Com → Num-finset

associate with every command the set of all labels visible in that command. The definition of J is routine (structural induction) and therefore we omit it. The clause for blocks is now as follows:

```
C[begin dec com end] =
 IF executing
 THEN λ(env,sto).
 (env,[(D[dec]∘C[com]∘Loop).(env.sto)]2)
 FI
```

where $[(env,sto)]_2 = sto$ and

```
Loop =
 IF λ(env,sto).(sto.JUMP∈J[com] → tt,ff)
 THEN C[com] ∘ Loop
 FI
```

The function Loop describes the recursive mechanism which passes back to the block body any state searching for a label declared in that body. Notice that the iteration in the block does not capture the declaration dec which is elaborated only once.

To complete this section a few general comments are in order. In the intuitive explanations of our clauses we have been using an "operational" wording: a state searches for a label, is tested, is returned back or passed forward etc. This supprts our intuition, but should not be understood as a description of an intended implementation of the language. Our semantics is user- and proof-oriented (cf. Blikle [10]) and therefore our main concern is to say what programs are doing rather than how we execute them.

It is frequently emphasised that one of the main advantages of denotational semantics is the possibility of describing the what without entering into the how. This opinion is, of course, true but must not be oversimplified by saying that how is totally beyond the reach of denotational semantics. Whenever we define a function - e.g. a denotation of a command - we use a meta expression which always describes some how. That how may indicate - or not - the intended implementation. For instance, our main concern in the description of goto's was to make the definition possibly transparent and simple. Our main concern was what and therefore our how, if implemented, would be very time consuming. In contrast to that, the VDM models of goto's (Jones [15] and Bjørner [4]) are much closer to the implementation. Their how's are much more "optimal" than ours, but their what's seem, in turn, less transparent.

7. HIERARCHICAL PROCEDURES WITH STATIC BINDING

By a static binding in procedures we mean that at the invocation time, the body of a procedure is executed in the declaration-time (i.e. static) environment and the invocation-time (i.e. dynamic) store. Procedures with static binding may be represented by curried functions of the form

$$Proc = Par \tilde{\to} Store \tilde{\to} Store$$

where Par denotes the domain of parameters (for simplicity we assume that each procedure takes only one parameter). Since parameters may be either locations, or data or procedures, we have another equation

$$Par = Loc | Data | Proc$$

This leads to heavy recursion in domain equations and therefore is illegal in our model. In order to avoid such recursion we have to restrict the class of definable languages by requesting a hierarchy in the domain of procedures: procedures of degree 0 accept only data and locations as parameters; procedures of degree $n \geq 1$ accept only procedures of degree $n-1$. Of course, our hierarchy requirement is not satisfied by all languages - ALGOL-60 is a typical example - but still the class of definable languages seems of interest since it contains such languages as, for instance, PASCAL or MODULA.

Below we give a set of domain equations which describe our hierarchy. For simplicity we assume that non-procedural parameters are of variable (reference) type only and we shift the discussion of goto's to the end of the section. In order to handle recursion we assume that all procedures of degree $n \geq 1$ are continuous. The continuity is defined with respect to the horizontal ordering in $Store \tilde{\to} Store$ and the vertical ordering in $Proc_n$.

$$\begin{aligned}
&Data = \text{as in Sec. 5}\\
&Loc = \text{as in Sec. 5}\\
&Store = \text{as in Sec. 5}\\
&Value = Data | Loc | Proc\\
&P: Proc = \cup_{n=1}^{\infty} Proc_n\\
&\quad Proc_0 = Par_0 \to Store \tilde{\to} Store\\
&par: Par_0 = Loc\\
&\quad Proc_n = Par_n \vec{c} Store \tilde{\to} Store \qquad n = 1,2,\ldots\\
&par: Par_n = Proc_{n-1} \qquad n = 1,2,\ldots\\
&\quad Env = Ide \overset{\to}{pm} Value | UNBOUND\\
&\quad State = Env \times Store
\end{aligned}$$

For further investigations we have to introduce some syntax for procedure declarations and calls. Its particular form is, of course, only a technical matter. What is really important is the possibility of recognizing the degree of a procedure from its declaration. This establishes the required hierarchy in the language.

A procedure declaration is of the form proc p(fp : n)com where p, fp and com are the procedure name, formal parameter and body and where $n \geq 0$ indicates the degree of the parameter.

A procedure call is of the form p(ap) where ap is an actual parameter.

Now, we are ready to define the semantics of procedure declarations and calls. We assume that the types of the semantic functions D and C remain as they were defined in Sec. 5.

The semantics of procedure declarations

$$D[\text{proc } p(fp{:}n)\ com].(s\text{-}env, s\text{-}sto) = (s\text{-}env[P/p], s\text{-}sto).$$

A procedure declaration modifies the state by binding the declared procedure P to its name p in the environment. The argument of a procedure declaration is referred to as a static state and is denoted by (s-env,s-sto) in order to explicitly distinguish it from the argument of a procedure call which is referred to as a dynamic state and is denoted by (d-env,d-sto). The procedure P is a function

$$P : Par_n \vec{c} Store \tilde{\to} Store$$

defined by the fixed-point equation

$$P = \lambda par.\lambda d\text{-}sto.[C[com].(s\text{-}env[par/fp, P/p], d\text{-}sto)]_2 \qquad (4)$$

where $[(env,sto)]_2 = sto$. Notice that the body com of the procedure gets the static environment modified by sending par to fp (parameter passing) and P to p (recursion). This corresponds to

the mechanism of static binding. The recursion described here corresponds to the simple case where a procedure calls itself recursively. Mutual recursion of many procedures can be described by an appropriate fixed-point set of equations.

The fixed-point definition of P requires the continuity in P of the right-hand side of eq. (4). The proof of that fact is quite routine and proceeds by structural induction. In the proof we use the fact that procedures are continuous in their parameters. This is necessary when considering an inner call of the form r(p), where r is a global procedure in the body of p, or when considering an inner recursive declaration of a procedure q which contains in its body a call of the form fp(p) or p(q).

The semantics of procedure calls

```
C[p(ap)].(d-env,d-sto) =
 let P = d-env.p and par = d-env.ap
 p ∉ Proc  or  par ∉ dom.P        → ?,
 d-sto ∉ dom.(P.par)              → ?,
 TRUE          → (d-env, P.par.d-sto)
```

where dom.f denotes the domain of the function f. A procedure call takes the procedure P bound to p in the environment, applies it to the value of the actual parameter and then applies the resulting function to the store. In that way we get a new store. The environment remains unchanged. It is understood that the checks of the conditions $P \notin Proc$ and $par \notin dom.P$ are effectuated at compile time, i.e. that they should belong to static semantics. The last condition essentially means that $par \in Par_n$ and $P \in Proc_n$ for some $n \geq 0$.

In our semantics of procedures we have omitted goto's. This mechanism can be easily incorporated by using the model of searching states described is Sec. 6. We can also allow that labels are passed as parameters. In that case labels must become denotable values (i.e. values attachable to identifiers in the environment) and the semantics of the goto command must be appropriately modified.

8. DYNAMIC BINDING AND NON-HIERARCHICAL PROCEDURES

By dynamic binding we mean that at the invocation time the body of a procedure gets the dynamic environment. In that case

$$Proc_n = Par_n \xrightarrow[c]{} State \xrightarrow{\sim} Store \qquad (5)$$
$$State = Env \times Store$$
$$Env = Ide \xrightarrow[pm]{} [Data|Loc|Proc]|UNBOUND$$

which, of course, leads to heavy recursion. This means that (full) dynamic binding is not definable in the model of Sec. 7. On the other hand some partial dynamic binding can be handled. For instance, if we assume that recursion is elaborated statically, i.e. that the corresponding fixed-point is computed in the static environment, and if we introduce a hierarchy of procedures and environments for handling procedure parameters and procedure globals, then we can still construct a semantics in the formerly described style. This requires the following domain equations:

$$Proc_n = Par_n \xrightarrow[c]{} State_n \xrightarrow{\sim} Store$$
$$State_n = Env_n \times Store$$
$$Env_n = Ide \xrightarrow[pm]{} [Data|Loc|Proc_{n-1}]|UNBOUND$$

Further extension of the class of languages definable in our approach is impossible unless we make a major change in our denotational principle. It is well known (Landin [17]) that full dynamic binding (including recursion) with non-hierarchical procedures can be easily handled if we assume that a procedure declaration sends the procedure text, rather than its denotation, to the environment. Formally this leads to the following definition of procedures (the other equations of (5) remain the same)

```
Proc = Ide × Com
D[proc p(fp) com].(env,sto) =
(env[(fp,com)/p],sto)
C[p(ap].(env,sto) =
(env,[C[com].(env[env.ap/fp].sto)]2)
```

where env.p = (fp,com)

Informally speaking, a procedure declaration sends the formal parameter and the body to the environment and a procedure call first modifies the current state by passing the parameter and then applies the modified state to the denotation of the procedure body. That semantics has been known for years as a copy-rule semantics and has been used at the intuitive level in the definitions of almost all programming languages from ALGOL-60 to ADA. As a descriptive means it is strong enough to give semantics to Church's λ-calculus (Landing [17]), hence to handle practically any scheme of recursion. In applications it is more convenient for dynamic than for static recursion, although the latter can also be described with some extra technical effort. Copy-rule semantics has been also proved useful, and convenient, in the development of Hoare-like logic for recursive procedures with procedural parameters (Olderog [23]).

Formally speaking a copy-rule semantics is not quite a denotational semantics since it mixes syntax with semantics and the triple of semantic functions (E, D, C) is no more a many-sorted homomorphism [12]. On the other hand it is "nearly" denotational since it violates the denotational principle only for procedure declarations and calls. All the other

clauses remain unchanged. It also provides the same structuralization of language definition and the same set of useful concepts to talk about a programming language such as location, environment, store, state, etc. Finally, it is mathematically as sound as the other and being simpler it is certainly much safer for many practitioners.

An objection which is sometimes raised against the copy-rule semantics concerns the property of full abstractness. A semantics is called fully abstract (Milner [21]) if any two program's components (e.g. expressions, commands or declarations) which give the same effect in any programming context have the same semantic denotation. In other words, a semantics is fully abstract if contextual equivalence implies denotational equivalence. In a fully abstract semantics, if we prove the denotational non-equivalence of two program components, then there must be a programming context where these components lead to different effects.

The copy-rule semantics is, of course, "highly" non-fully-abstract, but whether this is a real disadvantage is a matter of taste. In fact, no standard denotational semantics of a language without parallelism can be made fully abstract (Plotkin [24]). For instance, the standard Scott-Strachey denotational semantics of ALGOL-like languages is never fully abstract. On the other hand, the trivial semantics which maps identically syntax to syntax is, of course, fully abstract.

ACKNOWLEDGEMENTS

The earlier versions of this paper were presented at informal seminars in Warsaw, Aachen, St.Augustin (Bonn), Stuttgart, Saarbruecken, Muenchen, Linköping and at a meeting of IFIP's W.G.2.2 in Garmisch-Partenkirchen. In all these places our paper benefited from discussions, comments and criticism. We also wish to express our special thanks to Dines Bjørner and Peter Mosses who have read the early versions of the paper and conveyed many relevant remarks.

REFERENCES

{1} D.Andrews and W.Henhalp, PASCAL, in [7].
{2} J.de Bakker, Mathematical theory of program correctness, Prentice Hall International 1980.
{3} D.Bjørner, ed. Abstract software specification (1979 Copenhagen Winter School Procedings), LNCS 86, Springer Verlag 1980.
{4} D.Bjørner, Experiments in block--structured goto language modeling: exits versus continuations, in: [3].
{5} D.Bjørner and P.Haff, CHILL-formal definition, Danish Datamation Centre, Lyngby 1980.
{6} D.Bjørner and C.B.Jones eds. The Vienna development method: The meta language, vol.61, Springer Verlag 1978.
{7} D.Bjørner and C.B.Jones, Formal specification & software development, Prentice Hall International 1982.
{8} D.Bjørner and O.N.Oest, eds. Towards a formal description of ADA, LNCS 98, Springer 1980.
{9} A.Blikle, Equational languages, Inform. Contr. 21(1972), 134-147.
{10} A.Blikle, An analytic approach to the verification of iterative programs, in: Information Processing 77 (Proc. IFIP Congres 1977, B. Gilchrist ed.) North Holland 1977, 285-290.
{11} P.M.Cohn, Universal algebra, D. Reidel Publishing Company 1981.
{12} J.A.Goguen and J.W.Thatcher and E.G.Wagner and J.B.Wright, Initial algebra semantics and continuous algebras, Journal ACM, 24(1977), 68-95.
{13} M.J.C.Gordon, The denotational description of programming languages, Springer Verlag, New York, Heidelberg, Berlin 1979.
{14} W.Hengalp and C.Jones, ALGOL-60, in [7].
{15} C.B.Jones, Denotational semantics of goto: an exit formulation and its relation to continuations. In: [6].
{16} C.B.Jones, The meta-language: a reference manual. In: [6].
{17} P.Landin, The mechanical evaluation of expressions, BSC Computer Journal, 6(1964), 308-320.
{18} M.Marcotty and H.F.Ledgard and G.W. Bochman, A sampler of a formal definition, Comp. Surveys 8(1976), 191-276.
{19} A.Mazurkiewicz, Iteratively computable relations, Bull. Acad. Polon. Sci., Sér. Sci. Math. Astr. et Phys. 20 no 9(1972), 793-797.
{20} R.Milne and Ch.Strachey, A theory of programming language semantics, Chapman and Hall, London 1977.
{21} R.Milner, Fully abstract semantics of typed λ-calculi, Theoretical Computer Science, 4(1977), 1-22.
{22} P.D.Mosses, The mathematical semantics of ALGOL-60, Technical Monograph PRG-12, Oxford University, 1974.
{23} E.R.Olderog, Sound and complete Hoare-like calculi based on copy rules, Acta Informatica 16(1981), 161-197.
{24} G.D.Plotkin, LCF considered as a programming language, Theoretical Computer Science 5(1977), 223-255.
{25} D.Scott, Data types as lattices, SIAM Journal on Computing, vol 5 (1976), 522-587.
{26} D.Scott, Domain equations, Proc. 6th IBM symp. on Mathematical Foundations of Computer Sciences, IBM Japan 1981.

{27} D.Scott, Domains for denotational semantics, Proc. ICALP 82 (M.Nielsen; E.M.Schmidt eds), pp. 577-613, Springer-Verlag 1982.

{28} R.Scott and Ch.Strachey, Toward a mathematical semantics for computer languages, Technical Monograph PRG-6, Oxford University, August 1971. Also in: Computer and Automata (ed. J.Fox), Microwave Res. Inst. Symposium vol. XXI, Polytechnic Institute of Brooklyn Press, 1971.

{29} J.E.Stoy, Denotational semantics: The Scott-Strachey approach to programming language theory, The MIT Press, Cambridge Mass. 1977.

{30} J.E.Stoy, Foundations of denotational semantics, in: [3] and [7].

{31} C.Strachey and C.P.Wadsworth, Continuations, a mathematical semantics for handling full jumps, Technical Monograph PRG-11, Oxford 1974.

{32} R.D.Tennent, Principles of programming languages, Prentice-Hall International 1981.