THE SEMANTICS OF NONDETERMINISM

M. Hennessy
E.A. Ashcroft

Department of Computer Science
University of Waterloo
Waterloo, Ontario

CS-76-17
April 1976
THE SEMANTICS OF NONDETERMINISM

This paper is concerned with the problems encountered in defining the semantics of nondeterministic algorithms. A nondeterministic control structure is added to the typed \(\lambda\)-calculus and the usual operational semantics for the deterministic language is generalised to take into account the more complex behaviour of nondeterministic algorithms. Then a mathematical model is given for the language and the relationship between the denotational and operational semantics is explored.

1. INTRODUCTION

1.1 Various authors, for example, Plotkin (1975), Vuillemin and Courcelle (1974), Wadsworth (1975), have investigated the problem of providing a mathematical semantics which coincides with the operational semantics of a given deterministic language. In this paper we introduce into one such language a simple choice mechanism and attempt to define a mathematical semantics which adequately models the more complicated operational behaviour of the resulting language.

As an example of such a nondeterministic language take the recursive definitions of Vuillemin and Courcelle (1974) with a call-by-name evaluation mechanism, and add the syntactic entity 'or'. Thus using the notation of elementary number theory we can form equations such as

\[
F_1 \langle N, Y \rangle \Leftarrow \text{if } N \mid Y \text{ then } F_2 \langle 2, Y+1 \rangle \text{ else if } N > \lfloor Y/2 \rfloor \text{ then } (Y \equiv F_2 \langle 2, Y+1 \rangle) \text{ else } F_1 \langle N+1, Y \rangle.
\]

The usual evaluation mechanism can be extended to terms involving 'or' by allowing the term \((T_1 \equiv T_2)\) to evaluate to \(T_1\) or \(T_2\). Each such equation then determines an algorithm which, when applied to an element of some datatype (in this case the integers), is in general capable of producing many data elements and possibly also of diverging. For example, when the above algorithm is applied to \(\langle 2, 2 \rangle\) any prime number can be produced.
This complicates the operational semantics of the language. In the deterministic case it is sufficient to say that two algorithms are operationally equivalent if the input–output behaviour of each coincide. But consider

\[ F_2^{\langle X \rangle} \leftarrow \text{if } X = 0 \text{ then } 0 \text{ else } (X \lor F_2^{\langle X-1 \rangle}) \]
\[ F_3^{\langle X \rangle} \leftarrow \text{if } X = 0 \text{ then } 0 \text{ else } (X \lor F_3^{\langle X-1 \rangle}) \lor G^{\langle X \rangle} \]
\[ G^{\langle X \rangle} \leftarrow G^{\langle X+1 \rangle} \]

Then \( F_2 \) and \( F_3 \) exhibit the same input–output behaviour but we do not consider them equivalent since \( F_3 \) can always diverge whereas \( F_2 \) can never diverge. In this paper we introduce a new data element '1' and use the euphemism 'F outputs 1' when we actually mean 'F diverges'. Then two algorithms are operationally equivalent if their input–output behaviour w.r.t. all data elements coincide. Thus the two algorithms defined by

\[ F_4^{\langle X \rangle} \leftarrow \text{if } X = 0 \text{ then } 1 \text{ else } F_4^{\langle X+1 \rangle} \lor F_4 \]

and

\[ F_5^{\langle X \rangle} \leftarrow \text{if } X = 0 \text{ then } 1 \text{ else } 1 \]

are not operationally equivalent since \( F_4 \) can diverge on any input other than 0, if it wants to.

This convention makes operational equivalence much more complicated for nondeterministic algorithms than for deterministic ones. This is however a true reflection of the complexity of nondeterminism as opposed to that of determinism.

1.2 From the above discussion we see that every nondeterministic algorithm determines a total mapping from the set of values \( V \) (data elements plus '1') to the set of non-empty subsets of values \( 2^V - \{\phi\} \). To define a mathematical semantics for such algorithms it seems sufficient to restrict our attention to such mappings. However, consider the following equation:

\[ F_6^{\langle X \rangle} \leftarrow \text{if } X = 0 \text{ then } 0 \text{ else } G^{\langle X-1 \rangle} \lor G \]
\[ G^{\langle X \rangle} \leftarrow \text{if } X = 0 \text{ then } 0 \text{ else if } X = 1 \text{ then } 1 \text{ else } 2. \]

Then \( F_6^{\langle 1 \rangle} \) evaluates to 0, 1 or 2. If a mapping \( g \) from \( V \to 2^V - \{\phi\} \) is given as a denotation of \( G \) then, since the meaning should be independent of the context, the same \( g \) will be used to define \( f_6 \), the denotation of \( F_6 \). Since
g:V \to 2^V - \{\phi\}, the only possible interpretation of G(0 or 1) is \{g(0), g(1)\}. So we would incorrectly get \(f_6(1) = \{0, 1\}\). A solution which was originally suggested in Egli (1975) is to interpret algorithms as mappings from \(2^V - \{\phi\}\) to \(2^V - \{\phi\}\) and this is the approach we take.

1.3 In Plotkin (1975) a typed \(\lambda\)-calculus is considered as a programming language and using an evaluation mechanism an operational semantics is defined. Roughly speaking, two programs \(P_1, P_2\) are considered to be operationally equivalent if \(P_1\) may be replaced by \(P_2\) in any program \(P_3\) (which uses \(P_1\) as a subprogram) without altering the input-output behaviour of \(P_3\).

Then using the models of L.C.F. developed by Milner (1973) a denotation is given to each program in such a way that two programs receive the same denotation if and only if they are operationally equivalent.

In section 2 this language is extended to allow nondeterministic programs by introducing the control structure 'or' and the evaluation mechanism is suitably extended. An operational semantics is then defined which takes into consideration not only the input-output behaviour of programs but also the possibility of diverging, as exemplified in the algorithm \(P_3\) in section 1.1. The problem then is to associate with each program of the extended language a denotation so that two programs are operationally equivalent in this new sense if and only if they receive the same denotation. In section 3 a model, derived from Egli (1975) is used to give each program a denotation and in section 4 the problem is partially resolved.

In general proofs are either sketched or omitted entirely. Many of the theorems are fairly simple extensions of results found in Plotkin (1975) and readers familiar with this work will have no difficulty in expanding on the sketched or nonexistent proofs. Apart from this, familiarity with Plotkin (1975) is not assumed and is not essential to the understanding of the paper. However, an acquaintance with various concepts and techniques prevalent in mathematical semantics is taken for granted. For an introduction to this area the authors recommend Scott (1970), Manna and Vuillemin (1975).

1.4 In his paper on the \(\text{P}\omega\) model of the untyped \(\lambda\)-calculus (Scott, 1975), D. Scott introduces 'multiple integers' and 'many-valued functions' and the nondeterministic control structure 'or' can presumably be accommodated within this
framework. However, since 'I' has as a denotation the empty set, terms such as $F_4$ and $F_5$ above would receive the same interpretation. So it seems that the $F_n$ model would only be suitable for modelling the input-output behaviour of terms. The same observation applies to both Nivat (1974) and Engelfriet and Schmidt (1975) who discuss the semantics of nondeterministic recursive schemas without tests. Nevertheless, the relationship between the programs of higher type in the present paper and the objects in the hierarchies in section 7 of the latter is worth investigating.

2. THE LANGUAGE $L$

We first define a system of types.

**Definition:** i) $i$ and 0 are types, called the ground types.

  ii) if $\alpha, \sigma$ are types, so is $(\alpha \rightarrow \sigma)$.

A type of the form $\sigma_1 \rightarrow (\sigma_2 \rightarrow (\ldots \rightarrow \sigma_n)\ldots)$ will be represented as $(\sigma_1, \ldots, \sigma_n)$. The integers will be of type $i$ whereas 'true' and 'false' will be of type 0.

To define the terms of $L$ we need

i) an infinite set of constants $k_0, k_1, \ldots$ of type $i$.

ii) $tt, ff$, constants of type 0.

iii) $[+1], [-1]$, constants of type $i \rightarrow i$.

iv) $z$, a constant of type $i \rightarrow 0$.

v) for every type $\sigma$, a constant $IF_\sigma$ of type $(0, \sigma, \sigma, \sigma)$

and a constant $Y_\sigma$ of type $((\sigma \rightarrow \sigma) \rightarrow \sigma)$.

vi) for every type $\sigma$, an infinite set of variables $\{x_1^\sigma\}$.

**Definition:**

a) a variable or constant of type $\sigma$ is a term of type $\sigma$.

b) If $t$ is a term of type $\tau$, $\lambda x^\sigma.t$ is a term of type $(\sigma \rightarrow \tau)$.

c) If $t$ is a term of type $(\sigma \rightarrow \tau)$, $s$ a term of type $\sigma$, then $(ts)$ is a term of type $\tau$.

d) If $t, s$ are terms of type $\sigma$, so is $(t \sigma s)$.

The definition of free and bound variable is as usual. A term is closed if it has no free variables. For the sake of
clarity indication of types will be omitted whenever possible. Non-essential brackets will also be suppressed. Terms will usually be denoted by letters such as $t$, $s$, etc. possibly subscripted.

A context $C[ , , , ... ]$ is a term with 'holes', which may be replaced by appropriate terms, to obtain $C[t_1, ..., t_n]$.

$[t|X]s$ is the result of substituting $t$ for all free occurrences of $X$, making appropriate changes in the bound variables of $s$ so that no free variables of $t$ become bound. This definition can be generalized to $[t_1|X_1, ..., t_n|X_n]s$ in the obvious way. We omit the formal definitions.

The recursive equations considered in the introduction can all be represented as closed terms of type $(\alpha, \beta, \gamma, ...)$, for ground types $\alpha, \beta, \gamma, ...$, which we call simple programs. For example, $F_4$ can be represented by

$Y \lambda F. (IF, (ZX), 1, (F(([+1]X) \mathbin{or} ([-1]X))))$.

In this example, as in other examples in the paper, we write 0 instead of $k_0$, 1 instead of $k_1$ etc. Higher level programming concepts can also be represented within the language.

2.1 The Evaluation Mechanism

The evaluation mechanism is defined as a relation $\rightsquigarrow$ between terms.

**Definition 2.1.1:**

1. $[+1]k_n \rightsquigarrow k_{n+1}$
2. $[-1]k_{n+1} \rightsquigarrow k_n$
3. $Z k_0 \rightsquigarrow \text{tt}$
4. $Z k_{n+1} \rightsquigarrow \text{ff}$
5. $IF_{tt} t s \rightsquigarrow t$
6. $IF_{ff} t s \rightsquigarrow s$
7. $(\lambda X.t)s \rightsquigarrow [s|X]t$
8. $Y t \rightsquigarrow t(Yt)$
v) \[ t \triangleright s \overset{0}{\longrightarrow} t \]
\[ t \triangleright s \overset{0}{\longrightarrow} s \]

We say \( \alpha \overset{1}{\longrightarrow} \beta \) if \( \beta \) is obtained from \( \alpha \) by applying \( \overset{0}{\longrightarrow} \) to some subterm of \( \alpha \).

Let \( \overset{0}{\longrightarrow} \) be the reflexive transitive closure of \( \overset{1}{\longrightarrow} \). Thus the evaluation mechanism is just that of the typed \( \lambda \)-calculus discussed in Plotkin (1975) with the addition of a nondeterministic choice operator. Note that the parameter passing mechanism is the body replacement rule of Algol 60.

2.2 Operational Semantics

Given a closed term \( t \) of ground type, written \( \text{cgt} \), we say \( k_n \in \text{EVAL}(t) \) iff \( t \overset{0}{\longrightarrow} k_n \). As was stated in the introduction it is not sufficient to say that \( t_1 \) and \( t_2 \), both \( \text{cgt}s \), are operationally equivalent iff \( \text{EVAL}(t_1) = \text{EVAL}(t_2) \).

Example: \( t_1 \equiv 1 \)
\[ t_2 \equiv 1 \triangleright (Y \lambda G.\lambda X.(X \triangleright (GX)))1 \]
\( \text{EVAL}(t_1) = \text{EVAL}(t_2) \) but we would not call them equivalent.

We need to introduce an extra syntactic entity '★' and say \( ★ \in \text{EVAL}(t) \) whenever \( t \) can diverge. Unfortunately, 'divergence' is not the same as 'infinite computation'.

Example 1: If \( t \equiv [-1]0 \), then \( t \) cannot produce any constant and must be considered a divergent term. So we say a \( \text{cgt} \) \( t \) is blocked if \( t \) is not a constant and for no \( s \) does \( t \overset{1}{\longrightarrow} s \). Then if \( t \overset{0}{\longrightarrow} b \), where \( b \) is a blocked term, we say \( t \) can diverge. This type of divergence will be denoted by \( t \overset{0}{\longrightarrow} \Omega \). Note that for our language the only blocked \( \text{cgt} \) is \([]-1]0\).

Example 2: \( t \equiv Ys \equiv Y \lambda G.\lambda X.(IF(ZX), 0,(G[-1]X)) \). Then
\( t0 \equiv (Ys)0 \overset{0}{\longrightarrow} s(Ys)0 \overset{0}{\longrightarrow} \ldots \overset{0}{\longrightarrow} s^n(Ys)0 \overset{0}{\longrightarrow} \ldots \). So \( t0 \) has an infinite computation. But obviously we do not want to consider \( t0 \) as diverging.

For a proper definition of divergence we need the idea of a standard derivation.
Using the well-known technique, as in Duong (1974), every well-formed term can be represented as a tree. Thus

\[
\lambda x.t \quad \xi \quad t_1 \otimes t_2
\]

is represented as

\[
\begin{array}{c}
\gamma \\
 t \\
\lambda \\
 x \\
 t \\
\xi \\
 t_1 \\
 t_2 \\
\end{array}
\]

etc.

Every reduction sequence can then be considered as a sequence of transformations on trees, each transformation being applied to some subtree. Since each subtree has a unique root we can say that a transformation applies to a node of the tree.

Then a reduction sequence is **standard** if each transformation is either

i) applied to an 'or' node;

ii) applied to the topmost leftmost node of the current tree capable of being transformed.

Using the linear representation of terms a reduction sequence is standard if the reductions occur from left to right, except we allow choices to be made anywhere within the term.

**Proposition 2.2.1**

If \( k \) is a constant of ground type and \( t \xrightarrow{\cdot} k \), then there is a standard derivation of \( k \) from \( t \), written \( t \xrightarrow{S} k \).

**Proof** An adaptation of the proof in Duong (1974).

**Definition:** For any ground term \( t \), \( t \) can diverge, written \( \ast \in \text{EVAL}(t) \) iff \( t \xrightarrow{\cdot} \Omega \) or there exists an infinite standard derivation from \( t \), written \( t^\uparrow \).

An alternate and possibly more illuminating definition of divergence can be given as follows: A cgt \( t \) is unsolvable if for no constant \( k \) of ground type, does \( t \xrightarrow{\cdot} k \). So unsolvable terms are computationally useless. Then it is easy to see that
t can diverge iff $t \leadsto u$, u unsolvable or there exists a standard reduction sequence from t in which the 'or' reduction is used infinitely often.

Example: a) $(\lambda x.\lambda y.((\text{if } x, 1, (G[-1]x \lor G[+1]x)))1$ can diverge because an infinite number of choices can be made.

b) $(\lambda x.\lambda y.((\lambda f.\lambda w.\lambda z.((\text{if } x, (\lambda f.\lambda w.\lambda z.((f[+1]w)z), G[-1]f)))0)$ can diverge because it reduces to the unsolvable term $(\lambda f.\lambda w.\lambda z.((f[+1]w))0$.

Using EVAL as defined above we can compare the behaviour of any two terms $t_1, t_2$ of the same type. We say $t_1 \leq t_2$ iff, whenever $G[t_1]$ and $G[t_2]$ are cgt's, $\text{EVAL}(G[t_1]) \leq \text{EVAL}(G[t_2])$ (set theoretically). We say $t_1$ and $t_2$ are operationally equivalent, written $t_1 \equiv t_2$, iff $t_1 \leq t_2$ and $t_2 \leq t_1$. Obviously, $\equiv$ is an equivalence relation.

3. THE MATHEMATICAL MODEL

3.1 Introduction

We briefly recall some definitions. Details can be found in Milner (1973).

Let $<D,E>$ be a partial order. A subset $E$ of $D$ is directed iff it is non-empty and every pair of elements of $E$ has an upper bound in $E$. The partial order $<D,E>$ is a cpo iff every directed subset $X$ of $D$ has a least upper bound, $\cup X$, in $D$ and if $D$ contains a minimal element. A function $f:D \rightarrow E$, from one cpo $D$, to another cpo $E$, is continuous iff for every directed subset $X$ of $D$, $f(\cup X) = \cup \{f(x) : x \in D \}$. It is well known that the set of continuous functions from $D$ to $E$, $[D \rightarrow E]$, is a cpo under the induced pointwise ordering.

In the area of mathematical semantics, it has now become common to interpret a datatype as some cpo and in particular to interpret the integers as the 'flat' cpo $D_0$:

$$0 \quad 1 \quad 2 \quad 3 \ldots$$

$$\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ Quad
order on this set.

Let $X$ be the set of possible partial results obtained by a nondeterministic process after operating for a certain period of time. If the process is allowed to continue for some further period of time and $Y$ is the set of new partial results then it is natural to expect that $Y$ consists of a set of 'improvements' on the elements of $X$. This well-known concept of $X$ approximating $Y$ leads to the following definition.

**Definition:** For $X, Y \in \{2^D_0 - \phi\}$ let $X \subseteq Y$ iff

1) $\forall x \in X, \exists y \in Y \quad x \subseteq y$

2) $\forall y \in Y, \exists x \in X \quad x \subseteq y$

Then $\langle 2^D_0 - \phi, \subseteq \rangle$ is a complete partial order. Note however that this is true only because $D_0$ is a flat cpo. If we wish to consider more general datatypes, i.e. datatypes that cannot be considered as flat cpo's, the above definition must be modified. The most likely possibility is to restrict in some way the allowable subsets.

In our case we also wish to restrict the formation of sub-sets. If a nondeterministic algorithm is capable of outputting an infinite number of integers (on a single input), then it must also be capable of diverging. This is because in producing an infinite number of integers an infinite number of choices can be made and so the algorithm can also spend all of its time just choosing. This leads to the following definition.

Let $D_1 = \{X | X \in 2^{D_0 - \phi}, |X| = \infty \Rightarrow \bot \in X\}$. Then $\langle D_1, \subseteq \rangle$ is a cpo and is in fact algebraic (for definition see Plotkin (1975)). We also let $\langle D_0, \subseteq \rangle$ denote the subset cpo constructed, in a similar way, from the cpo:

```
  t
 /\f
\bot
```

Perhaps surprisingly the same process of taking some subset cpo is not necessary at higher levels. This was first suggested in Egli (1975). Cpo's do not however have sufficient structure for our purposes. Nondeterministic processes in general output sets of results and the partial order of inclusion on these sets generates a partial order on the process themselves. It seems necessary to consider this
partial order if we wish to have a semantic counterpart to the syntactic 'or' construct. Accordingly, we will have two partial orders, one computational, the other set theoretic.

3.2 The Model

The required setting for the interpretation of terms will be called a structure.

Definition:

\[ \langle \mathfrak{D}, \sqsubseteq \rangle \text{ is a structure if} \]

- a) \( \langle \mathfrak{D}, \sqsubseteq \rangle \) is a cpo
- b) \( \langle \mathfrak{D}, \sqsubseteq \rangle \) is a po such that any \( d_1, d_2 \in \mathfrak{D} \) has an upper bound \( d_1 \sqcup d_2 \) in \( \mathfrak{D} \)
- c) \( \sqsubseteq \) is \( \sqcup \)-complete
  - i.e. if \( x = \sqcup \{ x_n : n \in I \} \), \( y = \sqcup \{ y_n : n \in I \} \) both directed sets and \( x_n \sqsubseteq y_n \), for all \( n \in I \),
    then \( x \sqsubseteq y \).
- d) \( \sqcup \) is \( \sqsubseteq \)-continuous.

Theorem 3.2.1

- a) \( \langle \mathfrak{D}_0, \sqsubseteq \rangle \) and \( \langle \mathfrak{D}_1, \sqsubseteq \rangle \), where \( \sqsubseteq \) is set inclusion, are structures.
- b) if \( \langle \mathfrak{D}_1, \sqsubseteq_1 \rangle \), \( \langle \mathfrak{D}_2, \sqsubseteq_2 \rangle \) are structures then the set of \( \sqsubseteq \)-continuous, \( \sqsubseteq \)-monotonic functions from \( \mathfrak{D}_1 \) to \( \mathfrak{D}_2 \), written \( \left[ \mathfrak{D}_1, \mathfrak{D}_2 \right] \) is also a structure under the induced partial orderings.

Proof. Left to the reader.  

We are now ready to define the domains in which the language \( L \) of section 2.1 can be interpreted.

1) For type 0 let \( \mathbb{D}_0 \) be as defined above
2) For type \( i \) let \( \mathbb{D}_i \) be as defined above
3) For type \( (\alpha \rightarrow \tau) \) let \( \mathbb{D}_{\alpha \rightarrow \tau} \) be \( \left[ \mathbb{D}_\alpha, \mathbb{D}_\tau \right] \).

Define i) pone: \( \mathbb{D}_0 \to \mathbb{D}_0 \)
- by \( \text{pone}(X) = \cup \{ x+1 : x \in X \} \), where \( 1+1 = 1 \).
ii) mone: \( D_0 \rightarrow D_0 \)
by \( \text{mone}(X) = \bigcup \{ x - 1 \mid x \in X \} \), where \( 0 - 1 = 1 - 1 = 1 \)

iii) \( z: D_0 \rightarrow D_0 \)
by \( z(X) = \bigcup \{ p(x) \mid x \in X \} \), where \( p(0) = 1 \), \( p(n+1) = 0 \), \( p(1) = 1 \)

iv) \( \text{if}: D_0 \rightarrow (D_\tau \rightarrow (D_\tau \rightarrow D_\tau)) \)
by \( \text{if}_\sigma(x, \alpha, \beta) = \bigcup \{ q(x, \alpha, \beta) \mid x \in X \} \), where
\( q(t, \alpha, \beta) = \alpha, q(f, \alpha, \beta) = \beta, q(1, \alpha, \beta) = 1 \).

Proposition 3.2.2

a) \( \text{pone}, \text{mone}, z \in D_{0+0} \)
b) for every \( \sigma \), \( \text{if}_\sigma \in D_{0+(\sigma \rightarrow (\sigma \rightarrow \sigma))} \)
c) for every \( \sigma \), if \( y_\sigma \) is the fixpoint operator on the
\( \text{cpo} \langle D_\sigma, \preceq \rangle \), \( y_\sigma \in D_{(\sigma \rightarrow \sigma) \rightarrow \sigma} \).

Proof Omitted. \( \Box \)

We can now give an interpretation to all the well defined terms in \( L \). As usual an interpretation must be given w.r.t. an environment. Let \( \text{VAR} \) be the set of variables. Then \( \rho \) is an environment, written \( \rho \in \text{ENV} \), iff \( \rho \) is a type-respecting function from \( \text{VAR} \) to \( \bigcup \tau D_\tau \). For \( \alpha \) a variable and \( x \) an element of \( \bigcup \tau D_\tau \), \( \rho(x/\alpha) \) is the environment identical to \( \rho \) except that \( \alpha \) is mapped onto \( x \).

Then define \( V: L \rightarrow (\text{ENV} \rightarrow \bigcup \tau D_\tau) \) by
\( V[\alpha_\sigma](\rho) = \rho(\alpha_\sigma) \)
\( V[K_n](\rho) = \{n\} \)
\( V[+1](\rho) = \text{pone} \)
\( V[-1](\rho) = \text{mone} \)
\( V[Z](\rho) = z \)
\( V[IF_\sigma](\rho) = \text{if}_\sigma \)
\( V[M \NN](\rho) = V[M](\rho) \cup V[N](\rho) \)
\( V[\lambda x.M](\rho)(x) = V[M](\rho)(x/\alpha) \)
\( V[y_\sigma](\rho) = y_\sigma \)
\( V[M \geq N](\rho) = V[M](\rho) \cup V[N](\rho) \)

Note that for every term \( t_\sigma \) and \( \rho \in \text{ENV} \), \( V[t_\sigma](\rho) \in D_\sigma \).
Theorem 3.2.3

If $t \xrightarrow{\omega} s$ then $V[t](\rho) \geq V[s](\rho) \ \forall \rho \in \text{ENV}$.

Proof. If $\alpha \xrightarrow{\omega} \beta$ then it is easy to prove that $V[\alpha](\rho) \geq V[\beta](\rho)$. Because every well-formed term is interpreted as a $\leq$-monotonic function, we can apply the above remark to show $t \xrightarrow{1} s \Rightarrow V[t](\rho) \geq V[s](\rho)$. The result then follows by induction on the length of the reduction. □

So our interpretation is indeed a model of the evaluation mechanism.

4.1 Mathematical vs Operational Semantics of Closed Ground Terms

This section is devoted to showing that for ground terms the operational and denotational semantics coincide. In other words, given cghts $t_1$ and $t_2$, we prove that $\text{EVAL}(t_1) \subseteq \text{EVAL}(t_2)$ iff $V[t_1](\rho) \leq V[t_2](\rho), \forall \rho \in \text{ENV}$. Notice that since $t_1$ and $t_2$ are closed it suffices to consider only the trivial environment which maps all variables into 1. We also denote this environment by 1. It is also convenient to define $V[*](\rho) = 1$, $\forall \rho \in \text{ENV}$.

Lemma 4.1.1 For any cght $t$ and constant $k$

$$k \in \text{EVAL}(t) \text{ iff } V[k](1) \leq V[t](1).$$

Proof. If $k \in \text{EVAL}(t)$ it follows from Theorem 2.3.2 that $V(t)(1) \geq V[k](1)$. The converse is similar to Theorem 3.1 of Plotkin (1975) and is omitted. □

Lemma 4.1.2 For any cght $t$ if $* \in \text{EVAL}(t)$ then $V[t](1) \geq \{1\}$.

Outline of proof: We consider only the case of $t^+$.

For every $\sigma$, let $\Omega_\sigma = (Y_\sigma \lambda \alpha. \alpha)$.

Let $Y^0_\sigma = \Omega_\sigma$

and $Y^{n+1}_\sigma = \lambda \alpha. \alpha(Y^n_\sigma \alpha)$.

Then it can be proven that

$$V[Y_\sigma](\rho) = \cup_n V[Y^n_\sigma](\rho), \ \forall \rho \in \text{ENV}.$$
Furthermore, let $Y_{\sigma_1}, \ldots, Y_{\sigma_n}$ be all the $Y$ terms occurring in $t$. Then

$$V[t](\rho) = \cup_k V[[Y_{\sigma_1}^{k} | Y_{\sigma_1}^{k}, \ldots, Y_{\sigma_n}^{k} | Y_{\sigma_n}^{k}]t](\rho).$$

Now if $t$ has an infinite standard derivation, an instance of the reduction $(Ys) \rightarrow_s s(Ys)$ must be used an infinite number of times. Consequently, it can be proven that

$$[Y_{\sigma_1}^{k} | Y_{\sigma_1}^{k}, \ldots, Y_{\sigma_n}^{k} | Y_{\sigma_n}^{k}]t \rightarrow u,$n

for some unsolvable $u$.

So $V[[Y_{\sigma_1}^{k} | Y_{\sigma_1}^{k}, \ldots, Y_{\sigma_n}^{k} | Y_{\sigma_n}^{k}]t](\iota) \supseteq \{\iota\}$. Using the definition of $E$ on $D_0$ we can then show that $V[t](\iota) \supseteq \{\iota\}$. \hfill \Box

**Lemma 4.1.3** If $\iota \in V[t](\iota)$ then $* \in \text{EVAL}(t)$, for any cgt $t$.

Outline of proof: The method is similar to that of Theorem 3.1 of Plotkin (1975). A predicate $P_{\sigma}$ is defined by induction on types and then it is shown by structural induction on the terms that every term $t_{\sigma}$ has property $P_{\sigma}$.

Definition of $P_{\sigma}$:

a) If $t_{\sigma}$ is a cgt then $t_{\sigma}$ has $P_{\sigma}$ if $\iota \in V[t](\iota) \Rightarrow * \in \text{EVAL}(t)$. (Statement of Lemma).

b) If $t_{(\sigma \rightarrow_t)}$ is closed then it has $P_{(\sigma \rightarrow_t)}$ if whenever the closed term $s_{\sigma}$ has $P_{\sigma}$, $(ts)$ has $P_{\sigma}$.

c) If $t_{\sigma}$ is open with free variables $x_{\sigma_1}^{1}, \ldots, x_{\sigma_n}^{n}$, then it has property $P_{\sigma}$ if $[s_{\sigma_1}^{1} | x_{\sigma_1}^{1}, \ldots, s_{\sigma_n}^{n} | x_{\sigma_n}^{n}]t$ has property $P_{\sigma}$ whenever $s_{\iota}^{1}$ has property $P_{\sigma_{\iota}}$.

Let $t$ be okay if it has the property $P_{\sigma}$. As an example of the use of structural induction to prove that every term is okay, we outline a method to show that $Y_{\sigma}$ is okay. It is sufficient to show that $Y_{\sigma} t_{\iota}^{1} \ldots t_{\iota}^{k}$ is okay, where $t_{\iota}$ are okay and $Y_{\sigma} t_{\iota}^{1} \ldots t_{\iota}^{k}$ is of ground type. Let $\iota \in V[Y_{\sigma} t_{\iota}^{1} \ldots t_{\iota}^{k}](\iota)$. Then

$$\iota \in \bigcup_n V[[Y_{\sigma} t_{\iota}^{1} \ldots t_{\iota}^{k}](\iota) \Rightarrow \iota \in V[Y_{\sigma} t_{\iota}^{1} \ldots t_{\iota}^{k}](\iota), \forall n.$$
Using the induction hypothesis we can show that \( Y^\eta_0 \) is okay for every \( n \). We have three cases:

i) \[
Y^\eta_0 t_1 \ldots t_k \leadsto \Omega \text{ for some } n.
\]
Then \( Y^\eta_0 t_1 \ldots t_k \leadsto \Omega \) i.e. \( Y^\eta_0 \) is okay.

ii) For some \( n, Y^\eta_0 t_1 \ldots t_k \uparrow \) and in this sequence the reduction \( Y^k_0 t_1 \leadsto t_1(Y^{k-1}_0 t_1) \) is used a finite number of times. In this case \( Y^\eta_0 t_1 \leadsto Y^\eta_0 t_1 \) is never used and so exactly the same sequence of reductions can be applied to \( Y^\eta_0 t_1 \ldots t_k \).

iii) For every \( n, Y^\eta_0 t_1 \leadsto Y^\eta_0 t_1 \) is used (and therefore used an finite number of times). Then consider the tree of standard reductions from \( Y^\eta_0 t_1 \ldots t_k \). It is infinite and every node has a finite outdegree. By Köngs lemma there is an infinite path which represents an infinite standard derivation from \( Y^\eta_0 t_1 \ldots t_k \). □

**Theorem 4.1.4** For cgt \( t \), \( V[t](\uparrow) = \cup\{V[\alpha](\uparrow) | \alpha \in EVAL(t)\} \).

**Proof** Immediate from the previous lemmas. □

**Corollary 4.1.5** For cgts \( t_1 \) and \( t_2 \),

\[
EVAL(t_1) \subseteq EVAL(t_2) \text{ iff } V[t_1]\downarrow \subseteq V[t_2]\downarrow.
\]

**4.2 Mathematical vs Operational Semantics for Arbitrary Terms**

In this section we discuss the relationship between the behaviour of terms, i.e. their operational semantics and their interpretation in the model, i.e. their mathematical semantics.

**Proposition 4.2.1** If \( V[t_1](\rho) \subseteq V[t_2](\rho) \) then \( t_1 \preceq_\rho t_2 \).

**Proof** A simple deduction from Corollary 4.1.5 □

It follows from this proposition that if two terms have the same denotation they have the same behaviour. Unfortunately, we can only prove a partial converse.

**Proposition 4.2.2** If \( t_1, t_2 \) are simple programs (i.e. terms of type \( (\alpha, \beta, \gamma, \ldots) \), for ground types \( \alpha, \beta, \gamma, \ldots \)) then
\( t_1 \preceq_{\text{op}} t_2 \Rightarrow V[t_1](\rho) \preceq_{\text{op}} V[t_2](\rho), \forall \rho \in \text{ENV}. \)

**Proof** For simplicity assume \( t_1 \) and \( t_2 \) are closed and of type \((0 \to 0)\). The generalisation to arbitrary programs is not difficult.

Suppose \( V[t_1](\rho) \not\preceq V[t_2](\rho) \). Let \( V[t_1](\rho) \) define \( f \) and \( V[t_2](\rho) \) define \( g \). Then \( f, g \in \mathcal{D}(0 \to 0) \). Using the definition of \( \preceq \) on \( \mathcal{D}(0 \to 0) \) we can show that \( f \not\preceq g \) iff \( f(s) \not\preceq g(s) \) for some finite set \( s \). Since \( s \) is finite, it can be defined by \( S \) say. Therefore, \( \text{EVAL}(t_1 S) \not\preceq \text{EVAL}(t_2 S) \). So \( t_1 \not\preceq_{\text{op}} t_2 \). \( \square \)

In attempting to generalise Proposition 4.2.2 to arbitrary terms a possible weakness of our model crops up. The subsets of definable functions at any level seems to be a very small subset of the functions in the model. This is because the model is constructed using functions from subsets and as a language for such functions \( L \) is woefully inadequate. For example, the function \( f: \mathcal{D}_0 \to \mathcal{D}_0 \) defined by

\[
\begin{align*}
0 &\in f(X) \text{ if } 0 \in X \\
1 &\in f(X) \text{ if } 1 \in X \\
2 &\in f(X) \text{ if } \{0,1\} \subseteq X \\
\bot &\in f(X) \text{ if } \alpha \in X, \alpha \neq 0,1,\bot
\end{align*}
\]

is not definable.

One reason why the set of definable functions is so restricted is that the parameter passing mechanism is call-by-name. Informally speaking, this means that when operating on a set of values we cannot force two consecutive primitive operations to be applied to any one element of the set. To remedy this a call-by-value mechanism could be instituted.

To conclude, we point out that using the techniques of this paper a model for a nondeterministic typeless \( \lambda \)-calculus can be constructed but the properties of this model have yet to be investigated.

**REFERENCES**


