THE SEMANTICS OF NONDETERMINISM

M. Hennessy
E.A. Ashcroft

Department of Computer Science
University of Waterloo
Waterloo, Ontario

CS-76-17
April 1976

M. Hennessy
E.A. Ashcroft

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 λ-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 '$\underset{\sim}{\text{or}}$'. Thus using the notation of elementary number theory we can form equations such as

$$F_1\langle N,Y\rangle \Leftarrow \text{if } N \mid Y \text{ then } F\langle 2,Y+1\rangle \text{ else if } N > \lfloor Y/2 \rfloor$$
$$\text{then } (Y \underset{\sim}{\text{or}} F\langle 2,Y+1\rangle) \text{ else } F\langle N+1,Y\rangle.$$

The usual evaluation mechanism can be extended to terms involving '$\underset{\sim}{\text{or}}$' by allowing the term $(T_1 \underset{\sim}{\text{or}} T_2)$ 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

$$F_2\langle X\rangle \Leftarrow \text{if } X = 0 \text{ then } 0 \text{ else } (X \underset{\sim}{\text{or}} F_2\langle X-1\rangle)$$

$$F_3\langle X\rangle \Leftarrow \text{if } X = 0 \text{ then } 0 \text{ else } (X \underset{\sim}{\text{or}} F_3\langle X-1\rangle) \underset{\sim}{\text{or}} G\langle X\rangle$$

$$G\langle X\rangle \Leftarrow G\langle X+1\rangle$$

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 '$\perp$' and use the euphemism 'F outputs $\perp$' 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

$$F_4\langle X\rangle \Leftarrow \text{if } X = 0 \text{ then } 1 \text{ else } F_4\langle X+1 \underset{\sim}{\text{or}} X-1\rangle$$

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 '$\perp$') 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:

$$F_6\langle X\rangle \Leftarrow \text{if } X = 0 \text{ then } 0 \text{ else } G\langle X-1 \underset{\sim}{\text{or}} X\rangle$$

$$G\langle X\rangle \Leftarrow \text{if } X = 0 \text{ then } 0 \text{ else if } X = 1 \text{ then } 1 \text{ else } 2.$$

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 $G(0 \underset{\sim}{or} 1)$ 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 λ-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 '$\underset{\sim}{or}$' 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 F_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 non-existent 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 Pω model of the untyped λ-calculus (Scott, 1975), D. Scott introduces 'multiple integers' and 'many-valued functions' and the nondeterministic control structure '$\underset{\sim}{or}$' can presumably be accommodated within this

framework. However, since '$\perp$' 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 $P\omega$ 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 α,σ are types, so is $(\alpha \to \sigma)$.

A type of the form $\sigma_1 \to (\sigma_2 \to (\ldots \to \sigma_n)\ldots)$ 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 $i \to i$.

iv) Z, a constant of type $i \to 0$.

v) for every type σ, a constant IF_σ of type $(0,\sigma,\sigma,\sigma)$ and a constant Y_σ of type $((\sigma \to \sigma) \to \sigma)$.

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

Definition:

a) a variable or constant of type σ is a term of type σ.

b) If t is a term of type τ, $\lambda X^\sigma.t$ is a term of type $(\sigma \to \tau)$.

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

d) If t,s are terms of type σ, so is (t $\underset{\sim}{\text{or}}$ 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

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

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 $\overset{o}{\rightsquigarrow}$ between terms.

Definition 2.1.1:

i) $[+1]k_n \overset{o}{\rightsquigarrow} k_{n+1}$

$[-1]k_{n+1} \overset{o}{\rightsquigarrow} k_n$

$Z\ k_0 \overset{o}{\rightsquigarrow} tt$

$Z\ k_{n+1} \overset{o}{\rightsquigarrow} ff$

ii) $IF_\sigma tt\ t\ s \overset{o}{\rightsquigarrow} t$

$IF_\sigma ff\ t\ s \overset{o}{\rightsquigarrow} s$

iii) $(\lambda X.t)s \overset{o}{\rightsquigarrow} [s|X]t$

iv) $Yt \overset{o}{\rightsquigarrow} t(Yt)$

v) $t \underset{\sim}{or}\ s \overset{o}{\sim\!\!\rightarrow} t$

$t \underset{\sim}{or}\ s \overset{o}{\sim\!\!\rightarrow} s$

We say $\alpha \overset{1}{\sim\!\!\rightarrow} \beta$ if β is obtained from α by applying $\overset{o}{\sim\!\!\rightarrow}$ to some subterm of α.

Let $\sim\!\!\rightarrow$ be the reflexive transitive closure of $\overset{1}{\sim\!\!\rightarrow}$. Thus the evaluation mechanism is just that of the typed λ-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 cgt, we say $k_n \in EVAL(t)$ iff $t \sim\!\!\rightarrow k_n$. As was stated in the introduction it is not sufficient to say that t_1 and t_2, both cgts, are operationally equivalent iff $EVAL(t_1) = EVAL(t_2)$.

Example: $t_1 \equiv 1$

$t_2 \equiv 1 \underset{\sim}{or} (Y\ \lambda G.\lambda X.(X \underset{\sim}{or} (GX)))1$

$EVAL(t_1) = EVAL(t_2)$ but we would not call them equivalent.

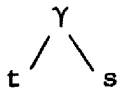
We need to introduce an extra syntactic entity '*' and say $* \in EVAL(t)$ whenever t can diverge. Unfortunately, 'divergence' is not the same as 'infinite computation'.

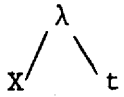
Example 1: If $t \equiv [-1]0$, then t cannot produce any constant and must be considered a divergent term. So we say a cgt t is blocked if t is not a constant and for no s does $t \overset{1}{\sim\!\!\rightarrow} s$. Then if $t \sim\!\!\rightarrow b$, where b is a blocked term, we say t can diverge.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

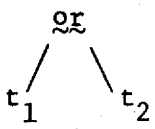
Example 2: $t \equiv Ys \equiv Y\ \lambda G.\lambda X.(IF(ZX), 0, (G[-1]X))$. Then $t0 \equiv (Ys)0 \sim\!\!\rightarrow s(Ys)0 \sim\!\!\rightarrow \ldots \sim\!\!\rightarrow s^n(Ys)0 \sim\!\!\rightarrow \ldots$. So t0 has an infinite computation. But obviously we do not want to consider t0 as diverging.

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

(ts) is represented as 

λX.t " " " 

t_1 or t_2 " " 

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 $t \rightsquigarrow k$, then there is a standard derivation of k from t, written $t \overset{s}{\rightsquigarrow} k$.

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

Definition: For any ground term t, t can diverge, written $* \in EVAL(t)$ iff $t \rightsquigarrow \Omega$ 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 $t \rightsquigarrow k$. So unsolvable terms are computationally useless. Then it is easy to see that

t can diverge iff t $\rightsquigarrow$ u, u unsolvable or there exists a standard reduction sequence from t in which the 'or' reduction is used infinitely often.

Example: a) (Y λG.λX(IF ZX,1,(G[-1]X or G[+1]X)))1 can diverge because an infinite number of choices can be made.

b) (Y λG.λX(IF ZX,(Y λF.λW.F[+1]W)X, G[-1]X))0 can diverge because it reduces to the unsolvable term (Y λF.λW.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 $t_1 \underset{op}{\subseteq} t_2$ iff, whenever $C[t_1]$ and $C[t_2]$ are cgts, $EVAL(C[t_1]) \subseteq EVAL(C[t_2])$ (set theoretically). We say t_1 and t_2 are operationally equivalent, written $t_1 \underset{op}{=} t_2$, iff $t_1 \underset{op}{\subseteq} t_2$ and $t_2 \underset{op}{\subseteq} t_1$. Obviously, $\underset{op}{=}$ is an equivalence relation.

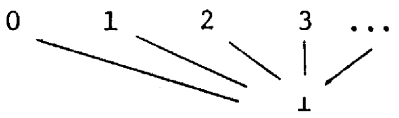
3. THE MATHEMATICAL MODEL

3.1 Introduction

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

Let $\langle D,E\rangle$ 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 $\langle D,E\rangle$ is a cpo iff every directed subset X of D has a least upper bound, $\sqcup X$, in D and if D contains a minimal element. A function $f:D \to E$, from one cpo D, to another cpo E, is continuous iff for every directed subset X of D, $f(\sqcup X) = \sqcup\{f(x) : x \in D\}$. It is well known that the set of continuous functions from D to E, $[D \to E]$, 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:



From the discussion in section 1, we see that programs must be interpreted as functions from $\{2^{D0}-\phi\}$ to $\{2^{D0}-\phi\}$. But it is necessary to define a computationally significant partial

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 $X \sqsubseteq Y$ iff

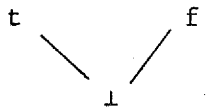
i) $\forall x \in X, \exists y \in Y \quad x \sqsubseteq y$

ii) $\forall y \in Y, \exists x \in X \quad x \sqsubseteq y$

Then $\langle 2^{D_0}-\phi, \sqsubseteq \rangle$ 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 subsets. 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 $\mathbb{D}_i = \{X \mid X \in 2^{D_0}-\phi, \ |X| = \infty \Rightarrow \perp \in X\}$. Then $\langle \mathbb{D}_i, \sqsubseteq \rangle$ is a cpo and is in fact algebraic (for definition see Plotkin (1975)). We also let $\langle \mathbb{D}_0, \sqsubseteq \rangle$ denote the subset cpo constructed, in a similar way, from the cpo:



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 '$\underset{\sim}{\text{or}}$' 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:

$\langle \mathbb{D}, \sqsubseteq, \subseteq \rangle$ is a structure if

a) $\langle \mathbb{D}, \sqsubseteq \rangle$ is a cpo

b) $\langle \mathbb{D}, \subseteq \rangle$ is a po such that any $d_1, d_2 \in \mathbb{D}$ has an upper bound $d_1 \cup d_2$ in $\mathbb{D}$

c) $\subseteq$ is $\bigsqcup$-complete
i.e. if $x = \bigsqcup\{x_n : n \in I\}$, $y = \bigsqcup\{y_n : n \in I\}$ both directed sets and $x_n \subseteq y_n$, for all $n \in I$,
then $x \subseteq y$.

d) $\cup$ is $\sqsubseteq$-continuous.

Theorem 3.2.1

a) $\langle \mathbb{D}_0, \sqsubseteq, \subseteq \rangle$ and $\langle \mathbb{D}_i, \sqsubseteq, \subseteq \rangle$, where $\subseteq$ is set inclusion, are structures.

b) if $\langle \mathbb{D}_1, \sqsubseteq_1, \subseteq_1 \rangle$, $\langle \mathbb{D}_2, \sqsubseteq_2, \subseteq_2 \rangle$ are structures then the set of $\sqsubseteq$-continuous, $\subseteq$-monotonic functions from $\mathbb{D}_1$ to $\mathbb{D}_2$, written $[\mathbb{D}_1, \mathbb{D}_2]$ 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.

i) For type 0 let $\mathbb{D}_0$ be as defined above

ii) For type i let $\mathbb{D}_i$ be as defined above

iii) For type $(\alpha \to \tau)$ let $\mathbb{D}_{\alpha\to\tau}$ be $[\mathbb{D}_\alpha, \mathbb{D}_\tau]$.

Define i) pone: $\mathbb{D}_0 \to \mathbb{D}_0$
by pone(X) = $\cup\{x+1 \mid x \in X\}$, where $\perp+1 = \perp$

ii) mone: $\mathbb{D}_0 \to \mathbb{D}_0$
by mone(X) = $\cup\{x-1 \mid x \in X\}$, where $0-1 = \perp-1 = \perp$

iii) z: $\mathbb{D}_0 \to \mathbb{D}_0$
by $z(X) = \cup\{p(x) \mid x \in X\}$, where $p(0) = 1$, $p(n+1) = 0$, $p(\perp) = \perp$

iv) if: $\mathbb{D}_0 \to (\mathbb{D}_\tau \to (\mathbb{D}_\tau \to \mathbb{D}_\tau))$
by $if_\sigma(X,\alpha,\beta) = \cup\{q(x,\alpha,\beta) \mid x \in X\}$, where
$q(t,\alpha,\beta) = \alpha$, $q(f,\alpha,\beta) = \beta$, $q(\perp,\alpha,\beta) = \perp$.

Proposition 3.2.2

a) pone, mone, $z \in \mathbb{D}_{0\to 0}$

b) for every σ, $if_\sigma \in \mathbb{D}_{0\to(\sigma\to(\sigma\to\sigma))}$

c) for every σ, if y_σ is the fixpoint operator on the cpo $\langle \mathbb{D}_\sigma, \sqsubseteq \rangle$, $y_\sigma \in \mathbb{D}_{(\sigma\to\sigma)\to\sigma}$.

Proof Omitted. □

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 VAR be the set of variables. Then ρ is an environment, written $\rho \in ENV$, iff ρ is a type-respecting function from VAR to $\cup_\tau \mathbb{D}_\tau$. For α a variable and x an element of $\cup_\tau\{\mathbb{D}_\tau\}$, $\rho(x/\alpha)$ is the environment identical to ρ except that α is mapped onto x.

Then define $V: L \to (ENV \to \cup_\tau\{\mathbb{D}_\tau\})$ by

$V[\alpha^\sigma](\rho) = \rho(\alpha_i^\sigma)$
$V[K_n](\rho) = \{n\}$
$V[[+1]](\rho) = pone$
$V[[-1]](\rho) = mone$
$V[Z](\rho) = z$
$V[IF_\sigma](\rho) = if_\sigma$
$V[MN](\rho) = V[M](\rho)\ (V[N](\rho))$
$V[\lambda\alpha.M](\rho)(x) = V[M]\rho(x/\alpha)$
$V[Y_\sigma](\rho) = y_\sigma$
$V[M \text{ or } N](\rho) = V[M](\rho) \cup V[N](\rho)$

Note that for every term t_σ and $\rho \in ENV$, $V[t_\sigma](\rho) \in \mathbb{D}_\sigma$.

Theorem 3.2.3

If $t \rightsquigarrow s$ then $V[t](\rho) \supseteq V[s](\rho)$ $\forall \rho \in ENV$.

Proof If $\alpha \overset{0}{\rightsquigarrow} \beta$ then it is easy to prove that $V[\alpha](\rho) \supseteq V[\beta](\rho)$. Because every well-formed term is interpreted as a $\subseteq$-monotonic function, we can apply the above remark to show $t \overset{1}{\rightsquigarrow} s \Rightarrow V[t](\rho) \supseteq V[s](\rho)$. 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 cgts t_1 and t_2, we prove that $EVAL(t_1) \subseteq EVAL(t_2)$ iff $V[t_1](\rho) \subseteq V[t_2](\rho), \forall \rho \in ENV$. Notice that since t_1 and t_2 are closed it suffices to consider only the trivial environment which maps all variables into $\perp$. We also denote this environment by $\perp$. It is also convenient to define $V[*](\rho) = \perp$, $\forall \rho \in ENV$.

Lemma 4.1.1 For any cgt t and constant k

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

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

Lemma 4.1.2 For any cgt t if $* \in EVAL(t)$ then $V[t](\perp) \supseteq \{\perp\}$.

Outline of proof: We consider only the case of $t\uparrow$.

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

Let $Y_\sigma^0 = \Omega_\sigma$

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

Then it can be proven that

$$V[Y_\sigma](\rho) = \sqcup_n V[Y_\sigma^n](\rho), \quad \forall \rho \in ENV.$$

Furthermore, let $Y_{\sigma_1},\ldots,Y_{\sigma_n}$ be all the Y terms occurring in t. Then

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

Now if t has an infinite standard derivation, an instance of the reduction $(Ys) \overset{o}{\rightsquigarrow} s(Ys)$ must be used an infinite number of times. Consequently, it can be proven that $[Y^k_{\sigma_1}|Y_{\sigma_1},\ldots,Y^k_{\sigma_n}|Y_{\sigma_n}]t \rightsquigarrow\rightsquigarrow u$, for some unsolvable u.

So $V[[Y^k_{\sigma_1}|Y_{\sigma_1},\ldots,Y^k_{\sigma_n}|Y_{\sigma_n}]t](\perp) \supseteq \{\perp\}$. Using the definition of $\sqsubseteq$ on D_0 we can then show that $V[t](\perp) \supseteq \{\perp\}$. □

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

Outline of proof: The method is similar to that of Theorem 3.1 of Plotkin (1975). A predicate P_σ is defined by induction on types and then it is shown by structural induction on the terms that every term t_σ has property P_σ.

Definition of P_σ:

a) If t_σ is a cgt then t_σ has P_σ if $\perp \in V[t](\perp) \Rightarrow * \in EVAL(t)$. (Statement of Lemma).

b) If $t_{(\sigma\to\tau)}$ is closed then it has $P_{(\sigma\to\tau)}$ if whenever the closed term s_σ has P_σ, (ts) has P_σ.

c) If t_σ is open with free variables $x_1^{\sigma_1},\ldots,x_n^{\sigma_n}$, then it has property P_σ if $[s_1|x_1,\ldots,s_n|x_n]t$ has property P_σ whenever s_i has property P_{σ_i}.

Let t be <u>okay</u> if it has the property P_σ. As an example of the use of structural induction to prove that every term is okay, we outline a method to show that Y_σ is okay. It is sufficient to show that $Y_\sigma t_1\ldots t_k$ is okay, where t_i are okay and $Y_\sigma t_1\ldots t_k$ is of ground type. Let $\perp \in V[Y_\sigma t_1\ldots t_k](\perp)$. Then

$$\perp \in \sqcup_n V[Y^n_\sigma t_1\ldots t_k](\perp) \Rightarrow \perp \in V[Y^n_\sigma t_1\ldots t_k](\perp),\ \forall n.$$

Using the induction hypothesis we can show that Y_σ^n is okay for every n. We have three cases:

i) $Y_\sigma^n t_1 \ldots t_k \rightsquigarrow \Omega$ for some n.

Then $Y_\sigma t_1 \ldots t_k \rightsquigarrow \Omega$ i.e. Y_σ is okay.

ii) For some n, $Y_\sigma^n t_1 \ldots t_k \uparrow$ and in this sequence the reduction $Y^k t_1 \rightsquigarrow t_1(Y_\sigma^{k-1} t_1)$ is used a finite number of times. In this case $Y_\sigma^0 t_1 \rightsquigarrow Y_\sigma^0 t_1$ is never used and so exactly the same sequence of reductions can be applied to $Y_\sigma t_1 \ldots t_k$.

iii) For every n, $Y_\sigma^0 t_1 \rightsquigarrow Y_\sigma^0 t_1$ is used (and therefore used an finite number of times). Then consider the tree of standard reductions from $Yt_1 \ldots t_k$. It is infinite and every node has a finite outdegree. By Königs lemma there is an infinite path which represents an infinite standard derivation from $Yt_1 \ldots t_k$. □

Theorem 4.1.4 For cgt t, $V[t](\perp) = \cup\{V[\alpha](\perp) \mid \alpha \in EVAL(t)\}$.

Proof Immediate from the previous lemmas. □

Corollary 4.1.5 For cgts t_1 and t_2,

$$EVAL(t_1) \subseteq EVAL(t_2) \text{ iff } V[t_1](\perp) \subseteq V[t_2](\perp).$$ □

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 $t_1 \underset{op}{\subseteq} t_2$.

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

$t_1 \underset{op}{\subseteq} t_2 \Rightarrow V[t_1](\rho) \underset{op}{\subseteq} V[t_2](\rho)$, $\forall \rho \in ENV$.

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

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 $f: D_0 \to D_0$ defined by

$$0 \in f(X) \quad \text{if} \quad 0 \in X$$
$$1 \in f(X) \quad \text{if} \quad 1 \in X$$
$$2 \in f(X) \quad \text{if} \quad \{0,1\} \subseteq X$$
$$\perp \in f(X) \quad \text{if} \quad \alpha \in X, \ \alpha \neq 0,1,\perp$$

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 λ-calculus can be constructed but the properties of this model have yet to be investigated.

REFERENCES

Duong, B. (1974) A high-level, variable-free calculus for recursive programming. *Research Report CS-74-03*, University of Waterloo.

Egli, H. (1975) A mathematical model for nondeterministic computations. *Unpublished report.*

Engelfriet, J. & E. Schmidt (1975) IO and OI, *DAIMI PB-47*, Mathematisch Institut, Aarhus Universitet, Denmark.

Manna, Z. & J. Vuillemin (1975) Fixpoint approach to the theory of computation. *CACM 15*, 528-536.

Milner, R. (1973) Models of LCF. *Stanford Artificial Intelligence Laboratory Memo AIM-186.*

Nivat, M. (1974) On the interpretations of recursive program schemes. *Rapport de Recherche No.84, IRIA.*

Plotkin, G.D. (1975) LCF considered as a Programming Language. *Symposium on Proving and Improving Programs*, Arc et Senans (eds. Huet and Kahn).

Scott, D. (1970) Outline of a mathematical theory of computation, in *Proc. of the Fourth Annual Princeton Conference on Information Sciences and Systems*, Princeton.

Scott, D. (1975) Data types as lattices, in *Lecture notes in mathematics No.499*, 579-651, Springer-Verlag.

Vuillemin, J. & B. Courcelle (1974) Semantics and Axiomatics of a simple programming language. *Rapport de Recherche No.60, IRIA.*

Wadsworth, C.P. (1975) The relationship between λ-expressions and their denotations in Scott's models of the λ-calculus. *Proceedings of the Orleans Conference* (eds. Calais, Dernck and Sobbagh).