JOURNAL OF COMPUTER AND SYSTEM SCIENCES 19, 50–62 (1979)

Parameter Passing in Nondeterministic Recursive Programs*

DAVID B. BENSON

Computer Science Department, Washington State University, Pullman, Washington 99164

Received January 4, 1978; revised December 8, 1978

Call by value and call by name have some subtleties when used in a nondeterministic programming language. A common formalism is used to establish the denotational semantics of recursive programs called by name and called by value. Ashcroft and Hennessy introduced the idea of differentiating between selecting arguments from a set of arguments at the point of invocation and selecting arguments during the run of the procedure. This distinction is shown to be independent of the evaluation according to value or name, giving rise to four possible parameter passing methods, all of which have a suitable least fixed point semantics.

INTRODUCTION

With the current interest in nondeterministic programs [1, 4, 5, 6, 8, 10], the question of formulating the denotational semantics of such programs naturally arises. One question in formulating denotational semantics is the parameter passing problem. Hennessy [4] and Hennessy and Ashcroft [5, 6] have formulated one approach to such a study. We offer another approach which seems both clearer and simpler. Indeed, we just follow Manna [9] to obtain the proofs in a strictly denotational semantics style. Hennessy and Ashcroft [6] use reduction sequences and some rather complex operators therein.

In this study, we explore denotational semantics for a variety of parameter passing mechanisms, using only sets of basic values as parameters. The problem is the expression of when particular arguments are to be chosen from sets of arguments. We show that this choice is independent of the choice of call by name or call by value. Indeed, the approach here gives a hierarchy of call mechanisms, which suggests that designers of nondeterministic programming languages give considerable attention to the issue of parameter passing so as to give flexibility without chaos.

The issues involved in passing programs as parameters will require a rather more intricate study using the most general form of a power domain construction. We believe that this would detract from the central issue of the variety of parameter passing mechanisms possible, and leave it for study after the best formulation for general power domains has been found. See [8, 10, 12].

Our notation differs slightly from other authors. The symbol $\equiv$ denotes equality in the extended setting in which undefined, $\perp$,is a value, while $=$ denotes the test for

* Research supported in part by NSF Grants MCS73-03497A01 and MCS77-08486.

equality. So $\equiv$ is always defined to be either *true* or *false*, whereas $=$ results in $\perp$ whenever at least one of its arguments is $\perp$. See [9]. When we are thinking of a program, the keywords are in boldface, viz: **if**, **or**. If the expression is better thought of as a function, the keywords are in Roman face. τ^∞ is the least fixed point of the functional $\tau[F]$ over the totally undefined function. If $\{A_i : i \in I\}$ is an indexed family of sets, the union of the family is denoted by $\{A_i : i \in I\}$ or $\bigcup_{i \in I} A_i$, and in the more complex situations both symbolisms are used in a single formula to keep the depth of set nesting correct, and also hopefully clear. Toward the end of the paper, we become rather careless about the distinction between the element x and the singleton set $\{x\}$, under the assumption that by that time the developed intuition makes the situation clear to the reader, and that the less messy notation is preferable.

An apology. Objections are frequently raised to the terminology "call by name" and "call by value" as used in theoretical studies such as [4, 5, 6, 9, 11], since the full impact of the ALGOL 60 distinctions are not modeled. It is standard in science to select what appear to be the crucial features of some phenomenon for abstract study and use the same name for the abstraction. The fact that ALGOL 60 implementations use procedures, the thunks of P. Z. Ingerman [7], to implement call by name is simply not modeled in these studies of semantics. The readers upset by this lack of detail will please prefix "theoretical" or "abstract" in front of each use of the phrases call by name and call by value.

Nondeterministic Functions

As is usual in studies of denotational semantics, each domain of values is a *cpo* $\langle A, \sqsubseteq, \perp \rangle$ where $\sqsubseteq$ partially orders A, $\perp$ is the least element in A, and every $\sqsubseteq$-chain in A has a least upper bound in A. Our interest lies in passing sets of basic values as parameters, so the domains of values are taken to be flat *cpos*, i.e., in $\langle A, \sqsubseteq, \perp \rangle$ A is denumerable and $\forall x, y \in A$, $x \sqsubseteq y$ implies $x \equiv y$ or $x \equiv \perp$. A power domain in this case, [10], is a system $\langle PA, \underline{\sqsubseteq}, \subseteq, \{\perp\} \rangle$ in which $PA = \{X \subseteq A \mid X \not\equiv \varnothing \text{ and } (X \text{ finite or } \perp \in X)\}$,

$$X \underline{\sqsubseteq} Y \text{ iff } (\forall x \in X\ \exists y \in Y \quad x \sqsubseteq y) \mathbin{\&} (\forall y \in Y\ \exists x \in X \quad x \sqsubseteq y),$$

from which one sees that $\langle PA, \underline{\sqsubseteq}, \{\perp\} \rangle$ is a *cpo*, that unions are continuous with respect to $\underline{\sqsubseteq}$, and that $\subseteq$ preserves least upper bounds with respect to $\underline{\sqsubseteq}$.

As a first approximation, a nondeterministic function is a function from a flat *cpo* $\langle A, \sqsubseteq, \perp \rangle$ to a power domain $\langle PB, \underline{\sqsubseteq}, \subseteq, \{\perp\} \rangle$ over a flat *cpo*, which is $(\sqsubseteq, \underline{\sqsubseteq})$-monotonic and therefore $(\sqsubseteq, \underline{\sqsubseteq})$-continuous. This idea is in strict analogy to nondeterministic automata and appears to capture the ideas of nondeterministic programming as expressed in, e.g., Dijkstra [1]. In order to compose nondeterministic functions, extend each $f\colon A \to PB$ to $\hat{f}\colon PA \to PB$ by $\hat{f}(X) = \bigcup_{x \in X} f(x)$. These extensions have all the properties one could desire for denotational semantics except for the ability to properly treat functions of more than one argument. If we followed the above notion of extension with multiargument functions, say g: $A \times B \to PC$, the construction $\hat{g}$: $P(A \times B) \to PC$ fails to have desired properties. For one, $A \times B$ is not flat so $P(A \times B)$ requires a much more complex construction [10]. Other difficulties arise. Intuitively, the failure is due to the fact that each element of $P(A \times B)$ is a relation, giving coupling between the arguments. Such couplings should only occur explicitly by the action of the functions making up the program, and not by default. Therefore, we have the following definitions.

1. Definition. A nondeterministic function of n arguments f: $A_1 \times \cdots \times A_n \to PB$ is a ($\sqsubseteq$, $\sqsubset$)-homomorphism from a product of flat *cpos* $\langle A_i, \sqsubseteq, \perp\rangle$ to the power domain $\langle PB, \underline{\sqsubset}, \subseteq, \{\perp\}\rangle$ over a flat *cpo* $\langle B, \sqsubseteq, \perp\rangle$. The extension of f is $\hat{f}$: $PA_1 \times \cdots \times PA_n \to PB$ such that for $X_i \in PA_i$, $1 \leqslant i \leqslant n$,

$$\hat{f}(X_1, \ldots, X_n) \equiv \bigcup_{\substack{x_1 \in X_1 \\ \vdots \\ x_n \in X_n}} f(x_1, \ldots, x_n).$$

An extended nondeterministic function is termed an *end* function.

To relate these definitions to programming, consider the usual monotonic deterministic base functions: conditional, boolean tests, and arithmetic. Each is easily turned into a nondeterministic function by the use of singletons. For example, nondeterministic addition is

$$a + b \equiv \{a + b\}$$

while nondeterministic conditional is

$$\textbf{if } a \textbf{ then } b \textbf{ else } c \equiv \begin{cases} \{\perp\} \text{ if } a \equiv \perp, \\ \{b\} \text{ if } a \equiv \mathit{true}, \\ \{c\} \text{ if } a \equiv \mathit{false}. \end{cases}$$

The only strictly nondeterministic base function considered is union: a **or** $b \equiv \{a, b\}$, with extension X **or** $Y \equiv X \cup Y$. The nondeterministic programs we model are constructed over these conditional, boolean test, arithmetic, and union operations. This suffices for Dijkstra's guarded commands and it is difficult to conceive of another strictly nondeterministic function which could not be constructed with the collection of base functions given.

The end functions have some pleasant properties which substantially reduce the effort required to study parameter passing methods.

2. Definition. Let A, B be collections of sets. A function f: $A \to B$ is uniformly additive if for every subset $\{A_i\}$ of A such that $\cup A_i \in A$, $f(\cup A_i) = \cup f(A_i)$.

3. Proposition. *End functions are uniformly additive in each variable separately.*

4. PROPOSITION. *Let f be an end function defined over the base functions. f may be written in "conjunctive normal form"*

$$f \equiv f_1 \textbf{ or } f_2 \textbf{ or } \cdots f_n$$

where each f_i may involve the composition of any base functions except union.

5. THEOREM. *End functions are continuous.*

Proof. By the properties of the Milner ordering used in the power domains, any chain $\{X_i\}$ either terminates or else $\perp \in X_k$, $k \geqslant 2$, in which case $X_i \sqsubseteq X_j$ implies $X_i \subseteq X_j$. Thus $\text{lub}\{X_i\} \equiv \cup X_i$. We now treat the case of two variable end functions. $f(\text{lub}\{X_i\}, \text{lub}\{Y_j\}) \equiv f(\cup X_i, \cup Y_j) \equiv \bigcup_i \bigcup_j f(X_i, Y_j)$, so $f(\text{lub}\{X_i\}, \text{lub}\{Y_j\}) \subseteq \text{lub} f\{(X_i, Y_j)\}$. Now any $w \in \text{lub}\{f(X_i, Y_j)\}$ is a member of some $f(X_k, Y_l)$. ∎

CALL BY NAME

To compute a nondeterministic program of one argument, called by name, an obvious strategy is to select an argument from the single set of arguments, compute the value, select another to compute the value at that point, and accumulate these values via union. Continue this process until all arguments have been exhausted. This strategy shows that there is nothing more here than the definition of an end function. The situation is only slightly more complicated for a recursive nondeterministic program called by name. We give an example before proceeding to the general case in Definition (6).

Example (deRoever [11], Morris [9, p. 389]). The program is written as though it were deterministic, but we intend for it to be called on sets of arguments.

```
integer procedure $fn(x, y)$; integer $x, y$;
$fn := $ if $x = 0$ then $0$ else $fn(x - 1, fn(x, y))$
```

From the computational idea of calculating on one pair of arguments x, y (drawn from the input sets X and Y) at a time, we see that the conditional will be calculated for each triple x, y, z where $z \in fn(\{x - 1\}, fn(\{x\}, \{y\}))$. Thereby the end functional for this nondeterministic program is:

$$\tau[F](X, Y) \equiv \bigcup_{\substack{x \in X \\ y \in Y}} \bigcup \{\text{if } \{x\} = \{0\} \text{ then } \{0\} \text{ else } \{z\} \colon z \in F(\{x - 1\}, F(\{x\}, \{y\}))\}$$

which may be written

$$\tau[F](X, Y) \equiv \bigcup_{\substack{x \in X \\ y \in Y}} \{\text{if } x = 0 \text{ then } 0 \text{ else } z \colon z \in F(\{x - 1\}, F(\{x\}, \{y\}))\}$$

since the conditional is actually used deterministically. The least fixed point solution is easily shown to be

$$\tau^\infty(X, Y) \equiv \{0 \colon \exists x \in X, x \geqslant 0\} \cup \{\perp \colon \perp \in X \vee \exists x \in X, x < 0\}$$

when the underlying arithmetic is on the integers.

6. Definition. The general form for end functionals called by name is the system for $1 \leqslant l \leqslant m$,

$$\begin{aligned}\tau_l[F_1, \ldots, F_m](X_1, \ldots, X_n) \\ \equiv \bigcup_{\substack{x_1 \in X_1 \\ \vdots \\ x_n \in X_n}} f_l(x_1, \ldots, x_n, & \tau_{l1}[F_1, \ldots, F_m](x_1, \ldots, x_n), \\ & \vdots \\ & \tau_{lk}[F_1, \ldots, F_m](x_1, \ldots, x_n))\end{aligned}$$

where the f_l are end functions of $n + k$ variables over the nondeterministic base functions and the τ_{li} are end functionals called by name. For clarity, we have avoided writing the brackets around the singleton sets $\{x_j\}$ which are the arguments of the f_l and the τ_{li}. The domain and codomain of the system $\{\tau_l\}$ are the m-tuples of end functions in $[PA_1 \times \cdots \times PA_n \rightarrow PB]^m$.

Note that each functional in the system may be written as

$$\begin{aligned}\tau_l[F_1, \ldots, F_m](X_1, \ldots, X_n) \\ \equiv \bigcup_{\substack{x_1 \in X_1 \\ \vdots \\ x_n \in X_n}} \bigcup \{f_l(x_1, \ldots, x_n, y_1, \ldots, y_k): \\ y_i \in \tau_{li}[F_1, \ldots, F_m](x_1, \ldots, x_n)\}\end{aligned}$$

due to uniform additivity.

7. Proposition. *End functionals called by name are monotonic.*

8. Theorem. *End functionals called by name are continuous.*

Proof. For simplicity of presentation, we consider only end functionals of one argument over one function variable. The proof technique easily extends to end functionals of n arguments with m function variables. Consider the end functional

$$\tau[F](X) \equiv \bigcup_{x \in X} f(\{x\}, \tau'[F](\{x\}))$$

where τ' is continuous. Let $\{h_i\}$ be any chain of end functionals.

$$\begin{aligned}\tau[\mathrm{lub}\{h_i\}](\{x\}) &\equiv \bigcup_{x \in \{x\}} f(\{x\}, \tau'[\mathrm{lub}\{h_i\}](\{x\})) \\ &\equiv \bigcup_{x \in \{x\}} f(\{x\}, \mathrm{lub}\{\tau'[h_i]\}(\{x\}))\end{aligned}$$

as τ' is continuous. Now $y \in \mathrm{lub}\{\tau'[h_i]\}$ $(\{x\})$ implies there exists k such that $y \in \tau'[h_{k+j}]$ $(\{x\})$, for all $j \geqslant 0$. Consider the minimum k which has this property for all $y \in \mathrm{lub}\{\tau'[h_i]\}$ $(\{x\})$, if it exists. Then

$$\begin{aligned}\tau[\mathrm{lub}\{h_i\}](\{x\}) &\equiv \bigcup_{x \in \{x\}} f(\{x\}, \tau'[h_k](\{x\})) \\ &\equiv \tau[h_k](\{x\}) \sqsubseteq \mathrm{lub}\{\tau[h_i]\}(\{x\}).\end{aligned}$$

If no such minimum k exists, then for all $k \geqslant 0$ there is a $y \in \text{lub}\{\tau'[h_i]\}$ $(\{x\})$ such that $y \notin \tau'[h_k]$ $(\{x\})$. As $\tau'[h_k]$ $(\{x\}) \sqsubseteq \text{lub}\{\tau'[h_i]\}$ $(\{x\})$ and as the power domain is over a flat *cpo*, $\perp \in \tau'[h_k]$ $(\{x\})$, for all $k \geqslant 0$. Obviously $\text{lub}\{\tau'[h_i]\}$ $(\{x\})$ is an infinite set, so $\perp \in \text{lub}\{\tau'[h_i]\}$ $(\{x\})$. With these facts in hand, $\text{lub}\{\tau'[h_i]\}$ $(\{x\}) = \bigcup_{k \geqslant 0} \tau'[h_k]$ $(\{x\})$ is easy to demonstrate. Then

$$\begin{aligned}
\tau[\text{lub}\{h_i\}](\{x\}) &= \bigcup_{x \in \{x\}} f\Big(\{x\}, \bigcup_{k \geqslant 0} \tau'[h_k](\{x\})\Big) \\
&= \bigcup_{k \geqslant 0} \bigcup_{x \in \{x\}} \{f(\{x\}, \tau'[h_k](\{x\}))\} \\
&= \bigcup_{k \geqslant 0} \tau[h_k](\{x\}) \\
&\sqsubseteq \text{lub}\{\tau[h_i]\}(\{x\}).
\end{aligned}$$

By these two cases and uniform additivity, $\tau[\text{lub}\{h_i\}]$ $(X) \sqsubseteq \text{lub}\{\tau[h_i]\}$ (X). The other direction follows directly from monotonicity, so $\tau[\text{lub}\{h_i\}]$ $(X) = \text{lub}\{\tau[h_i]\}$ (X). ▮

EXAMPLE. G computes values in Pascal's triangle, H computes all combinations of n objects taken $k, k-1, \ldots, 1, 0$ at a time as a nondeterministic program, while F computes, nondeterministically, a row of Pascal's triangle.

```
G(n, k) := if k = 0 then 1
           else if k = n + 1 then 0
           else G(n − 1, k − 1) + G(n − 1, k)
H(n, k) := if k = 0 then 1
           else G(n, k) or H(n, k − 1)
   F(n) := H(n, n)
```

For instance, $F(4) = \{1, 4, 6\}$ and $F(\{2, 4\}) = \{1, 2, 4, 6\}$. The readers may convince themselves that the least fixed point for $G(n, k)$ is $\binom{n}{k}$ whenever $n \geqslant k$, 0 if $n + 1 = k$, and $\perp$ if $n + 1 < k$. This last is an artifice to make a point later on. Safer programming technique would be to guard the second leg of G by $(k > n)$ rather than $(k = n + 1)$, in which case G would be totally defined on all pairs of natural numbers.

CALL BY VALUE

Call by value requires evaluating the parameters to something defined (not $\perp$) before calling the program. This technique requires the following definition for end functions, essentially due to Hennessy [4].

9. Definition. Let $\hat{f}$: $PA_1 \times \cdots \times PA_n \to PB$ be an end function over the nondeterministic function f: $A_1 \times \cdots \times A_n \to PB$. The value of $\hat{f}$, called by value, is

$$\hat{f}_v(X_1, ..., X_n) = \bigcup_{\substack{x_1 \in X_1 \\ \vdots \\ x_n \in X_n}} \Big[\{f(x_1, ..., x_n): x_i \not\equiv \perp, 1 \leqslant i \leqslant n\} \\ \bigcup \{\perp: x_1 \equiv \perp \vee \cdots \vee x_n \equiv \perp\}\Big].$$

We follow this general plan of explicitly checking for $\perp$ in the definition of end functionals called by value.

10. Definition. The general form for end functionals called by value on one function variable is

$$\rho[F](X_1, ..., X_n) = \bigcup_{\substack{x_1 \in X_1 \\ \vdots \\ x_n \in X_n}} \Big[\bigcup \{f(x_1, ..., x_n, y_1, ..., y_k): x_j \not\equiv \perp, \\ y_i \in \rho_i[F](\{x_1\}, ..., \{x_n\})\} \\ \bigcup \{\perp: x_1 \equiv \perp \vee \cdots \vee x_n \equiv \perp\}\Big]$$

where f is a nondeterministic function of $n + k$ variables over the base functions and the ρ_i are end functionals called by value. These end functionals are again over the end functions in $[PA_1 \times \cdots \times PA_n \to PB]$.

The general form for m function variables may be obtained by analogy with the previous case of end functionals called by name. The end functionals called by value are also obviously monotonic.

11. Theorem. *End functionals called by value are continuous.*

Proof. Variations on the theme of the previous theorem. ∎

Example (deRoever) [11].

integer procedure $fv(x, y)$; **value** x, y; **integer** x, y;
$fv :=$ **if** $x = 0$ **then** 0 **else** $fv(x - 1, fv(x, y))$

becomes the end functional

$$\rho[F](X, Y) \equiv \bigcup_{\substack{x \in X \\ y \in Y}} \Big[\bigcup \{\text{if } \{x\} = \{0\} \text{ then } \{0\} \text{ else } \{z\}: \\ x \not\equiv \perp, y \not\equiv \perp, z \in F(\{x - 1\}, F(\{x\}, \{y\}))\} \\ \bigcup \{\perp: x \equiv \perp \vee y \equiv \perp\}\Big]$$

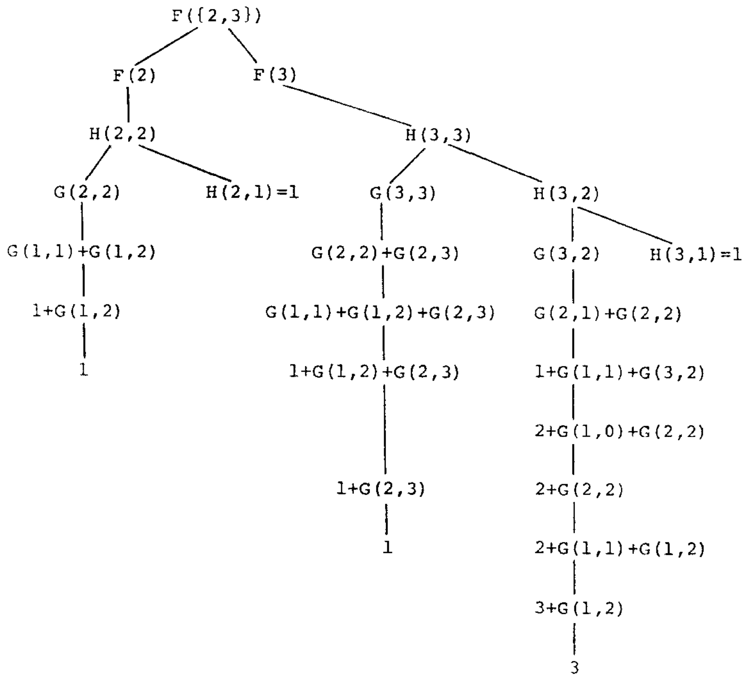
$$\equiv \bigcup_{\substack{x \in X \\ y \in Y}} \Big[\{\text{if } x = 0 \text{ then } 0 \text{ else } z: x \not\equiv \perp, y \not\equiv \perp, z \in F(\{x - 1\}, F(\{x\}, \{y\}))\} \\ \bigcup \{\perp: x \equiv \perp \vee y \equiv \perp\}\Big]$$

since the conditional is used deterministically. The least fixed point solution is

$$\rho^{\infty}(X, Y) \equiv \{0 : 0 \in X \;\&\; Y - \{\perp\} \neq \varnothing\}$$
$$\bigcup \{\perp : \perp \in X \vee \perp \in Y \vee X - \{0\} \neq \varnothing\}$$

A pressing question for nondeterministic recursive programs called by value is whether the leftmost-innermost computation rule will obtain the least fixed point solution of this class of end functionals. Leftmost-innermost replaces, at each substitution step, the leftmost occurrence of the function variable F which has all its arguments free of F.

A computation sequence for the leftmost-innermost computation rule is a tree. Each branch of the tree is a substitution-simplification pair for each tuple of arguments in the input sets of arguments. Using the previous "Pascal row" example, the computation sequence for $F(\{2, 3\})$ is:



The value of the computation tree is the union of all leaves, together with $\perp$ if there are any infinite paths.

12. THEOREM. *The leftmost-innermost computation rule computes the least fixed point of those end functionals called by value which are defined over conditional, naturally extended (with respect to $\perp$) base functions, and union.*

Proof. The terminology is from [9, p. 375ff]. Consider a term occurring in any computation substitution step of an end functional called by value, $g(F(t_1),\ldots, F(t_n))$ where g is an end function and the t_i are terms. In particular, $t_1 = g_1(F(\ldots F(\alpha_1)\ldots),\ldots)$, where g_1 is an end function and $F(\alpha_1)$ is the lefmost-innermost occurrence of the function variable F.

If $F(\alpha_1)$ is replaced by $\perp$, the form of end functionals called by value guarantees that the $\perp$ propagates to $g_1(\perp,\ldots)$. Now g_1 is defined over conditional, naturally extended base functions, and union, so $\perp \in g_1(\perp,\ldots)$. By the same argument, $\perp \in g(\perp, F(t_2),\ldots,F(t_n)) = g(F(\perp),\ldots, F(t_n)) \subseteq g(F(g_1(\perp,\ldots)), F(t_2),\ldots, F(t_n))$.

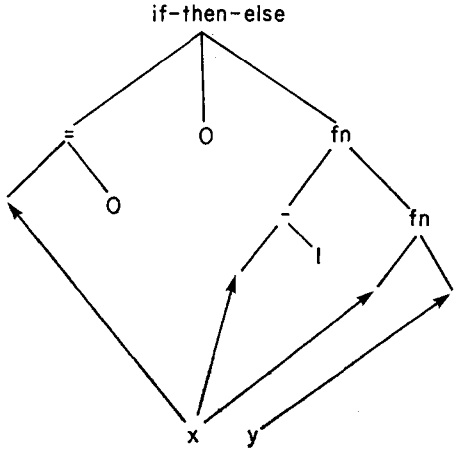
This gives a slight variant on the notion of safe computation rule, and making minor modifications to Vuillemin's proof [13] that a deterministic safe computation rule is a fixed point rule proves the theorem. ▮

While deRoever [11] pointed out that call by value has a fixed point semantics of its own, no previous proof exists of the—by now fairly obvious—fact that lefmost-innermost does indeed compute the call by value least fixed point. As deterministic programs are just our nondeterministic programs without union, this gap in the semantical theory of programs is closed.

Run Time Choice

Hennessy and Ashcroft [6] introduce the idea of letting the particular values used in each branch of the computation tree be chosen during the computation rather than have them selected in the call, as we did in call by name. The apparent result of this late choosing is to select different values from the set of arguments for each occurrence of the variable in the program. Suppose F is a nondeterministic program with corresponding non-deterministic function f: $A_1 \times \cdots \times A_n \rightarrow PB$, when called by name. In computing this function, some variables may be duplicated, rearranged, or not used at all.

EXAMPLE.



In the deRoever–Morris example, x appears three distinct times in the definition of *fn* while y appears once. This is immediately clear from looking at the computation "tree"—actually a doag—drawn above. In linear form, we may write

$$\textbf{if } x_1 = 0 \textbf{ then } 0 \textbf{ else } fn(x_2 - 1, fn(x_3, y))$$

where x_1, x_2, and x_3 indicate the three occurrences of x in the definition of *fn*. Consider such a function *fn'* with arguments $fn'(x_1, x_2, x_3, y)$. The computation tree for *fn'* actually is a tree. We may identify the first three arguments to obtain $fn(x, y)$ again by pre-composing *fn'* with the map (x, x, x, y): $Z^2 \to Z^4$. In the diagram, *fn'* is the tree at the top where lines are drawn without arrows while the map (x, x, x, y) is the drawing with arrows at the bottom.

Returning to the general nondeterministic program F, we want to divide it into the 'computation tree' part and the 'rearrangement of variables' part. Let f': $A_{\pi(1)} \times \cdots \times A_{\pi(k)} \to PB$ be the nondeterministic function in which all variable occurrences are explicitly in the same order as in the definition of F and such that

$$f = A_1 \times \cdots \times A_k \xrightarrow{\pi} A_{\pi(1)} \times \cdots \times A_{\pi(k)} \xrightarrow{f'} PB.$$

Thus f' is the 'computation tree' part and π is the 'rearrangement of variables' part.

In f': $A_{\pi(1)} \times \cdots \times A_{\pi(k)} \to PB$ there are now distinct variables for each occurrence of variables in F. In making the run time choice then, it is possible to choose independently from each of the k copied arguments. The value of F called with run time choice on argument sets $X_1, \ldots, X_n$ depends on the rearrangement π taking $A_1 \times \cdots \times A_n$ into $A_{\pi(1)} \times \cdots \times A_{\pi(k)}$. The value is

$$F_{rn}(X_1, \ldots, X_n) = \bigcup_{\substack{x_1 \in X_{\pi(1)} \\ \vdots \\ x_k \in X_{\pi(k)}}} f'(x_1, \ldots, x_k)$$

EXAMPLE. Consider the previous Morris–deRoever example.

$$\begin{aligned} fn(\{2, 4\}, 6) = \{&\text{if } 2 = 0 \text{ then } 0 \text{ else } fn(1, fn(2, 6)),\\ &\text{if } 2 = 0 \text{ then } 0 \text{ else } fn(1, fn(4, 6)),\\ &\text{if } 2 = 0 \text{ then } 0 \text{ else } fn(3, fn(2, 6)),\\ &\text{if } 2 = 0 \text{ then } 0 \text{ else } fn(3, fn(4, 6)),\\ &\text{if } 4 = 0 \text{ then } 0 \text{ else } fn(1, fn(2, 6)),\\ &\text{if } 4 = 0 \text{ then } 0 \text{ else } fn(1, fn(4, 6)),\\ &\text{if } 4 = 0 \text{ then } 0 \text{ else } fn(3, fn(2, 6)),\\ &\text{if } 4 = 0 \text{ then } 0 \text{ else } fn(3, fn(4, 6))\}\\ = &\text{ if } \{2, 4\} = 0 \text{ then } 0 \text{ else } fn(\{1, 3\}, fn(\{2, 4\}, 6))\\ = &\ \{0\}. \end{aligned}$$

Call by value nondeterministic programs can be handled the same way. The value of F called by value with run time choice is

$$F_{rv}(X_1, \ldots, X_n) \equiv \bigcup_{\substack{x_1 \in X_{\pi(1)} \\ \vdots \\ x_k \in X_{\pi(k)}}} \left[\bigcup \left\{ f'(x_1, \ldots, x_n) \colon \bigwedge (x_i \neq \perp) \right\} \cup \left\{ \perp \colon \bigvee (x_i = \perp) \right\} \right].$$

Example. Continuing the deRoever example,

$$\begin{aligned} fv(\{0, 2\}, 4) \equiv \{&\text{if } 0 = 0 \text{ then } 0 \text{ else } fv(-1, fv(0, 4)), \\ &\text{if } 0 = 0 \text{ then } 0 \text{ else } fv(-1, fv(2, 4)), \\ &\text{if } 0 = 0 \text{ then } 0 \text{ else } fv(1, fv(0, 4)), \\ &\text{if } 0 = 0 \text{ then } 0 \text{ else } fv(1, fv(2, 4)), \\ &\text{if } 2 = 0 \text{ then } 0 \text{ else } fv(-1, fv(0, 4)), \\ &\text{if } 2 = 0 \text{ then } 0 \text{ else } fv(-1, fv(2, 4)), \\ &\text{if } 2 = 0 \text{ then } 0 \text{ else } fv(1, fv(0, 4)), \\ &\text{if } 2 = 0 \text{ then } 0 \text{ else } fv(1, fv(2, 4))\} \\ \equiv{} &\text{if } \{0, 2\} = 0 \text{ then } fv(\{-1, 1\}, fv\{(0, 2\}, 4)) \\ \equiv{} &\{0, \perp\}. \end{aligned}$$

The form of the functionals in these two cases is now clear:

13. **Definition.** Run time choice call by name end functionals.

$$\tau_r[F](X_1, \ldots, X_n) \equiv \bigcup_{\substack{x_1 \in X_{\pi(1)} \\ \vdots \\ x_k \in X_{\pi(k)} \\ \vdots \\ x_l \in X_{\pi(l)}}} \bigcup \{ f(x_1, \ldots, x_k, y_1, \ldots, y_p) \colon y_i \in \tau_{ri}[F](x_{k+1}, \ldots, x_l) \}$$

where f is a nondeterministic function of $k + p$ variables which are not further duplicated or deleted, but may be permuted, and each τ_{ri} is a run time choice call by name end functional.

The proof of continuity is as easy as before and it is clear that $\tau^\infty \subseteq \tau_r^\infty$.

Example. The Pascal row as computed with run time choice gives for $F_r(\{2, 4\})$,

$$\begin{aligned} F_r(\{2, 4\}) &\equiv H(2, 2) \cup H(2, 4) \cup H(4, 2) \cup H(4, 4) \\ &\equiv \{1, 2\} \cup \{0, \perp\} \cup \{1, 4, 6\} \cup \{1, 4, 6\} \\ &\equiv \{\perp, 0, 1, 2, 4, 6\}, \end{aligned}$$

which was perhaps not what was intended, due to the spurious results included by the $H(2, 4)$ term. Even if the guard on the second leg in the definition of G is changed to $(k > n)$, we would still have $F_r(\{2, 4\}) \equiv \{0, 1, 2, 4, 6\}$, with the zero results included by the $H(2, 4)$ term. I draw the conclusion that run time choice is dangerous.

14. Definition. Run time choice call by value end functionals.

$$\rho_r[F](X_1, \ldots, X_n) \equiv \bigcup_{\substack{x_1 \in X_{\pi(1)} \\ \vdots \\ x_k \in X_{\pi(k)} \\ \vdots \\ x_l \in X_{\pi(l)}}} \left[\bigcup \left\{ f(x_1, \ldots, x_k, y_1, \ldots, y_p) : \begin{array}{l} \\ y_i \in \rho_{ri}[F](x_{k+1}, \ldots, x_l) \wedge \bigwedge (x_j \not\equiv \bot) \end{array} \right\} \cup \left\{ \bot : \bigvee (x_j \equiv \bot) \right\} \right]$$

where f is a nondeterministic function of $k + p$ variables which are not further duplicated or deleted, but may be permuted, and each ρ_{ri} is a run time choice call by value end functional.

Again the proof of continuity follows from the previous proofs and again $\rho^\infty \subseteq \rho_r^\infty$.

The run time choice end functional forms may not, at first glance, appear to satisfy the intuitive idea of passing the arguments as sets with choice of value made subsequent to the call. But as the functionals are uniformly additive, the choice may be accomplished as shown above, giving the same results as if the choice is made after the call.

Final Remarks

Hennessy and Ashcroft [5, 6] treat call time choice and run time choice as subdivisions of call by name. As the definitions here show, either call time choice or run time choice can be used with either call by value or call by name. Just as a single procedure may combine arguments called by value and called by name, so may some arguments be used as call time choice and others used as run time choice. Indeed, a single argument may occur as both a run time and a call time argument, by constraining some, but not all, occurrences of the argument to be identical during the selection process. For example, in the Morris–deRoever function, fn, the first and third occurrences of x may be so constrained to form

$$fn(X, Y) \equiv \bigcup_{\substack{x_1 \in X \\ x_2 \in X \\ y \in Y}} \{\text{if } x_1 = 0 \text{ then } 0 \text{ else } fn(x_2 - 1, fn(x_1, y))\}.$$

Thus one obtains a lattice of possible evaluation mechanisms, with "all arguments chosen at the call" as the lattice zero and "all arguments chosen at run time" as the lattice one. Crosscombining this with comparable structures for the range from all arguments called by value to all arguments called by name should produce a rather rich algebra of call mechanisms and their associated least fixed point values.

One strongly suspects that all of the results herein are universal in the sense that they could be developed in an appropriate iterative or rational algebraic theory [2, 3, 14] equipped with an appropriate notion of subset. The intuition is that these results do not begin to use the richness of power domain constructions in [8, 10, 12], and thus the structure of nondeterministic programs called only on basic values may well be best exposed in an algebraic theory setting.

Perhaps more interesting to the language designer engaged in attempting to specify a nondeterministic programming language are the following two observations: First, run time choice gives surprising and non-intuitive values to functions called on perfectly reasonable arguments. Further, since run time choice differs from the usual call time choice only in the automatic duplication and selection of arguments, a programmer who actually needed this effect could readily produce it by doing the argument manipulations himself. Second, call by value is available as a semantically respectable programming construct for nondeterministic languages and the usual mechanism to evaluate it produces the semantically expected value.

Acknowledgment

I appreciate the care exercised by a referee in keeping my lapses from being printed.

References

1. E. D. Dijkstra, "A Discipline of Programming," Prentice-Hall, Englewood Cliffs, N. J., 1976.
2. C. C. Elgot, Monadic computation and iterative algebraic theories, *in* "Proc. Logic Colloq., Bristol, 1973" (Rose and Shperdson, Eds.), North-Holland, Amsterdam, 1975.
3. C. C. Elgot, Matricial theories, *J. Algebra* **42** (1976), 391–422.
4. M. Hennessy, "The Semantics of Call-by-Value and Call-by-Name in a Nondeterministic Environment," Tech. Report CS-77-13, Univ. of Waterloo.
5. M. Hennessy and E. A. Ashcroft, The semantics of nondeterminism, *in* "Third International Colloq. on Automata, Languages and Programming, Edinburgh, 1976."
6. M. Hennessy and E. A. Ashcroft, Parameter-passing mechanicms and non-determinism, *in* Proc. 9th ACM Symp. Theory of Computing, Boulder, 1977," pp. 306–311.
7. P. Z. Ingerman, Thunks, *Comm. ACM* **4** (1961), 55–58.
8. D. J. Lehmann, Categories for fix point semantics, *in* "Proc. 17th IEEE Symp. on Foundations of Computer Science, Houston, 1976," pp. 122–126.
9. Z. Manna, "Mathematical Theory of Computation," McGraw-Hill, New York, 1974.
10. G. D. Plotkin, A powerdomain construction, *SIAM J. Comput.* **5** (1976), 452–487.
11. W. P. deRoever, Call-by-value versus call-by-name: A proof theoretic comparison, pp. 451–463, Lecture Notes in Computer Science No. 28, Springer-Verlag, New York, 1975.
12. M. B. Smyth, Power Domains, *J. Comput. System Sci.* **16** (1978), 23–36.
13. J. Vuillemin, "Proof Techniques for Recursive Programs," Ph. D. thesis, Stanford University, 1973.
14. J. B. Wright, J. A. Goguen, J. W. Thatcher, and E. G. Wagner, Rational algebraic theories and fixed-point solutions, *in* "Proc. 17th IEEE Symp. on Foundations of Computer Science, Houston, 1976," pp. 147–158.