THE SEMANTICS OF SHARING IN PARALLEL PROCESSING*

by

T.S.E. Maibaum

Computer Science Dept.
University of Waterloo
Waterloo, Ontario, Canada

Research Report CS-79-05

January 1979

*This research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT

Concurrent computation can be divided roughly into two categories according to whether the method of synchronisation is via communication or via sharing of resources (which we interpret to mean sharing of data values). In this report we propose a formalisation of the semantics of concurrent processing using the latter kind of synchronisation. Our formalisation is given in an algebraic setting and we make extensive use of the concept of abstract data type.

The programming language whose semantics is specified is a simple (standard) language with recursion and forking. We axiomatise the behaviour of the language by formalising algebraically the concept of environment (and stack of environments) to handle recursion. The fork statement is then reduced to non-determinism and recursion. Sharing of data values is formalised by allowing separate processes to have (parts of) their stacks of environments in common.

0 Introduction

Our aim in this report is to provide an algebraic axiomatisation of the behaviour of processes. Before we can proceed, however, we must make more precise what we mean by the word "process". We have found a three point classification of definitions of processes to be useful:

(i) Processes are intended either to terminate after a finite amount of time in the sense that sequential programs do on a finite amount of input or they are intended to run forever on an infinite stream of inputs (as in an operating system). Examples of the former interpretation can be found in R and T , examples of the latter in K&M.

(ii) Processes are intended to synchronise their activities by the sharing of values (through access to common areas of store) or by explicit communication of these values by sending and receiving messages. Examples of the use of shared values can be found in H , R, and D. Examples of communicating processes can be found in T,D,K&M.

(iii) Processes are organised hierarchically (i.e. one process may control the behaviour of another) or anarchically (processes cannot control each other's behaviour (except by creation and the synchronisation mechanisms). Examples of hierarchically organised processes can be found in T,D, and H. Examples of anarchically organised processes can be found in K&M and R.

We intend to study anarchic, terminating processes with synchronisation achieved by sharing of values. Attempts have been made in this direction beginning with the pioneering works of B,K, and Mi(1). The idea of an algebra of processes

was put forward in both B and Mi(1). However, when these works were written, the concept of continuous algebra did not exist. Nor did the concept of a powerdomain and since attempts at defining processes have involved the use of non-determinism, this was also a great drawback. With the advent of continuous algebras (ADJ(2), Mi(2)) and satisfactory definitions of powerdomains (P,S,H,H&P,M(2)), we are now in a position to make more precise our formalisation.

We will use the powerful tools developed in the study of continuous algebras and abstract data types (L,L&M(1)-(2),ADJ(2)) to axiomatise in an algebraic setting the behaviour of programs containing process declarations, assignments, conditionals, while loops, procedure calls and fork statements (our parallelism construct). The reason that this is possible is because we regard a program as an expression mapping assignments of values to variables (the input) to assignments of values to variables (the output). In particular, each program statement defines such a map. To facilitate our development, we assume that expression evaluation has no side effect and that the assignment statement and evaluation of boolean expressions in while loops and conditionals are indivisible.

The key idea in our development is the encoding of the concept of a "local environment" in an algebraic setting. Our ideas stem from the use of structures in L, L&M(1)-(2) to encode values of data structures in the study of abstract data types. Structures (or, as we call them, environments) are assignments of values (of expressions) to variables (or identifiers). The variables (identifiers) are regarded as constants of the algebra whose meanings (i.e. the values which they denote) change over time. As the values change, so does the algebra (since the constant operations denoted by the identifiers have different meanings). We handle process declarations in a similar way.

Process names denote operations of our algebra. The operation they denote is specified by the "body" of the procedure declaration. That these declarations may be (mutually) recursive presents no difficulty.

Related work can be found in H&P, Mi & Mi, and Mi(2). In H&P an attempt is made to define a domain of interpretation for a simple language allowing both parallelism and non-determinism. Solutions to domain equations are used as the main tool. Sharing and communication are both ignored. Mi&Mi (and Mi(2)) use the algebraic approach to study communicating processes (by abstracting their communications "capabilities"). R has also studied an operational definition of a language closely related to ours and the existence of this model was a challenge for the algebraic method we have adopted.

The next section contains some mathematical preliminaries and the outline of our language. The following section contains the axiomatisation of our language of processes. The third section contains an evaluation of the following sample program guided by the axioms of the previous section.

```
P(t) = local z;
       z:= t;
       while z ≠ Λ do
           x:= x+1;
           fork (P(rt(z)));
           z:= lt(z)
       end:
local x;
x:= 0;
P(t).
```

(The program is meant to count the nodes of a binary tree. It does this by having the main process count the nodes of the left subtree of a node and having a separate process count the nodes of the right subtree.)

The two final sections outline the way in which we assign meanings to programs, test for equivalence of programs and provide a summary, respectively.

1.0 Mathematical Preliminaries

We assume the reader is familiar with rudimentary notions of universal algebra, complete partially ordered sets and continuous algebras. We review here briefly some definitions and results to make our use of notation clear. The notation we use is that of ADJ(2) and M(2).

Let S be a set of sorts. A (many-sorted) alphabet (sorted by S) is an indexed family of sets $\Sigma = \{\Sigma_{w,s}\}_{<w,s>\in S^*\times S}$. A ($\Sigma$-)algebra A_Σ (or A if Σ is obvious from context) is an indexed family of sets $A = \{A_s\}_{s\in S}$ and for each $f\in\Sigma_{s_1\ldots s_n,s}$ (any $<s_1\ldots s_n,s>\in S^*\times S$) an assignment of an operation $f_A: A_{s_1}\times\cdots\times A_{s_n} \to A_s$ to f. f(or f_A) is said to be of type $<w,s>$, arity w, sort s, rank $|w|$ (where $|w|$ is the length of w). If $f\in\Sigma_{\lambda,s}$ (for λ the empty string), f is said to be a constant symbol and f_A is said to be a constant of A. We denote by A^w the set $A_{s_1}\times\cdots\times A_{s_n}$ for $w = s_1\ldots s_n$.

A (Σ-)homomorphism between A_Σ and B_Σ is an indexed family of mappings $h = \{h_s: A_s \to B_s\}_{s\in S}$ such that for all $f\in\Sigma_{w,s}$, $<a_1,\ldots,a_n>\in A^w$ we have $h_s(f_A(a_1,\ldots,a_n)) = f_B(h_{s_1}(a_1),\ldots, h_{s_n}(a_n))$. h is said to be injective, surjective, and bijective if and only if each h_s is. We will often omit the subscripts from h_s as they will be clear from the context. h is a Σ-isomorphism if and only if h is a bijective Σ-homomorphism.

A is initial in a class C of Σ-algebras if and only if:

(i) $A_\Sigma\in C$,

and (ii) for each $B_\Sigma \in C$, there is an unique Σ-homomorphism

$$h_B : A_\Sigma \to B_\Sigma .$$

A (Σ-)congruence q on an algebra A_Σ is an indexed family of equivalence relations $q = \{q_s\}_{s \in S}$ such that for all $f \in \Sigma_{w,s}$ and for all $\langle a_1, \ldots a_n \rangle$, $\langle b_1, \ldots, b_n \rangle \in A^w$ such that $a_i q_{s_i} b_i$ for $1 \leq i \leq n$, we have

$$f_A(a_1, \ldots, a_n) q_s f_A(b_1, \ldots, b_n).$$

The quotient of A_Σ by q is denoted A_Σ/q. Given a partially ordered set $\langle D, \leq \rangle$ (or D), we denote by $\perp_D$ (or $\perp$) the least element of D (if it exists) and by $\bigsqcup_{i \in I} d_i$ the least upper bound of the set $\{d_i\}_{i \in I}$ (if it exists). A ω-complete partial order (cpo) D is a partially ordered set which is strict (i.e. has a least element) and ω-complete (i.e. has least upper bounds for all countable chains). A mapping $f: D \to D'$ between partially ordered sets is (ω-)continuous if and only if $f(\bigsqcup_{i \in I} d_i) = \bigsqcup_{i \in I} f(d_i)$ (when the least upper bounds exist.)

A Σ-algebra A_Σ is (ω-)continuous iff each A_p is a cpo and each $f \in \Sigma_{w,p}$ is continuous (as a mapping from A^w to A_p). A homomorphism $n: A_\Sigma \to B_\Sigma$, is (ω-)continuous iff each $h_p : A_p \to B_p$ is continuous.

The class of ω-continuous Σ-algebras together with ω-continuous Σ-homomorphisms, denoted $\underline{CAlg}_\Sigma$, has an initial algebra CT_Σ. For our purposes, we can regard CT_Σ as the algebra of finite and infinite expressions over Σ (where $\perp_s \in \Sigma_{\lambda,s}$ for each s) and the order is the least order compatible with the rule $\perp_s \leq t$ for all $t \in CT_{\Sigma,s}$.

A congruence q on $A_\Sigma \in (\underline{Alg}_\Sigma$ is said to be (ω-)<u>continuous</u> iff for all pairs of chains $\{a_i\},\{b_i\}$ in A_p such that $a_i q_s b_i$ for all $i \in \omega$, we have $(\bigsqcup_{i\in\omega} a_i) q_s (\bigsqcup_{i\in\omega} b_i)$. That is, if two chains are pairwise related, then so are the limits.

<u>Lemma:</u> If A_Σ is continuous and A_Σ/q is continuous, then q is continuous. Note that if A_Σ and q are continuous A_Σ/q may still not be continuous. See L&M for details.

We use the standard definition of equations, satisfaction and least congruence q_ϵ generated by a set of equations ϵ (see L&M(1)). However, when we use continuous algebras, q_ϵ will represent the least continuous congruence satisfying ϵ. Such a congruence always exists as is shown in L&M(1).

Let ϵ be a set of equations. Let $\underline{\mathbb{C}Alg}_{\Sigma,\epsilon}$ be the class of continuous Σ-algebras satisfying ϵ, together with continuous Σ-homomorphisms between them.

<u>Theorem</u> (see H, Le): $\underline{CAlg}_{\Sigma,\epsilon}$ has an initial algebra, denoted by $CT_{\Sigma,\epsilon}$.

Let Σ be such that $+ \in \Sigma_{ss,s}$ for each $s \in S$. Then any algebra in $\underline{CAlg}_{\Sigma,\eta}$ where η is $\{+xy = +yx, ++xyz = +x+yz, +xx = x\}_{s \in S}$ is said to be a non-deterministic domain (ηdd). See H, M(2), H&P, H&M for more details. Objects of $A_\Sigma \in \underline{CAlg}_{\Sigma,\eta}$ can be thought of as elements of a powerdomain with + being the finitary join operation for the elements of the powerdomain. The <u>subset</u> (only suggestive!!) order $\underline{\subseteq}$ on A_s can be defined in terms of + by $x \underline{\subseteq} y$ iff $+xy = y$. Non-deterministic domains will play a large role in our development because we will reduce parallelism to non-determinism.

1.1 The Language

We assume that we have available some basic data type (e.g. the natural numbers) which is presented by D_Δ the initial algebra in the class of (continous) algebras satisfying some set of equations ε_d. The operators of this algebra are named by the alphabet Δ (sorted by S_Δ) and we assume that $\eta \subseteq \varepsilon_d$. That is, algebras satisfying ε_d are non-deterministic domains.

The syntax of our programming language is given as follows:

<program>::=<process declarations>:<prog>

<prog>::=<locals>;<statements>

<process declarations>::=<pdeclaration>;<process delcarations>|λ

<pdeclaration>::=<process name>(<formal parameters>)=<prog>

(We assume we have process names available for every combination of formal paramters we might require. It is also assumed that if P is of type $s_1 \ldots s_n$, s then the formal parameters are $x_1, \ldots, x_n$ where x_i is of sort s_i for $1 \leq i \leq n$.)

<locals>::=local<identifier>;<locals>|λ

(We assume we have available a countable set of identifiers x_0^s, x_1^s, ... for each sort in our underlying domain D_Δ. We denote by X the family $\{x_0^s, x_1^s, \ldots\}_{s \in S}$. In practice we omit the superscript indicating the sort of an identifier as it is usually defined by the context.)

<statements>::=<statement>;<statements>|λ

<statement>::=<identifier>:=<expression>

(An expression is an element of $D_\Delta(X)$, the expressions over Δ and the identifiers X.)

|null

(This statement will do nothing to affect the outcome of a program.)

|if<boolean>then<statements>else<statements>

(<boolean>is a simple proposition in terms of <expression>s . Its value is true or false or undefined as described in the sequel.)

|while<boolean>do<statements>end

|<process call>

(<process call>is used for recursion.)

|fork(<prog>)

(fork is used to start the execution of a process in parallel with the execution of the forking program.)

<process call>::=<process name>(<actual parameters>)

(<actual parameters>are<expression>s whose sorts match those of the corresponding formal parameters.)

We assume we have available a free monoid on <statement>'s whose constant (empty sequence) is nil and whose binary operation is denoted by ";". We call the sort of which statements are elements statement. We also have a sort called prog whose elements are the same as those of statement. We consider the two sorts, however, for their different effects on environments. The map converting a program s into a sequence of statements will be denoted by $\bar{\ }$: prog → statement. Thus $s \in$ prog is mapped onto $\bar{s} \in$ statement.

An example of a program is:

```
P(t) = local z;
       z:=t;
       while z ≠ Λ do
             x:= x+1;                                   (I)
             fork (P(rt(z)));
             z: = ℓt(z)
       end:
local x;
x:=0;                    } ρ
P(T);
```

Our intended interpretation for D_Δ is in this case a combination of the natural numbers and (finite) binary trees (equiped with operations lt and rt to obtain left and right subtrees and the constant Λ to denote the empty tree). This program will count the number of nodes of the "input" tree T by using the process P . The segment ρ is the main program and initialises the value of x (the "counter") to zero. P is then called and increments x by one if the actual parameter is not Λ . It then forks a copy of P to count the nodes of the right subtree of the actual parameter and continues counting the nodes of the left subtree. Note that the number of activated processes depends only on T and cannot be predicted ahead of time independently of the input.

Our intention is to formalise this intuitive behaviour algebraically. This means that we will want programs to be finite expressions in some algebra and meanings of programs to be (possibly) infinite expressions obtained by taking the least fixed point of the functional defined by the program. Some of the obstacles in this formalisation are as follows:

(i) What is the meaning of procedure declarations?

(ii) How do we incorporate the many possible parallel invocations of a given "body of code" into our system? (This amounts to defining explicitly some way of handling local variables. The problem is somewhat similar to that of block structured languages.)

1.2 Environments

The beginnings of our solution to the "problem" of local variables is to use the concept of environment. However, we do not mean the usual "operational" view of environments which includes such concepts as "program counter". On the other hand, the inclusion of assignment as a basic operation forces us to adopt a view which, even if handled abstractly, "smells" of the operational viewpoint.

For us, an environment is, intuitively, a mapping δ from identifiers (of sort s) to values in D_Δ (of sort s). An assignment statement x:=t is then used to update the environment δ to a new one δ' where the value of x is the value of t in δ. This would be sufficient if we did not have process names (and thus the problem of scope of identifiers). Because of the presence of process names, we must have stacks of enviornments. In fact we will use sequences rather than stacks as we will need to concatenate "stacks". We give the specification of sequence of $\mathcal{D}$ (for $\mathcal{D}$ unspecified) in the style of abstract data type specifications. This is a parameterised data type and we will later substitute environments for $\mathcal{D}$. (See ADJ(1), G).

In order to define sequences (and other types we will use) we need the type bool defined as follows:

bool:: Sorts ≡ bool

Operations ≡

true, false, &: → bool

¬: bool → bool

∨: bool × bool → bool

Cond: bool × bool × bool → bool

Variables ≡ b, b', b_1, b_2: bool

Axioms ≡

$$\neg(\perp) = \perp$$

$$\perp \vee b = b \vee \perp = \perp$$

(ϵ_b) $\quad$ $\underset{\sim\sim\sim\sim}{\text{true}} \vee b = \underset{\sim\sim\sim\sim}{\text{true}}$ if $b \neq \perp$

$$\underset{\sim\sim\sim\sim\sim}{\text{false}} \vee b = b$$

$$b_1 \vee b_2 = b_2 \vee b_1$$

$$\text{cond}(\underset{\sim\sim\sim\sim}{\text{true}}, b_1, b_2) = b_1$$

$$\text{cond}(\underset{\sim\sim\sim\sim\sim}{\text{false}}, b_1, b_2) = b_2$$

$$\text{cond}(\perp, b_1, b_2) = \perp$$

We need for bool, the operation + with the usual axioms together with:

$$\neg + b_1\ b_2 = + \neg b_1 \neg b_2$$

$$(+b_1\ b_2) \vee b = +(b_1 \vee b,\ b_2 \vee b)$$

$$\text{cond}(+b_1\ b_2, b, b') = +(\text{cond}(b_1, b, b'), \text{cond}(b_2, b, b'))\ .$$

We will also assume that in any algebra that we define, for every sort s we have

$$\text{cond}: \underline{\text{bool}} \times s \times s \to s$$

with axioms analogous to those above. Moreover, as in the case of sequences (and according to the practice for defining abstract data types), we implicitly use the sort bool in the definition of the algebras in which we are interested.

Now for the definition of sequence.

seq of $\mathcal{D}$ :: Sorts $\equiv$ seq, $\mathcal{D}$ (or those of $\mathcal{D}$)

Operations $\equiv$

$\perp, \Lambda$: $\to$ seq

hd, $t\ell$: seq $\to$ seq

$\cdot$: seq $\times$ seq $\to$ seq (Used as an infix operator.)

We also assume an operation to convert each value in $\mathcal{D}$ into a sequence "containing" only that value. This operation is invoked implicitly and so we use $t \in \mathcal{D}$ both to denote the value in $\mathcal{D}$ and the sequence containing only t .

Variables $\equiv$ st_1, st_2, st_3: seq; t:$\mathcal{D}$

Axioms $\equiv$ $hd(\perp) = t\ (\perp) = \perp$

$hd(t) = t$

$hd(\Lambda) = \perp$

$t\ell(t) = \Lambda$

(ϵ_{seq}) $t\ell(\Lambda) = \Lambda$

$\perp \cdot st_1 = st_1 \cdot \perp = \perp$

$\Lambda \cdot st_1 = st_1 \cdot \Lambda = st_1$

$hd(st_1 \cdot st_2) = cond(st_1 = \Lambda, hd(st_2), hd(st_1))$

$t\ell(st_1 \cdot st_2) = t\ell(st_1) \cdot st_2$

$hd(t \cdot st_1) = cond(st_1 = \perp, \perp, t)$

Again we will need + together with the usual axioms and the following:

$hd(+st_1\ st_2) = +(hd(st_1), hd(st_2))$

$t\ell(+st_1\ st_2) = +(t\ell(st_1), t\ell(st_2))$

$(+\ st_1\ st_2) \cdot st_3 = +(st_1 \cdot st_3, st_2 \cdot st_3)$

$(st_1 \cdot (+st_2\ st_3)) = +(st_1 \cdot st_2, st_1 \cdot st_3)$

We can now give the definition of environments (env).

env:: Sorts $\equiv$ env, S_Δ , id_s for each $s \in S_\Delta$

Operations $\equiv$

All operations of D_Δ .

ϕ : $\rightarrow$ env

$\perp$: $\rightarrow$ env

x_i^s : $\rightarrow$ env $\rightarrow$ s for each $i \in \omega$, $s \in S_\Delta$.

x_i^s : $\rightarrow$ id_s

update: $id_s \times s \times env \rightarrow env$ for each $s \in S_\Delta$.

Variables $\equiv$ δ, δ_1, δ_2: env; t : $s \in S_\Delta$; x, y: id_s

Axioms $\equiv$

All axioms used to define D_Δ (i.e. ϵ_d).

$$x(\phi) = \perp_s \quad \text{for each} \quad s \in S,\ i \in \omega$$

$$(\epsilon_{env}) \quad x(\perp_{\underline{env}}) = \perp_s \quad \text{for each} \quad s \in S,\ i \in \omega$$

$$x(\text{update}(y, t, \delta)) = \text{cond}\ (x == y, t(\delta), x(\delta))$$

Note that == is syntactic equality. That is == returns true (false) iff its arguments are exactly the same (unequal) strings. We have also extended the "evaluation" map on environments from identifiers to expressions in the obvious way.

$$\text{update}(x, t, \perp_{\underline{env}}) = \perp_{\underline{env}}$$

This axiom makes sure that the undefined environment remains undefined under updates. Although ϕ and $\perp_{\underline{env}}$ give the same values ($\perp$) for identifiers, the former is updatable--it is the initial environment--while the latter is not.

We will also need to add to env , the operation + on the sort env with the usual axioms and the following:

$$x(+\ \delta_1\ \delta_2) = +\ x(\delta_1)\ x(\delta_2)$$

$$\text{update}(x, t, +\delta_1\ \delta_2) = +(\text{update}(x, t, \delta_1), \text{update}(x, t, \delta_2)) .$$

2. Processes

In this section we will axiomatise the way in which program statements change environments. We begin in subsection 2.1 with program statements which do not call for new environments to be created (recursion and forking). In subsection 2.2 we will deal with the remaining problems.

2.1 Simple Program Statements

local, null, if then else, while and assignments are statements which map a sequence of environments to some other sequence of environments. So the syntax of the operations is as follows (with seq ≡ seq of env):

$$\text{local} : \text{id}_s \times \text{seq} \to \text{seq}$$

$$\text{null} : \text{seq} \to \text{seq}$$

$$\text{if then else} : \text{bool} \times \text{statement} \times \text{statement} \times \text{seq} \to \text{seq}$$

$$\text{while do end} : \text{bool} \times \text{statement} \times \text{seq} \to \text{seq}$$

$$:= : \text{id}_s \times \text{s} \times \text{seq} \to \text{seq}$$

We will also need some boolean valued operations in order to define assignment. These functions decide whether or not an identifier is local in an environment or somewhere in a sequence of environments.

$$\text{local} : \text{id}_s \times \text{env} \to \text{bool}$$

$$\text{local} : \text{id}_s \times \text{seq} \to \text{bool}$$

Now for the axiomatisation of our operations. Note, firstly, that we have not included ";" (semi-colon) as an operation. This is because it represents functional composition and is thus already within the algebraic framework. Secondly, we do not state any axioms to indicate what exactly

local statements do to environments. The reason for this will become apparent when we look at local (and slocal).

Variables ≡ S_1, S_2 : statements; S : prog; $\sigma_1, \sigma_2, \sigma_3$: seq; t,t' : D;
and other variables as before.

Axioms ≡ $S(\sigma_1) = \bar{S}(\phi \cdot \sigma_1)$

(Thus, to execute a program S, we execute the corresponding sequence of statements on a new local environment.)

$$\underline{null}(\sigma) = \sigma$$

(null is the identity on environments.)

$$\underline{if}\ b\ \underline{then}\ S_1\ \underline{else}\ S_2(\sigma) = cond(b(\sigma) = \underline{true},\ S_1(\sigma), cond(b(\sigma) = \underline{false},\ S_2(\sigma), \perp))$$

(If b is true in σ then do S_1 on σ and if $b(\sigma)$ is false then do S_2 on σ, otherwise return the undefined sequence of environments.)

$$\underline{while}\ b\ \underline{do}\ S_1\ \underline{end}(\sigma) = \underline{if}\ b\ \underline{then}\ (S_1;\ \underline{while}\ b\ \underline{do}\ S_1\ \underline{end})\ \underline{else}\ \underline{null}\ (\sigma)$$

$$\underline{local}\ x\ (\delta \cdot \sigma_1) = \underline{local}\ x\ (\delta) \cdot \sigma_1$$

$$local\ (x)(\phi) = \underline{false}$$

$$local\ (x)(\perp_{env}) = \perp$$

$$local\ (x)(\underline{local}\ y(\delta)) = cond(x==y,\ \underline{true},\ local(x)(\delta)).$$

Let s_1 be either assignment, null, if then else, while, fork or recursion.

$$local(x)(s_1(\delta)) = local(x)(\delta)$$

$$local(x)(+\delta_1\delta_2) = +(local(x)(\delta_1), local(x)(\delta_2))$$

(The reason why we did not state any axiom as to the behaviour of local with

respect to environments was because we wanted to check its effect on environments syntactically. local ignores non-local statements and examines whether the environment has had a local statement applied to it and, if so, compares its argument with that of local.)

$$\text{slocal}(x)(\perp_{\underline{\text{seq}}}) = \perp$$

$$\text{slocal}(x)(\Lambda) = \underline{\text{false}}$$

$$\text{slocal}(x)(\delta \cdot \sigma_1) = \text{cond}(\text{local}(x)(\delta_1), \underline{\text{true}}, \text{slocal}(x)(\sigma_1))$$

$$\text{slocal}(x)(+\sigma_1\sigma_2) = +(\text{slocal}(x)(\sigma_1), \text{slocal}(x)(\sigma_2)).$$

(The behaviour of slocal is similar to that of local.)

$$x := t(\sigma) = \text{cond}(\text{local}(x)(\text{hd}(\sigma)), \text{update}(x,t,\text{hd}(\sigma)) \cdot t\ell(\sigma), \text{hd}(\sigma) \cdot x := t(t\ell(\sigma)))$$

$$x := t(\Lambda) = \perp_{\text{seq}}$$

(An assignment to be performed on the empty sequence of environments is undefined.)

Finally, we have

$$\text{update}(x,t,\underline{\text{local}}\ y(\delta)) = \underline{\text{local}}\ y(\text{update}(x,t,\delta)).$$

(The meaning of this axiom is clear.)

We now wish to take into account the ramifications of recursion and forking. This is the purpose of the next section.

2.2 Recursion and Forking

We will begin by treating recursion. We will then reduce forking to recursion and non-determinism using a semantic parallelism construct (named " $\|$ ").

Suppose that we have a process definition

$$P(x_1,\ldots,x_n) = S$$

where P is of type <w,s> and S is of sort s. (In fact S is a derived operator of type <w,s>. See ADJ(2), M(2), M&H,D.) We will add the following axiom for each process declaration of the above form:

(r)
$$P(x_1,\ldots,x_n)\,[t_1(\sigma_1),\ldots,t_n(\sigma_1)](\sigma_1) = S[t_1(\sigma_1),\ldots,t_n(\sigma_1)](\sigma_1)$$

(Here $t_1,\ldots,t_n$ are variables ranging over appropriate sorts in D_Δ and S [...] means substitution of expressions for variables in S. This can be axiomatised quite simply and we will not go through the details here. Note also that we have extended the evaluation of expressions from evaluation in an environment to evaluation in a stack or sequence of environments. This extension is quite simple to axiomatise and will not be presented here.) This expresses the usual interpretation of recursion in which a recursive call causes the execution of the body of the process declaration with a new empty environment for local values stacked onto the environments which exist at the time of call. Note that we are using call by value as each t_i is evaluated at the time of call. What might seem strange is the inclusion of such axioms in the specification of the algebra. More about this later, but it seems impossible to handle named procedures without such axioms. The symbolic meaning of the program should not and will not refer to these

process names.

In order to axiomatise fork, we need a semantic parallelism operator.

$$\| : \underline{seq} \times \underline{seq} \to \underline{seq}$$

First of all we have

(f) $$\underline{fork}(S);\ S_2(\sigma_1) = S(\sigma_1) \parallel S_2(\sigma_1).$$

In particular, using (r) and (f) we have the derived axiom

(rf) $$\underline{fork}(P)(x_1,\ldots,x_n)[t_1(\sigma_1),\ldots,t_n(\sigma_1)];\ S_2(\sigma_1) = S[t_1(\sigma_1),\ldots,t_n(\sigma_1)](\sigma_1) \parallel S_2(\sigma_1).$$

Now we have to worry about the axiomatisation of $\|$.

(pr1) $$\sigma_1 \parallel \sigma_2 = \sigma_2 \parallel \sigma_1$$
$$\sigma_1 \parallel (\sigma_2 \| \sigma_3) = (\sigma_1 \| \sigma_2) \parallel \sigma_3$$

Also, we need

$$(\sigma_1 \| \sigma_2) \cdot \sigma_3 = \sigma_1 \cdot \sigma_3 \parallel \sigma_2 \cdot \sigma_3$$

to handle common "global" environments.

Now we axiomatise the behaviour of programs "running in parallel".

$$(x := t;\ S_1(\sigma_1) \parallel y := t';\ S_2(\sigma_2)) \cdot \sigma_3$$

(pra) $$= +\ (\mathrm{cond}(\mathrm{slocal}(x)(\sigma_1), (S_1(\mathrm{update}(x,t,\sigma_1)) \parallel y := t';\ S_2(\sigma_2)) \cdot \sigma_3, (S_1(\sigma_1) \parallel y := t';\ S_2(\sigma_2)) \cdot \mathrm{update}(x,t,\sigma_3)(\sigma_3))$$

$$\begin{aligned}
&\mathrm{cond}(\mathrm{slocal}(y)(\sigma_2)\\
&\quad ,(x := t;\ S_1(\sigma_1) \parallel S_2(\mathrm{update}(y,t',\sigma_2)(\sigma_2))) \cdot \sigma_3\\
&\quad ,(x := t;\ S_1(\sigma_1) \parallel S_2(\sigma_2)) \cdot \mathrm{update}(y,t',\sigma_3)\\
&\quad)\\
&).
\end{aligned}$$

update was previously defined on an environment and not a sequence of environments. However, this simple update can easily be extended (using cond and local) to an operation of updating on sequences which we also call update. Its full axiomatisation (quite simple) is left to the reader.

Thus if two programs running in parallel are about to execute assignment statements, then one of the assignments must be done first and the other may be done on a possibly changed (stack of) environment(s).

(pr1)
$$\begin{aligned}
&\underline{\mathrm{local}}\ x;\ S_1(\sigma_1) \parallel \underline{\mathrm{local}}\ y;\ S_2(\sigma_2)\\
&\quad = S_1(\underline{\mathrm{local}}\ x(\mathrm{hd}(\sigma_1)) \cdot t\ell(\sigma_1)) \parallel S_2(\underline{\mathrm{local}}\ y(\mathrm{hd}(\sigma_2))\\
&\qquad\qquad \cdot t\ell(\sigma_2))
\end{aligned}$$

Thus the local statements can be applied truly in parallel.

(pral)
$$\begin{aligned}
&(x := t;\ S_1(\sigma_1) \parallel \underline{\mathrm{local}}\ y;\ S_2(\sigma_2)) \cdot \sigma_3\\
&\quad = +\ (\mathrm{cond}(\mathrm{slocal}(x)(\sigma_1)\\
&\qquad\qquad ,(S_1(\mathrm{update}(x,t,\sigma_1))) \parallel \underline{\mathrm{local}}\ y;\ S_2(\sigma_2)) \cdot \sigma_3\\
&\qquad\qquad ,(S_1(\sigma_1) \parallel \underline{\mathrm{local}}\ y;\ S_2(\sigma_2)) \cdot \mathrm{update}(x,t,\sigma_3)\\
&\qquad\qquad)\\
&\qquad ,\\
&\qquad (x := t;\ S_1(\sigma_1) \parallel S_2(\underline{\mathrm{local}}\ y(\mathrm{hd}(\sigma_2)) \cdot t\ell(\sigma_2))) \cdot \sigma_3\\
&\qquad).
\end{aligned}$$

Thus parallel executions of local and assignment statements must be done one at a time.

There is clearly an analogous axiom for the symmetric case (with respect to assignment and local).

After our program has finished running, we will be left with expressions in terms of +, $\|$ and seq of environments. We want to throw away the local environments we created and just keep the one with which we started. We can do this with

(en) $$S_1(\sigma_1 \cdot \sigma_3) \| S_2(\sigma_2 \cdot \sigma_3)$$
$$= \text{cond}(S_1 = \underline{\text{nil}} \wedge S_2 = \underline{\text{nil}} \wedge \sigma_3 \neq \Lambda, \sigma_3, S_1(\sigma_1 \cdot \sigma_3)$$
$$\| S_2(\sigma_2 \cdot \sigma_3))$$

Thus if we begin our programs with the sequence consisting of one element - namely the empty environment ϕ in env - we will end up with a sequence of environments consisting of one element - namely the one which records the values of the identifiers local to the main program.

We have now finished our axiomatisation of processes. Before we begin a discussion of this axiomatisation in section 4, we present a brief execution of the program (I) of subsection 1.1 on a sample value of T (by using the axioms as rewriting rules on expressions).

3. A Sample Program

Assume D_Δ is the domain of natural numbers and (finite) binary trees with operations &t, rt and Λ having the meaning ascribed to them in 1.1. We will use symbolic representations such as $_*{}^*_*$ for binary trees. We will identify segments of code as follows

```
P(t) =
        local z;                                  } q0
        z := t;                                   } q1
        while z ≠ Λ do
                x := x+1;                 } q'20
                fork (P(rt(z)));          } q'21   } q'2   } q2   } q
                z := &t(z)                } q'22
        end:
local x;
x := 0;                                           } p'    } p
P(_*_*)
```

We will use T for $_*{}^*_*$. We use the notation [[x=t,y=t']] to indicate values of identifiers in local environments.

$P(\Lambda)$

$= \underline{\text{local}}\ x;\ p'(\phi)$

$= x := 0;\ P(T)(\phi_1)$ ($\underline{\text{local}}\ x(\phi) \equiv \phi_1$)

$= P(T)(\text{cond}(\text{local}(x)(\text{hd}(\phi_1)), \text{update}(x,0,\text{hd}(\phi_1))\cdot t\ell(\phi_1)$

$, \text{hd}(\phi_1)\cdot x := 0\ (t\ell(\phi_1))))$

$= P(T)(\phi_2)$ ($\underline{\text{local}}\ x\ [[x=0]] \equiv \phi_2$)

- since $\text{hd}(\phi_1) = \phi_1$ and $\text{local}(x)(\phi_1) = \underline{\text{true}}$.

$= q[T](\phi\cdot\phi_2)$

$= \underline{\text{local}}\ z; q_1; q_2(\phi\cdot\phi_2)$

$= z := T; q_2(\phi_3)$ ($\underline{\text{local}}\ z\ (\phi)\ \cdot\ \phi_2 \equiv \phi_3$)

$= q_2(\text{cond}(\text{local}(z)(\text{hd}(\phi_3)), \text{update}(z,T,\text{hd}(\phi_3))\cdot t\ell(\phi_3)$

$, \text{hd}(\phi_3)\cdot z := T(t\ell(\phi_3))))$

$= q_2(\phi_4)$ ($\underline{\text{local}}\ z\ [[z=T]]\cdot\phi_2 \equiv \phi_4$)

- since $\text{local}(z)(\text{hd}(\phi_3)) = \underline{\text{true}}$.

$= \underline{\text{if}}\ z \neq \Lambda\ \underline{\text{then}}\ q_2'; q_2\ \underline{\text{else}}\ \underline{\text{null}}\ (\phi_4)$

$= \text{cond}(z\neq\Lambda)(\phi_4) = \underline{\text{true}},\ q_2'; q_2(\phi_4),$

$\text{cond}((z\neq\Lambda)(\phi_4) = \underline{\text{false}},\ \underline{\text{null}}(\phi_4), \perp))$

$= q_2'; q_2(\phi_4)$

- since $(z\neq\Lambda)(\phi_4) = \underline{\text{true}}$.

$= x := x+1; \bar{q}(\phi_4)$ ($q_{21}'; q_{22}'; q_2 \equiv \bar{q}$)

$= \bar{q}(\text{cond}(\text{local}(x)(\text{hd}(\phi_4)), \text{update}(x,x+1,\text{hd}(\phi_4))\cdot t\ell(\phi_4)$

$, \text{hd}(\phi_4)\cdot x := x+1(t\ell(\phi_4))))$

$= \bar{q}(\text{hd}(\phi_4)\cdot\text{cond}(\text{local}(x)(\phi_2), \text{update}(x,x+1,\text{hd}(\phi_2))\cdot t\ell(\phi_2)$

$, \text{hd}(\phi_2)\cdot x := x+1(t\ell(\phi_2))))$

- since $t\ell(\phi_4) = \phi_2$ and $\text{local}(x)(\text{hd}(\phi_4)) = \underline{\text{false}}$.

$= \bar{q}(\phi_5)$ ($\underline{\text{local}}$ z [[z=T]]$\cdot\underline{\text{local}}$ x[[x=1]] $\equiv \phi_5$)

- since local(x)(hd(ϕ_2)) = $\underline{\text{true}}$ and x was 0.

$= q'_{21};q'_{22};q_2(\phi_5)$

$= (q[\text{rt}(T)](\phi)]]q'_{22};q_2(\Lambda))\cdot\phi_5$

$= +\ (q'_{22};q_2(\Lambda)||q_1;q(\underline{\text{local}}\ z(\text{hd}(\phi))\cdot t\ell(\phi))\cdot\phi_5$

,cond(slocal(z)(hd(Λ))

,q_2(update(z,ℓt(T),hd(Λ))$\cdot$tℓ(Λ))||$\underline{\text{local}}$ z;q_1;q(ϕ))$\cdot\phi_5$

,$q_2(\Lambda)||\underline{\text{local}}$ z;q_1;q(ϕ))$\cdot$update(z,ℓt(z),ϕ_5))

)

$= (q_2(\Lambda)||q_1;q[*](\underline{\text{local}}\ z(\phi)))\cdot\phi_6$ ($\underline{\text{local}}$ z[[z=*]]local x[[x=1]] $\equiv \phi_6$)

- since both arguments of + will evaluate to this expression (and so we use + $\sigma_1\sigma_1 = \sigma_1$) and since ϕ_6 is obtained by updating the z (= $_*{}^*{}_*$) in ϕ_5 to ℓt(z) = *.

$= \underline{\text{if}}\ z \neq \Lambda\ \underline{\text{then}}\ q'_2;q_2\ \underline{\text{else}}\ \underline{\text{null}}(\phi_6)||q_1;q_2[*](\underline{\text{local}}\ z(\phi)\cdot\phi_6)$

$= (q'_2;q_2(\Lambda)||q_1;q_2(\underline{\text{local}}\ z(\phi))\cdot\phi_6$

- since $(z\neq\Lambda)(\phi_6)$ = $\underline{\text{true}}$.

At this point, we want to "execute" two assignments in parallel. Rather than indicate in our calculation the possible non-deterministic choice necessary at this point (using axiom (pra)), we will "make" a choice by using one of the rewrite rules

$+xy \rightarrow x$

$+xy \rightarrow y$.

Thus the above sequence of equalities continues with

$=$ cond(slocal(z)($\underline{\text{local}}$ z(ϕ))

,(q_2[*](update(z,*,$\underline{\text{local}}$ z(ϕ))||q'_2;$q_2(\Lambda)$)$\cdot\phi_6$

,(q_2[*]($\underline{\text{local}}$ z(ϕ))||q'_2;$q_2(\Lambda)$)$\cdot$update(z,*,ϕ_6))

$= (q_2[*](\phi_8)]]q_2';q_2(\Lambda))\cdot\phi_6$ (local z [[z=*]] $\equiv \phi_8$)

- since slocal(z)(local z(ϕ)) = true.

$= (\underline{\text{if}}\ z \neq \Lambda\ \underline{\text{then}}\ q_2';q_2[*]\underline{\text{else}}\ \underline{\text{null}}\ (\phi_8)]]q_2';q_2(\Lambda))\cdot\phi_6$

- since $(z=\Lambda)(\phi_8)$ = true.

$= (q_2';q_2[*](\phi_8)]]q_2';q_2(\Lambda))\cdot\phi_6$

$= (x := x+1,\bar{q}[*](\phi_8)]]x := x+1;\bar{q}(\Lambda))\cdot\phi_6$

$= (\bar{q}[*](\phi_8)]]x := x+1;\bar{q}(\Lambda))\cdot\phi_6'$ ($x := x+1(\phi_6)$ = local z[[z=*]]

·local x[[x=2]] $\equiv \phi_6'$)

$= (\bar{q}[*](\phi_8)]]\bar{q}(\Lambda))\cdot\phi_6''$ ($x := x+1(\phi_6')$ = local z [[z=*]]

·local x[[x=3]] $\equiv \phi_6''$)

$= (((q_{22}';q_2[*](\Lambda)]]q[\Lambda](\phi))\cdot\phi_8)]](q_{22}';q_2(\Lambda)]]q[\Lambda](\phi)))\cdot\phi_6''$

$= (((q_{22}';q_2[*](\Lambda)]]\bar{\phi})\cdot\phi_8)]](q_{22}';q_2(\Lambda)]]\bar{\phi}))\cdot\phi_6''$ (local z [[z=Λ]] $\equiv \bar{\phi}$)

- $\bar{\phi}$ is obtained by executing only $q_0;q_1$ on ϕ since in both cases $z \neq \Lambda$ will fail (and thus execution of those processes will terminate).

$= ((\Lambda]]\bar{\phi})\cdot\phi_9]](q_{22}';q_2(\Lambda)]]\bar{\phi}))\cdot\phi_6''$ (local z [[z=Λ]] $\equiv \phi_9$)

- since q_{22}' on $\Lambda\cdot\phi_8$ will set $z=\Lambda$ and so the condition $z \neq \Lambda$ in q_2 will fail.

$= ((\Lambda]]\bar{\phi})\cdot\phi_9]](\Lambda]]\bar{\phi}))\cdot\phi_{10}$ ($z := \ell t(z)(\phi_6'')$ = local z [[z=Λ]]

·local x [[x=3]] $\equiv \phi_{10}$)

- reasoning similar to the above.

$= (\phi_9]](\Lambda]]\bar{\phi}))\cdot\phi_{10}$

$= (\phi_9\cdot\phi_{10}'](\Lambda]]\bar{\phi})\cdot\phi_{10}')\cdot\phi_{10}''$ (local z [[z=Λ]] $\equiv \phi_{10}'$,

local x [[x=3]] $\equiv \phi_{10}''$)

$= (\phi_9\cdot\phi_{10}'](\phi_{10}')\cdot\phi_{10}''$

$= \phi_{10}''$

Thus the number of nodes in T is

$$x(\underline{\text{local}}\ x\ [[x=3]]) = 3.$$

At least, this is the result of one possible execution of the program.

4. The Denotation of Programs

We would now like to discuss(somewhat informally) the "meaning" of programs using the fork construct. We have defined an algebra over the alphabet Φ sorted by $S_\Phi = \{$bool, S_Δ , seq of env, env, id_s for each $s \in S_\Delta$, statement, prog$\}$ Φ includes all the operations we mentioned in the axiomatisation together with the family of sets of process names $\{P_i^{w,s} \mid i \in \bar{w}\}_{\langle w,s\rangle \in S_\Delta^* \times S_\Delta}$.

Suppose we denote the set of all the equations we have used, less any equations involving process invocation (such as (r) and (rf)), by E. Then we are guaranteed of the existence of a continuous algebra $CT_{\Phi,E}$ which is initial in the class of all algebras satisfying E . We denote this class by $CAlg_{\Phi,E}$. This algebra allows us to discuss equivalence between programs **only if we ignore the meaning of all process names. That is, a process $P_i^{w,s}$ is equivalent only to itself as we have no axioms telling us the relationship** between $P_i^{w,s}$ and any other $P_j^{w,s}$.

So if our programs do not include any process declarations, then we can use this framework to discuss program equivalence. Twopprograms p_1 and p_2 (with no process declarations) are equivalent if and only if $[p_1] = [p_2]$ where $[...]$ denotes congruence class of ... . These and related problems are discussed in C&N,H.

In case we have process delcarations

$$(P) \qquad \begin{aligned} P_1(x_1, \ldots x_{n_1}) &= t_1 \\ &\vdots \\ P_m(x_1, \ldots, x_{n_m}) &= t_m \end{aligned}$$

in our program, then we add an axiom of the form (r) (and one of the form (rf) if desired, though this is redundant) to E for each process delcaration. Call this set of new equations E_p . Since each $P_i \in \Phi$, we think of it as an operation **and the equations E_p relate the behaviour of the operations** named by

P_i to that of the other elements of Φ (including some other process names if the P_i are recursive). We are now guaranteed of an initial algebra CT_{Φ,E,E_p} in the class of all algebras satisfying $E \cup E_p$. We denote this class by $\underline{CAlg}_{\Phi,E,E_p}$. Programs p_1 and p_2 using processes in P are equivalent if and only if $[p_1] = [p_2]$ in CT_{Φ,E,E_p}.

Now suppose we have two sets of process declarations p_1 and p_2 giving rise to sets of equations E_{p_1} and E_{p_2}. We can say that the declarations define equivalent sets of processes if and only if any program p behaves in the same way using p_1 as it does using p_2. A more precise way of stating this is that p_1 and p_2 are equivalent if and only if $CT_{\Phi,E,E_{p_1}}$ is isomorphic to $CT_{\Phi,E,E_{p_2}}$. Note that we have taken a rather unusual approach to define the meaning of a program in that we do not allow the arbitrary interpretation of any symbols in a program. The meaning of any symbol in our alphabet is specified totally by the set of equations $E \cup E_p$. This approach is borrowed from the study of abstract data types in which a data type is specified uniquely (up to isomorphism) by a set of equations. We could relax this condition somewhat by allowing D_Δ (the algebra of data objects) to come from the class of continuous Δ-algebras satisfying some equations rather than being the initial algebra in this class. The definitions of equality could then be extended in the obvious way. In such cases, the class of objects CT_{Φ,E,E_p} would become a class of semi-Herbrand interpretations in the sense of H.

5. Conclusions

We have presented an algebraic axiomatisation of a simple language containing a parallelism construct. The axiomatisation was based upon an appropriate "encoding" of the concept of local environment. The motivation for this treatment was obtained from the use of structures in L and L&M(2). We make no claims concerning the "completeness" or the "uniqueness" of the axiomatisation as the axiomatisation was meant to be purely illustrative.

In Section 3, we have used the standard interpretation of equations as rewriting rules (see V, H(2), and H&M(2)) to indicate how an operational definition of the meaning of a program may be used to "evaluate" the program. We do not claim any equivalence between such an operational semantics and the semantics we discussed above, although the connection could probably be formalised.

The semantics discussed in Section 4 can be shown to be fully abstract (in the sense of P(2)) using the congruence properties of the algebras CT_{Φ,E,E_p}. Moreover, although we have not above taken the problem of inputs and outputs into account, we could do this by applying a program to an initial (sequence) of enviornments δ_I defined as follows: $\delta_I = \phi' \cdot \Lambda$ where ϕ' is $\underline{\text{local}}_{x_1}(\ldots(\underline{\text{local}}_{x_n}(\underline{\text{local}}_{y_1}(\ldots(\underline{\text{local}}_{y_m}([[x_1 = t'_1, \ldots, x_n = t'_n, \ldots, x_n = t'_n]])\ldots)$ where the x_i are input variables, the y_i are output variables and we have used the notation of section 3 to indicate values of variables in ϕ'. The effect of a program S on δ_I would then be to "run"the statements $\overline{S}$ on $\phi \cdot \delta = \phi \cdot \phi'$. Any updates to the output variables would then be done on ϕ' and these values could be recovered when the program terminated.

The obvious extension to this work would be to try to axiomatise in a similar way the behaviour of communicating processes. This has already been tackled by Mi&Mi in a somewhat different setting. We would also like to define

classes of domains over which processes could be interpreted in the same sense that continuous algebras provide domains of interpretations for ordinary sequential programs and powerdomains for sequential programs with a choice construct (H&P, H). This seems to be a difficult task and will not be accomplished easily.

Acknowledgements

I would like to thank M.R. Levy and many others at Waterloo for their helpful suggestions.

Bibliography

ADJ(1) : J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright--An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, "Current Trends in Programming Methodology, Volume IV", Ed. R.T. Yeh, Prentice Hall, 1978.

ADJ(2) : J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright--Initial Algebra Semantics and Continuous Algebras, JACM, Vol. 24, No. 1, 1977.

B : H. Bekic--Towards a Mathematical Theory of Processes, Tech. Rep. TR25.125, IBM Laboratory, Vienna, 1971.

C : P.M. Cohn--"Universal Algebra", Harper and Row, 1965.

C&N : B. Courcelle, M. Nivat--Algebraic Families of Interpretations, Proceedings of the Symposium on the Foundations of Computer Science, Houston, 1976.

D(1) : W. Damm--Higher Type Program Schemes and their Tree Languages, 3rd GI Conference on Theoretical Computer Science, Lecture Notes in Computer Science #48, Springer-Verlag, 1977.

D(2) : W. Damm--Languages Defined by Higher Type Program Schemes, 4th Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science #52, Springer-Verlag, 1977.

D : F. Baskett, J.H. Howard, J.T. Montague--Task Communication in DEMOS, Proceedings of the 6th ACM Symposium on Operating Systems Principles, 1977.

H : M.C.B. Hennessy--Initial Algebras and Herbrand Interpretations, Research Report, Universidade Federal de Pernambuco, 1978.

H&M : M.C.B. Hennessy, T.S.E. Maibaum--In preparation.

H&P : M.C.B. Hennessy, G. Plotkin--In preparation.

H : E. Cohen, D. Jefferson--Protection in the HYDRA Operating System
and
W. Wulf, R. Levin, C. Pierson--Overview of the HYDRA Operating System Development
both in
Proceedings of the 5th ACM Symposium on Operating Systems Principles, 1975.

K : G. Kahn--The Semantics of a Simple Language for Parallel Programming, Proceedings of IFIP Congress 74, North-Holland, 1974.

K&M : G. Kahn, D. MacQueen--Co-routines and Networks of Parallel Processes, Research Report 202, IRIA, Paris, 1976.

Le : D. Lehmann--On the Algebra of Order,Proceedings of the 19th Symposium on the Foundations of Computer Science, 1978.

L : M.R. Levy--Verification of Programs with Data Referencing, Proceedings of the 3^{e} Colloque International sur la Programmation, Dunod, 1978.

L&M(1) : M.R. Levy, T.S.E. Maibaum--Continuous Data Types, In preparation.

L&M(2) : M.R. Levy, T.S.E. Maibaum--Data Types with Sharing and Circularity, In preparation.

M(1) : T.S.E. Maibaum--Generalized Formal Language Theory, JCSS, Vol. 8, No. 3, 1974.

M(2) : T.S.E. Maibaum--The Semantics of a Simple Non-deterministic Language, Proceedings of the 3^{e} Colloque International sur la Programmation, Dunod, 1978.

Mi(1) : R. Milner--Processes, a Mathematical Model of Computing Agents, Proceedings of Logic Colloquium, Bristol, North-Holland, 1973.

Mi(2) : R. Milner--Flowgraphs and Flow Algebras, Tech. Rep. CSR-5-77, University of Edinburgh, 1977.

Mi&Mi : G. Milne, R. Milner--Concurrent Processes and Their Syntax, Tech. Rep. CSR-2-77, University of Edinburgh, 1977.

P(1) : G. Plotkin--A Powerdomain Construction, SIAM Journal on Computing, Vol. 5, No. 3, 1976.

P(2) : G. Plotkin--Call by name, call by value and the λ-calculus, Res. Memo. SAI-RM-6, University of Edinburgh, 1973.

R : N. Redding--In preparation.

S : D. Scott--The Lattice of Flow Diagrams, Symposium on Semantics of Algorithmic Languages, ed. E. Engeler, Springer Lecture Note Series No. 188, Springer-Verlag, Heidelberg, 1971.

Sm : M. Smyth--Powerdomains,Mathematical Foundations of Computer Science 76, Lecture Notes in Computer Science #45, Springer-Verlag, 1976.

T : D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager--Thoth, a Portable Real-Time Operating System, to appear in CACM.

V : J. Vuillemin--Syntaxe, Semantique, et Axiomatique d'un Langage de Programmation Simple, These de doctoral d'etat, Paris, 1974.