

Concurrent Processes and Their Syntax

GEORGE MILNE AND ROBIN MILNER

University of Edinburgh, Edinburgh, Scotland

ABSTRACT A mathematical model of concurrent computation is presented. Starting from synchronized communication as the only primitive notion, a *process* is defined as a set of communication capabilities. The domain of processes is built using the weak powerdomain construction of Smyth, which evolved from that of Plotkin.

A minimal set of operations for composing processes is defined. These operations suggest a corresponding minimal syntax—the language of *flowgraphs*—in which to specify these compositions. The concept of *flow algebra* is defined; processes and flowgraphs are examples of flow algebras.

Elsewhere it will be shown that flowgraphs are free (over a set of generators) in the category of flow algebras; here it is shown that processes are a flow algebra, and therefore constitute a suitable semantics for flowgraphs. However, we emphasize that the notion of flowgraph evolved from the notion of process and not the reverse.

KEY WORDS AND PHRASES concurrency, parallelism, process, semantics, algebraic semantics, communicating processes, flow diagrams, nondeterminism, semantic domains, continuous functions

CR CATEGORIES 4.22, 4.32, 5.21, 5.24

1. Introduction

Concurrency among computations which communicate with each other presents some of the most exciting challenges in the study of computation. Not only is it of great practical relevance, but it also appears to demand a theory of its own, which at present exists only tentatively. One aspect of this theory must be quantitative; the question of which tasks can usefully employ more than one processor, and which tasks by contrast are inherently sequential, is at present under active study. Another (perhaps *the* other) aspect is qualitative; what conceptual framework is needed to understand and discuss the behavior of concurrent computations? Although these two aspects should presumably interact, it appears to be sound methodology to separate them initially. There is evidence for the soundness of this separation in what has been done so far for sequential computation; the techniques of computational complexity and the quantitative analysis of algorithms are rather independent from those of the semantics and design of algorithmic languages and the verification of programs.

This paper deals with the qualitative aspect. We attempt to provide a part of the necessary conceptual framework by attacking the question: What sort of mathematical object is a *process*? Our approach can be outlined by analogy with sequential computation. A sequential program undoubtedly means different things to different people. It can mean to one person a description of a set of computation sequences or evaluations, and to another it can mean a mathematical function. Methods of formal semantic description of languages reflect these two viewpoints; for example, the Vienna PL/1 definition [9] and the Algol 68 report [22] adopt the first viewpoint, and the method of Scott and Strachey, exemplified by Mosses [14] and Tennent [21] among others, adopts the second. Of course

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

The work of G. Milne was supported by the British Science Research Council.

Authors' address: Department of Computer Science, James Clerk Maxwell Building, University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, Scotland.

© 1979 ACM 0004-5411/79/0400-0302 \$00.75

the two viewpoints are not incompatible, nor are they the only ones. But they are probably the main ones; a third viewpoint, the axiomatic approach of Hoare [6] and others, while very important in its implications for verification, appears to us to be different in nature in that it presupposes or demands the existence of a model of its axioms (for example to demonstrate their consistency).

But, to return to the important point in our analogy, there is little doubt that whatever *formal description* of a sequential language is adopted, the notion of a mathematical function has pervasively informed its *design*. The fact that this notion is well understood must have contributed to the success of sequential programming languages; by contrast, other notions which were imperfectly understood—consider scope, sharing, assignment, own variable, type, and many others—have clearly contributed to their shortcomings. Often these notions have been named before the notion itself is exhibited! But this “back-to-front” way of proceeding is probably the only way of arriving at a notion which is useful as well as precise.

Now consider the notion of *process*. It has certainly been named, but no real agreement exists on what is named. However, there now exist programming languages which employ processes—whatever they are—in a rather smooth and consistent style; examples are in Wirth [23] and Hoare [7]. These languages justify some optimism for the development of languages for concurrent programming, and for this reason it is all the more urgent to study mathematical concepts of process which should inform the development. Also, the intricacy of the problem of verifying concurrent programs—some positive contributions to this, such as Greif [5] and Owicki [15], have already been made—suggests that it is important to have a mathematical model in terms of which to justify the proof techniques used.

We try here to bring together the theory of continuous functions due to Scott [19], the extension of his work to nondeterminism in the construction of powerdomains by Plotkin [17] and subsequently Smyth [20], and the work of Bekic [1] and Milner [12] on processes. The latter paper preceded the work on powerdomains; one of its defects was that it only achieved nondeterminism (which arises in concurrency as soon as the relative speed of concurrent processors is left unspecified) by means of oracles, which were admitted to be clumsy and mathematically intractable. Our present approach is to use a powerdomain to define both a notion of process and a minimal set of operations for composing processes. Nondeterminism is naturally represented by the set of “next possible moves” available to a process; the work on powerdomains provides a restricted power-set construction which preserves certain desirable properties of domains—properties which allow its use in building recursively defined domains, of which our process domain is just one example.

We thereafter propose a correspondingly minimal language—the algebra of nets or flowgraphs—in which to specify process composition. We hope that this language may be useful, dressed up in convenient syntax, as a subset of future concurrency languages. It is interesting that both Wirth and Hoare, in the papers referenced above, explicitly adopt an approach in which two phases of the work of a program are kept rather separate: the phase in which a net or linkage of processes is set up, and the phase in which the net is activated. Our algebra of nets represents the first phase; whether or not this factorization into two phases is ultimately adequate, it seems to be a valuable first step in establishing a theory.

The work of Goguen, Thatcher, Wagner, and Wright [4] on initial algebra semantics provides another important element of our framework. They propose that syntax—any programming language—should be an initial algebra, in some category in which the other algebras are candidates for the semantics (via a unique homomorphism from the syntax). Indeed, we show elsewhere [13] that our algebra of nets is a free algebra in the category of *flow algebras*; the algebra of processes introduced here is just one flow algebra. In fact it is also a *continuous* flow algebra—that is, it is closed under limits of directed sets and the operations are continuous—and if we admit infinite nets or flowgraphs then the algebra of nets is also free in the category of continuous flow algebras. The process algebra presented

here is not adequate for all purposes; for example, it appears not to be sufficiently refined to reflect the possibility of *deadlock* in a net of processes, as we indicate in Section 5, though other flow algebras exist for the treatment of deadlock. A detailed analysis of other flow algebras remains to be done. As for the category of flow algebras itself, we would not claim at this stage that it is the only—or the best—possible choice; its advantage is that it has a particularly simple algebraic theory, which is likely to be of considerable help in further investigations.

We will not attempt here to compare our process model with other models of concurrent computation, but merely comment on its nature. First, we reject the idea of a global environment, such as a memory shared by all processes in a system. Processes may be composed to form other processes, and any shared resource will enter into the composition as simply one more process. In principle the behavior (or intended behavior) of a user sitting at a terminal can be expressed as another component process of a system.

Second, we take synchronized communication as a primitive notion; a process is no more than the set of its communication capabilities. Since we wish communication to be really primitive, we reject (for our model) the idea of messages being queued on a communication line; if a queue or buffer is needed, then it is another component process. We would not deny that a model in which communication lines are queues is likely to be useful at a higher level, particularly when the system as a whole is determinate (because the disciplined concurrency of queue processing preserves determinacy); see Kahn and MacQueen [8] and Zave [24] for models of this type. But queues do not directly model synchronization between computations, and it therefore seems at least worthwhile to study a more fundamental model in which synchronization is primitive.

(The present paper is a revision of a research report [11] which bears the same title. In this revision we have ensured compatibility with the companion paper [13], and included a more careful intuitive description of our model.)

2. Notation and Definitions

For the purpose of this paper, a *domain* D is an ω -algebraic complete partial order (cpo). That is, there is a partial order $\sqsubseteq$ over D such that (i) D has a minimum element $\perp_D$, and (ii) each directed set $X \subseteq D$ has a least upper bound (lub) $\sqcup X \in D$ (these conditions make D a cpo). Moreover, there is a denumerable subset $E \subseteq D$ of *isolated* elements such that for each $x \in D$, $\{e \in E; e \sqsubseteq x\}$ is directed and has x as its lub. An element e is *isolated* (the word *finite* is also used) if for all directed $X \subseteq D$,

$$e \sqsubseteq \sqcup X \Rightarrow e \sqsubseteq x \quad \text{for some } x \in X.$$

The importance of this notion of domain is that it is preserved by several operations. First, the Cartesian product of two domains is a domain. Second, for any denumerable indexing set L and any family $\{D_\lambda; \lambda \in L\}$ of domains, the *indexed sum* $S = \sum_{\lambda \in L} D_\lambda$ of the family is also a domain, where

$$S = \{\langle \lambda, d \rangle; \lambda \in L, d \in D_\lambda\} \cup \{\perp_S\},$$

in which $s \sqsubseteq s'$ iff $s = \perp_D$ or $s = \langle \lambda, d \rangle$, $s' = \langle \lambda, d' \rangle$, and $d \sqsubseteq d'$. We shall write $\langle \lambda, d \rangle \in S$ as $\lambda:d$ for clarity in what follows. We shall be concerned only with *sorts* (defined in Section 3) as indexing sets.

Third, for any pair D, E of domains, the *function domain* $D \rightarrow E$ is the set of continuous functions from D to E , ordered by

$$f \sqsubseteq f' \Leftrightarrow \forall d \in D. fd \sqsubseteq f'd.$$

A function f from D to E is *continuous* if, for every directed subset X of D , $\sqcup \{f(x); x \in X\}$ is directed in E and equal to $f(\sqcup X)$; see [19].

Finally, for any domain D there is a domain $\mathcal{P}(D)$, the *powerdomain* of D , whose members are a certain subset of the power set of D and whose ordering will be discussed

later. The results of this paper are based on a minor variant of the powerdomain construction due to Smyth [20]; similar but not identical results may also be obtained using that of Plotkin [17], which was the first powerdomain construction to be found.

3. Processes

We now introduce our domain P of processes. We have already said that a process is to be a set of communication capabilities, so our first task is to make the latter notion more precise. We assume first that a process has a collection of ports through which it may communicate with other processes. Two familiar types of communication are input and output. In the case of input, a communication capability of a process $p \in P$ indicates that p is able to acknowledge an input value v —in domain V say—and then to transform itself into another process (dependent on v) which itself consists of a new set of capabilities; an input capability may therefore be represented by an element of the domain $V \rightarrow P$. An output capability of p , on the other hand, indicates that p is ready to communicate an output value u —in domain U say—and to react to the acknowledgment of the receipt of this value by transforming itself into another process; this kind of capability may therefore be represented as an element of $U \times P$.

We wish to treat both input and output capabilities, for technical convenience, as special cases of a single kind of capability. We therefore propose that a communication consists of an *exchange of values*; a member of U will be output, or sent, in exchange for a member of V input, or received, and after the exchange a new process (dependent on the received value) will result. This more general kind of capability is thus a member of $U \times (V \rightarrow P)$. In the case of an input capability we take $U = 1$, the domain with a single member ($1 = \{0\}$), while for an output capability we take $V = 1$. In the former case we have the isomorphism $1 \times (V \rightarrow P) \simeq V \rightarrow P$, and in the latter case $U \times (1 \rightarrow P) \simeq U \times P$, so we have recovered our earlier proposal for the capability domains. In both cases the element 0 may be thought of as an acknowledgment of, or a request for, a value. The case $U = V = 1$ represents a pure synchronization capability; also, the way in which we shall compose processes with one another will ensure that a communication can only take place when both parties have appropriate matching capabilities, so that synchronization is a part of every communication. The general case in which $U \neq 1 \neq V$ may have useful applications, but we have introduced value exchange mainly as a device to provide homogeneous treatment of input, output, and pure synchronization, rather than as a conceptually necessary idea.

When we return to the process ports mentioned above, it is clear that we may require different domains U, V for different ports; one port may be used for output of integer values, for example, and another for pure synchronization. We therefore introduce a set Λ of labels (to be defined presently), and each port will be labeled by some member $\mu \in \Lambda$; we further assume a fixed assignment of pairs of domains U_μ, V_μ to each label μ . We also find it convenient to group together, in one process domain P_L , all processes whose port labels form a finite set $L \subseteq \Lambda$; we call such a set L a *sort*.

Finally then, since we have no need to determine the number of ports of a process having a given label, we define the domain P_L of processes of sort L as a particular solution of the isomorphism

$$P_L \simeq \mathcal{P} \left(\sum_{\mu \in L} (U_\mu \times (V_\mu \rightarrow P_L)) \right). \quad (1)$$

The existence of solutions of such isomorphisms (without using $\mathcal{P}$) is due to Scott; a major purpose in the work of Plotkin and Smyth was to justify the use of the powerdomain construction in such isomorphisms.

The particular solution of (1) that we choose is standard; it has the property that the function $proc \in P_L \rightarrow P_L$, defined as the least fixed point of the functional Φ given by

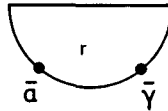
$$\Phi(g)(p) \stackrel{\text{def}}{=} \llbracket \mu : \langle u, g \circ f \rangle ; \mu : \langle u, f \rangle \in p \rrbracket, \quad (2)$$

is the identity function over P_L . To put it another way,

$$p = \bigsqcup_i \text{proc}_i(p), \quad \text{where} \quad \text{proc}_i = \Phi^i(\perp). \quad (3)$$

The set forming construction $\llbracket \dots; \dots \rrbracket$ in (2) requires careful interpretation; we shall return to this later, but for the present it may be safely interpreted as the usual $\{\dots; \dots\}$. (We use a semicolon here, rather than the more conventional notation $\{\dots|\dots\}$, since we have another use for the vertical bar.)

As a simple example of a process, we show how to define a register r storing integer values, and having sort $\{\bar{\alpha}, \bar{\gamma}\}$. (The significance of labels such as $\bar{\alpha}$ will be made clear shortly.) We may represent r by the following diagram:



and consider it to have two ports; it will accept integers “assigned” to it at the port labeled $\bar{\alpha}$ and its contents may later be “fetched” from it at the port labeled $\bar{\gamma}$, so that we have

$$U_{\bar{\alpha}} = V_{\bar{\gamma}} = 1, \quad V_{\bar{\alpha}} = U_{\bar{\gamma}} = Z,$$

where Z is the flat cpo of integers $\{\perp_Z, 0, 1, -1, 2, -2, \dots\}$. In the “initial state” r is empty, so its only capability is to accept an integer assigned; thus

$$r = \llbracket \bar{\alpha} : \langle 0, \lambda z. \text{REG}z \rangle \rrbracket,$$

where $\text{REG} \in Z \rightarrow P_{\{\bar{\alpha}, \bar{\gamma}\}}$ is given recursively by

$$\text{REG}z = \llbracket \bar{\alpha} : \langle 0, \lambda z'. \text{REG}z' \rangle, \bar{\gamma} : \langle z, K(\text{REG}z) \rangle \rrbracket.$$

Here we use the standard combinator $K = \lambda x. \lambda y. x$. Note that the process $\text{REG}z$ represents a register whose contents is z .

The significance of the overbarred label $\bar{\alpha}$ is that any process wishing to assign an integer to the register must use a port of its own, labeled α , for this purpose. Thus, a complementary pair $\{\alpha, \bar{\alpha}\}$ of labels is used to establish a communication channel between processes. We can now define our set Λ of labels precisely.

Let Δ be a denumerable alphabet of *names*. We shall use $\alpha, \beta, \gamma, \dots$ to range over Δ . Let $\bar{\Delta}$, the alphabet of *conames*, be disjoint from and in bijection with it; under this bijection $\alpha(\in \Delta) \mapsto \bar{\alpha}(\in \bar{\Delta})$. The set of *labels* is $\Lambda = \Delta \cup \bar{\Delta}$. α and $\bar{\alpha}$ are *complementary* labels. We use $\lambda, \mu, \nu, \dots$ to range over Λ (but will avoid λ in the presence of lambda-abstraction!). We define the *sign* of a label by $\text{sign}(\alpha) = +$, $\text{sign}(\bar{\alpha}) = -$, and the *name* of a label by $\text{name}(\alpha) = \text{name}(\bar{\alpha}) = \alpha$. $\bar{\lambda}$ denotes the *complement* of λ , so that $\bar{\bar{\lambda}} = \lambda$.

A finite subset L of Λ is called a *sort*; we define $\bar{L} = \{\bar{\lambda}; \lambda \in L\}$ and $\text{names}(L) = \{\text{name}(\lambda); \lambda \in L\}$.

We now turn to the operations by which processes may be expressed in terms of other processes. There are three classes of operations; the first we collectively call *process composition*, and it provides a precise understanding of communication.

(i) *Process Composition*. Let $p_1 \in P_{L_1}$ and $p_2 \in P_{L_2}$, with $L_1 = \{\alpha, \bar{\gamma}\}$ and $L_2 = \{\alpha, \bar{\beta}, \gamma\}$. They may be diagrammed thus:



Their *composite*, $p_1|p_2$ (to be defined) may then be diagrammed as follows:



That is, p_1 and p_2 are joined through their complementarily labeled pairs of ports (just one pair in this case). The sort of $p_1|p_2$ is $\{\alpha, \beta, \gamma, \bar{\gamma}\} = L_1 \cup L_2$.

The communication capabilities which constitute a process $p|p'$ will be expressed by the recursive definition of the binary operations

$$|_{L,M} \in P_L \times P_M \rightarrow P_{L \cup M},$$

and we shall frequently omit the subscripting sorts on these operations. Before giving the definition, let us look at a particular case in which p and p' each have just two capabilities:

$$p = [\alpha: \langle u_1, f_1 \rangle, \beta: \langle u_2, f_2 \rangle], \quad p' = [\bar{\alpha}: \langle v_1, g_1 \rangle, \bar{\beta}: \langle v_2, g_2 \rangle]$$

Then $p|p'$ will have, first, four capabilities of communication directly with its environment, at ports labeled $\alpha, \beta, \bar{\alpha}$, and $\bar{\beta}$ respectively. One of these is $\alpha: \langle u_1, \lambda v. (f_1 v | p') \rangle$, since p would transform itself into $f_1 v$ —which we call a *renewal* of p —on receiving a value v in exchange for u_1 , the corresponding renewal of the composite is a new composite $f_1 v | p'$, indicating that the second component is unchanged. Similarly, the first capability of p' gives a capability $\bar{\alpha}: \langle v_1, \lambda u. (p | g_1 u) \rangle$ of the composite. But since p and p' may communicate between their ports labeled α and $\bar{\alpha}$, their composite will also contain all members of $f_1 v_1 | g_1 u_1$; here the values u_1 and v_1 have been exchanged and both components are renewed. It will depend on the functions f_1 and g_1 whether or not a β - $\bar{\beta}$ communication (also possible between p and p') will remain possible between the renewals $f_1 v_1$ and $g_1 u_1$ after the α - $\bar{\alpha}$ communication, and similarly on the functions f_2 and g_2 whether or not an α - $\bar{\alpha}$ communication may follow the β - $\bar{\beta}$ communication. Even if both possibilities exist, the capabilities arising after *both* communications have occurred may differ according to the *order* in which they occurred; we would only wish to say that p and p' could perform both communications “simultaneously” if the order made no difference. It may help the reader here to think of p' as a register *REGz* as we defined it earlier, and of p as a process willing both to assign a value to p' and to fetch a value from it, it is clear that the order does make a difference in this case.

With this insight, we can now give our general definition of $|_{L,M}$

$$\begin{aligned} p|p' = & [\mu: \langle u, \lambda v. (f v | p') \rangle; \mu: \langle u, f \rangle \in p] \\ & \cup [\mu: \langle u', \lambda v'. (p | f' v') \rangle; \mu: \langle u', f' \rangle \in p'] \\ & \cup \cup [(\bar{f} u' | f' u); \mu: \langle u, f \rangle \in p, \bar{\mu}: \langle u', f' \rangle \in p']. \end{aligned} \quad (4)$$

Note that the effect of the third clause is that the overt dialogue of $p|p'$ with its environment is enriched by the possibility of covert dialogue between the component processes. Our next class of operations provides for the case in which certain ports are intended *only* for covert dialogue, and must be blocked from communication with the environment.

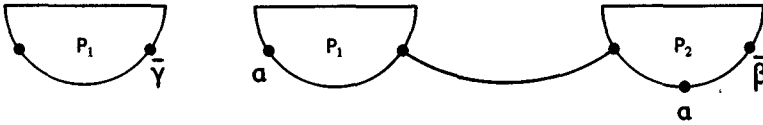
(ii) *Process Restriction*. We wish to be able to restrict the capabilities of a process p to certain of its ports, or equivalently to eliminate, from p and all its renewals, all members of certain summand domains. For any name α , we define the (postfixed) unary operation

$$\backslash_L \alpha \in P_L \rightarrow P_{L - \{\alpha, \bar{\alpha}\}}$$

as follows (again omitting the sort subscript):

$$p \backslash \alpha = [\mu: \langle u, \lambda v. (f v \backslash \alpha) \rangle; \mu: \langle u, f \rangle \in p, \mu \notin \{\alpha, \bar{\alpha}\}].$$

Then for p_1 and p_2 as illustrated above, diagrams for $p_1 \backslash \alpha$ and $(p_1 | p_2) \backslash \gamma$ (with sorts $\{\bar{\gamma}\}$ and $\{\alpha, \bar{\beta}\}$) are:



The omission of labels γ and $\bar{\gamma}$ in the second diagram signifies that the $\bar{\gamma}$ -capabilities of p_1 can only be “met” by γ -capabilities of p_2 , and conversely.

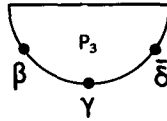
A derived operation of composition followed by restriction is sufficiently important to deserve a name. First, let us define

$$L.M = (L \cap \bar{M}) \cup (\bar{L} \cap M).$$

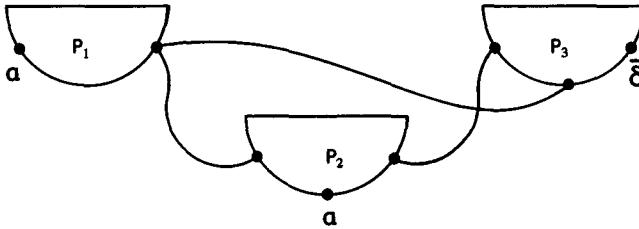
Then we can derive an operation $\|_{L,M} \in P_L \times P_M \rightarrow P_{L \cup M - L.M}$ as follows:

$$p \| p' = (p | p') \setminus \alpha_1 \dots \setminus \alpha_n,$$

where $L.M = \{\alpha_1, \bar{\alpha}_1, \dots, \alpha_n, \bar{\alpha}_n\}$. But we shall continue to take the operations $|$ and $\setminus \alpha$ as primitive, first because (as will appear) they satisfy rather simple laws, and second because they allow forking of communication lines. For example, let $p_3 \in P_{L_3}$, $L_3 = \{\beta, \gamma, \bar{\delta}\}$;



then a diagram of $((p_1 | p_2) | p_3) \setminus \beta \setminus \gamma = (p_1 | p_2) \| p_3$ is



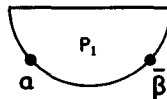
The covert dialogue between the three processes allows p_1 to communicate with either p_2 or p_3 through its port (which was) labeled $\bar{\gamma}$. This composite process could not be built with $\|$ alone.

It is easy to see that the above *flowgraph* or *net* (as we shall call these diagrams) also pictures $(p_1(p_2 | p_3)) \setminus \gamma \setminus \beta$, $((p_2 | p_3) \setminus \beta | p_1) \setminus \gamma$, and many other combinations, but not $((p_1 | p_2) \setminus \gamma | p_3) \setminus \beta \setminus \gamma = (p_1 \| p_2) \| p_3$, for example. The nets forget—within certain constraints—the order of composition and restriction.

(iii) *Process Relabeling*. Since the composition operation establishes the linkage which the port labeling dictates, some relabeling of ports may be required “before” composition. We will require this relabeling to be one-one, and to respect complements. To this end, let $S: L \rightarrow M$ be a bijection for which $S(\bar{\lambda}) = \bar{S}(\lambda)$ whenever $\lambda, \bar{\lambda} \in L$; we call S a *relabeling*. We then define the (postfixed) unary operation $[S]_L \in P_L \rightarrow P_M$ by

$$p[S] = [S(\mu): \langle u, \lambda v.(fv[S]) \rangle; \mu: \langle u, f \rangle \in p].$$

We shall often write $S: L \rightarrow M$ as $\lambda_1 \dots \lambda_k / \alpha_1 \dots \alpha_k$ or $\lambda_1 / \alpha_1, \dots, \lambda_k / \alpha_k$, which indicates that $S(\alpha_i) = \lambda_i$ and $S(\bar{\alpha}_i) = \bar{\lambda}_i$ for $1 \leq i \leq k$, and $S(\mu) = \mu$ for $\mu \notin \{\alpha_1, \bar{\alpha}_1, \dots, \alpha_k, \bar{\alpha}_k\}$. As an example, for p_1 illustrated above we may diagram $p_1[\beta/\gamma]$ as



As we mentioned earlier, the same diagram may stand for differently constructed processes; another example is that $p_1 \setminus \gamma$ and $p_1[\beta/\gamma] \setminus \beta$ will have the same diagram.

Having now described our three classes of operations over *processes*, we may point out that, as our diagrams suggest, all the operations have their syntactic analogues over *nets*. To ensure then that a net, however built by these operations, diagrams a unique process (once the processes at each node are determined), we need to describe the net operations more precisely, and then to show that all the laws which they satisfy are also satisfied by the corresponding process operations. We discuss this in a later section; the outcome is that nets under these operations—though we may indeed wish to add others—constitute a *minimal programming language for processes*, just as flowcharts of various kinds constitute a programming language for serial algorithms. In fact we believe it will turn out that our flowgraphs or nets are in some sense a generalization of these kinds of flowchart, but we shall not pursue this further here.

4. Examples of Processes

(1) *Registers and Memories*. Recall the register process r , of sort $\{\bar{\alpha}, \bar{\gamma}\}$, defined in Section 3:

$$r = [\bar{\alpha}:\langle 0, \lambda z. REGz \rangle], \\ REGz = [\bar{\alpha}:\langle 0, \lambda z'. REGz' \rangle, \bar{\gamma}:\langle z, K(REGz) \rangle].$$

We shall use it to provide a more definite example of process communication, which might arise if we used processes to define the semantics of a concurrent programming language.

First, recall the “ $\parallel$ ” operation defined in Section 3. It can be shown that this operation may be defined directly by recursion (instead of derived from composition and restriction) as follows. If $p \in P_L$ and $p' \in P_M$ then

$$p \parallel p' = [\mu:\langle u, \lambda v.(fv \parallel p') \rangle; \mu:\langle u, f \rangle \in p, \mu \in L - L.M] \\ \cup [\mu:\langle u', \lambda v'.(p \parallel f'v') \rangle; \mu:\langle u', f' \rangle \in p', \mu \in M - L.M] \\ \cup \cup [(fu' \parallel f'u); \mu:\langle u, f \rangle \in p, \bar{\mu}:\langle u', f' \rangle \in p'].$$

This differs from the definition of “ $\parallel$ ” only in the restriction (in the first two clauses) that $\mu \notin L.M$; that is, labels used for internal communication are not to be used for external communication.

Now suppose that a concurrent programming language contains the following simple and useless program:

```
begin integer x;
  x := 1 par x := 2,
  output x;
end
```

where the infix **par** means that the operations of both parts are to be arbitrarily interleaved. Clearly the program may output either 1 or 2. We want to model it by a process p of sort $\{\rho\}$, where ρ is the label of its output port; since its *internal* communications with the (local) program variable x are invisible from the outside we should obtain

$$p = [\rho:\langle 1, K\emptyset \rangle, \rho:\langle 2, K\emptyset \rangle],$$

where the renewal $\emptyset$ indicates that the program does nothing further after output.

But we should be able to build p from a register process r of sort $\{\bar{\alpha}, \bar{\gamma}\}$, and a process p_0 of sort $\{\alpha, \gamma, \rho\}$ representing the body of the program, using the combination

$$p = (p_0 | r) \setminus \alpha \setminus \gamma = p_0 \parallel r.$$

Now it is reasonable that, from the semantic definition of our language, we should obtain

$$p_0 = [\alpha:\langle 1, Kp_1 \rangle, \alpha:\langle 2, Kp_2 \rangle],$$

where

$$p_1 = [\alpha: \langle 2, Kp_3 \rangle], \quad p_2 = [\alpha: \langle 1, Kp_3 \rangle]$$

and (corresponding to the *output* statement)

$$p_3 = [\gamma: \langle 0, \lambda v. [\rho: \langle v, K\emptyset \rangle] \rangle].$$

Then, using the direct recursive definition of the “ $\parallel$ ” operation, we can compute

$$p = p_0 \parallel r = (p_1 \parallel REG1) \cup (p_2 \parallel REG2),$$

reflecting the two possible first communications with the register; furthermore,

$$\begin{aligned} p_1 \parallel REG1 &= p_3 \parallel REG2 \\ &= [\rho: \langle 2, K\emptyset \rangle] \parallel REG2 \\ &= [\rho: \langle 2, K\emptyset \rangle], \end{aligned}$$

while a similar computation yields $p_2 \parallel REG2 = [\rho: \langle 1, K\emptyset \rangle]$. We have therefore obtained

$$p = [\rho: \langle 1, K\emptyset \rangle, \rho: \langle 2, K\emptyset \rangle],$$

as expected.

The composition operation also allows us to build a multicell memory. Suppose we require an n -cell memory, whose i th cell is assigned at a port labeled $\bar{\alpha}_i$ and fetched at a port labeled $\bar{\gamma}_i$. First, let $R = \{\alpha, \gamma\}$, so that $\bar{R}$ is the sort of our register r ; then let S_i be the relabeling $\alpha_i \gamma_i / \alpha \gamma$, and let $R_i = S_i(R)$ and $r_i = r[S_i]:\bar{R}_i$. Then the n -cell memory m_n of sort $\bar{M}_n$, where $M_n = \bigcup_{i=1}^n R_i$, is simply

$$m_n = r_1 | r_2 | \dots | r_n.$$

(Since registers cannot communicate with each other, this is in fact equivalent to $r_1 \parallel r_2 \parallel \dots \parallel r_n$.) The order of combination of the r_i is immaterial, since we later show that $|$ is associative and commutative.

Now if $p \in P_L$ is some process which wishes to use some or all of the registers in m_n —that is, we expect that $L \cap M_n \neq \emptyset$ —then $p | m_n$ is the process which results from linking p with m_n . p may well be composed of separate processes which wish to “share” m_n . Then if p' is some further process which needs to share m_n we can build

$$(p | m_n) | p' = (p | p') | m_n.$$

On the other hand, if we wish to dedicate m_n to p , or to $p | p'$, we may build

$$(p | m_n) \setminus M_n \quad \text{or} \quad ((p | p') | m_n) \setminus M_n,$$

where $\setminus M_n$ abbreviates $\setminus \alpha_1 \setminus \gamma_1 \dots \setminus \alpha_n \setminus \gamma_n$.

(2) *Binary Semaphores.* We suppose that processes using a semaphore, as defined by Dijkstra [3], will each have a port labeled π through which they will request the semaphore (the P operation), and a port labeled ϕ through which they will release it (the V operation).

A binary semaphore $s \in P_S$, where $S = \{\pi, \phi\}$, may be defined recursively by

$$s = [\bar{\pi}: \langle 0, Ks' \rangle], \quad s' = [\bar{\phi}: \langle 0, Ks \rangle],$$

where we assume that $U_\pi = V_\pi = U_\phi = V_\phi = 1$; that is, the communications of the semaphore are pure synchronizations. s is the “free” semaphore, and s' the “busy” semaphore. Analogous to the case for registers, if we wish to devote the semaphore s to a set $p_1 \dots p_n$ of processes $p_i \in P_L$, where we expect in general that $S \subseteq L_i$, we build

$$(p_1 | \dots | p_n | s) \setminus S.$$

(3) *General Semaphores.* A general semaphore may be represented by $sem(z) \in P_S$, $z > 0$ initially, where $S = \{\pi_1, \pi_2, \phi\}$, and where $sem \in Z \rightarrow P_S$ is defined by

$$\begin{aligned} sem(z) &= [\bar{\pi}_1: \langle 0, K(sem'(z-1)) \rangle, \bar{\phi}: \langle 0, K(sem(z+1)) \rangle] \quad \text{if } z > 0, \\ sem(0) &= [\bar{\pi}_1: \langle 0, K(sem(-1)) \rangle, \bar{\phi}: \langle 0, K(sem(+1)) \rangle], \\ sem(z) &= [\bar{\pi}_1: \langle 0, K(sem(z-1)) \rangle, \bar{\phi}: \langle 0, K(sem'(z+1)) \rangle] \quad \text{if } z < 0, \end{aligned}$$

where

$$sem'(z) = \llbracket \pi_2: \langle 0, K(sem(z)) \rangle \rrbracket.$$

(Since only pure synchronizations are involved, we could have adopted some abbreviative device in order to avoid the repeated $\langle 0, K(\dots) \rangle$. Note also that this definition by cases could be expressed formally as a single equation, using a conditional function and union.) This definition reflects Dijkstra's definition of semaphores [3] in which a process q that performs a P operation (on the semaphore) decreases the value of the semaphore by 1.

If the resulting value is nonnegative then q can complete a P operation, which it performs in two parts (on lines π_1 and π_2). If the resulting value is negative then q may only perform the first part (π_1) of the P operation, and may then be thought of as on some sort of waiting list.

A process q' that performs a V operation on a semaphore increases the value of that semaphore by 1. If the resulting value is positive, then the operation has no further effect; if the resulting value is nonpositive, any one of the processes booked on the waiting list is removed, and this process can then continue.

We can see that any process which communicates with $sem(z)$ and is awaiting a communication on its line labeled by π_2 will be waiting for "enough" other processes to communicate with the semaphore on their ϕ communication lines.

We must emphasize that our process model in no way solves the fair scheduling problem; it is possible that one process q may regain access to the semaphore infinitely often, while another process q' never even enters the waiting list. But in this definition, once q' has entered the waiting list (with a π_1), any ϕ performed by q may not be followed by a π_1 from q until q' , or some other waiting process, has completed its P operation (with a π_2).

Suppose, in particular, we wish to use a semaphore to control the access of a set $\{q_1, \dots, q_n\}$ of processes to a resource process r . If we wish the maximum number of processes simultaneously accessing r to be m , where $m \leq n$, then we build $((q|sem(m))\backslash S)|r$ where $q = q_1 | \dots | q_n$.

Here we assume that each process q_i performs its π_1 , π_2 , and ϕ operations in a disciplined manner, namely $\pi_1, \pi_2, \phi, \pi_1, \pi_2, \phi, \dots$.

Now let $s^* \in P_S$ be the process which permits all sequences of communications on its three lines $\bar{\pi}_1$, $\bar{\pi}_2$, and $\bar{\phi}$; it is given by

$$s^* = \llbracket \bar{\pi}_1: \langle 0, Ks^* \rangle, \bar{\pi}_2: \langle 0, Ks^* \rangle, \bar{\phi}: \langle 0, Ks^* \rangle \rrbracket.$$

We would expect that in general

$$(q|sem(m))\backslash S \subsetneq (q|s^*)\backslash S \quad \text{when } m < n,$$

but that

$$(q|sem(n))\backslash S = (q|s^*)\backslash S.$$

We have not attempted a proof of this.

(4) *Queues.* Let Z^* be the flat domain of finite sequences of (defined) integers, with $\perp_{Z^*}$ adjoined. We assume the functions

$$\begin{aligned} \hookrightarrow : Z^* \times Z &\rightarrow Z^* & (\text{add new last element}), \\ \hookleftarrow : Z \times Z^* &\rightarrow Z^* & (\text{add new first element}) \end{aligned}$$

to be strict. We use ϵ to stand for the empty sequence.

An initially empty queue, which accepts elements at port α and yields them up at port $\bar{\gamma}$, is $queue(\epsilon): \{\bar{\gamma}, \alpha\}$, where $queue: Z^* \rightarrow P_{\{\bar{\gamma}, \alpha\}}$ is given by

$$\begin{aligned} queue(\epsilon) &= \llbracket \alpha: \langle 0, \lambda z. queue(\epsilon \hookrightarrow z) \rangle \rrbracket, \\ queue(z \hookrightarrow s) &= \llbracket \alpha: \langle 0, \lambda z'. queue(z \hookrightarrow s \hookrightarrow z') \rangle, \bar{\gamma}: \langle z, K(queue(s)) \rangle \rrbracket, \\ queue(\perp) &= \perp. \end{aligned}$$

Now a simple theorem about queues, or buffers (of unbounded capacity) as they may

equally be called, is that the serial composition of two buffers behaves exactly as a single buffer. We state the theorem and outline its proof, even though it is straightforward, both as an illustration of the flow operations and to emphasize that if two agents have the same behavior it means that their meanings (processes) are identical.

First, let us abbreviate $queue(s)[\gamma'/\gamma, \alpha'/\alpha]$ by $queue(s)[\gamma', \alpha']$.

THEOREM.

$$queue(s_2)[\gamma, \beta] \parallel queue(s_1)[\beta, \alpha] = queue(s_2 \smallfrown s_1)[\gamma, \alpha],$$

where $\smallfrown$ is the usual concatenation of sequences, and elements in the queues "move" from right to left.

PROOF. We require two lemmas, whose proofs we omit.

LEMMA 1.

$$queue(s_2)[\gamma, \beta] \parallel queue(z \smallfrown s_1)[\beta, \alpha] = queue(s_2 \smallfrown z)[\gamma, \beta] \parallel queue(s_1)[\beta, \alpha].$$

LEMMA 2.

$$queue(s)[\gamma, \beta] \parallel queue(\epsilon)[\beta, \alpha] = queue(s)[\gamma, \alpha].$$

Now if s_1 is undefined the theorem is immediate. Otherwise:

$$queue(s_2)[\gamma, \beta] \parallel queue(s_1)[\beta, \alpha] = queue(s_2 \smallfrown s_1)[\gamma, \beta] \parallel queue(\epsilon)[\beta, \alpha]$$

by repeated use of Lemma 1 (expanding s_1 to $z_1 \smallfrown \dots \smallfrown z_n \smallfrown \epsilon$), and Lemma 2 completes the proof. $\square$

(5) *Card Reader.* As a final example, consider a process which models a card reader. This example illustrates how we can model both hardware and software computing agents without distinction, which is what we require since we are concerned with the behavior of computing agents and not with their implementation.

We model an actual card reader, the DIGITAL CR11 which is compatible with the PDP11 system philosophy of peripheral handling via status register interrogation. This is described in [16], and consists of four distinct units: a status register, a data buffer register, the actual reading mechanism, and the card reader driver. The first three components are hardware, while the latter may be thought of as a program whose description is in fact taken from Wirth [23].

Each of these four components can be modeled separately to give the processes which appear in Figure 1. In the figure we show the components separately, in order to indicate their sorts, and the dotted lines show where communication occurs between them. A

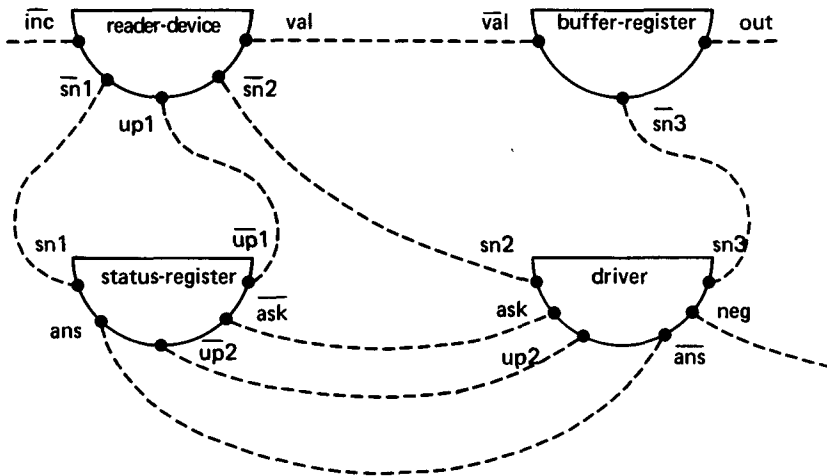


FIG 1

process which we call *cardreader*, a member of $P_{\{\overline{inc}, out, neg\}}$, may then be constructed from our four components using the $\parallel$ combinator; thus

$$\text{cardreader} = \text{reader-device} \parallel \text{status-register} \parallel \text{buffer-register} \parallel \text{driver}.$$

This process inputs cards at $\overline{inc}$, outputs characters at *out*, and outputs a delimiter at *neg*.

The component processes will not be defined here (they are defined in [10]), but they may be described informally as follows.

(1) The status-register process represents the various conditions the hardware may be in; it is updated by the reader-device process using the “up1” communication line, and by the driver process using the “up2” line. The driver also uses the “ask” and “ans” lines to interrogate the status-register, while the “sn1” line is used by the status-register as a synchronization to initiate the action of the reader-device process.

(2) The reader-device process, once brought into action via the “sn1” line, uses the “up1” line to update the status-register and input a data card on the “inc” line. Values representing characters belonging to the card are then output on the “val” line sequentially. The “sn2” line is used to synchronize the passing of each card column to the buffer-register.

(3) The buffer-register process receives column values from the reader-device on the “val” line. These values are passed on to the “outside world” one at a time on the “out” line, and this action is synchronized by the driver on line “sn3”.

(4) The driver process controls the actions of the previous components via the synchronization lines “sn2” and “sn3” together with the updating of the status-register by the “up2” line and its interrogation via the “ask” and “ans” lines. The driver controls when a new card is read by the reader-device, makes sure that the contents of the buffer-register have been output before the next character is transmitted from the reader-device, and outputs a delimiter on the “neg” line after a full 80 characters have been output on the “out” line.

In terms of the precise descriptions of these components, we then have an exact definition of the composite card reader process. We wish to determine whether this process represents the behavior that we expect a card reader to have. This expected behavior is represented by an independently defined process *crspec*, also of sort $\{\overline{inc}, out, neg\}$;

$$\text{crspec} = \llbracket \overline{inc} : \langle 0, \lambda c. \text{send}(c, 1) \rangle \rrbracket,$$

where $\text{send} \in \text{CARD} \times N \rightarrow P_{\{\overline{inc}, out, neg\}}$ is defined recursively as follows:

$$\text{send}(c, j) = \begin{cases} \llbracket \overline{neg} : \langle -1, K(\text{crspec}) \rangle \rrbracket & \text{if } j = 81, \\ \llbracket \text{out} : \langle \text{change}(c_j), K(\text{send}(c, j+1)) \rangle \rrbracket & \text{otherwise,} \end{cases}$$

where $\text{change} \in \text{COLUMN} \rightarrow N$. Here we assume that

$$\begin{aligned} V_{inc} &= V_{neg} = V_{out} = 1, \\ U_{inc} &= \text{CARD}, \quad U_{neg} = \{-1\}, \quad U_{out} = N = \{\perp_N, 0, 1, \dots\}, \end{aligned}$$

where $\text{CARD} = \text{COLUMN}^{80}$ and *COLUMN* is the domain of encoded card columns.

We have in fact shown in [10], by a straightforward inductive proof, that *cardreader* = *crspec*; that is, our composite *cardreader* process does indeed meet its specification. In fact, this proof is perhaps better regarded as a correctness proof of the driver program. Given the three hardware components, the driver must control their actions in such a way that a card is read and a sequence of values output together with a delimiter in a correct manner.

5. A Weak Powerdomain Construction

We here summarize the important features of a powerdomain construction essentially due to Smyth [20], but with a minor addition (to be precise, the addition of the empty set). We state without proof the main results we shall need; they can be found for the most part in

the above-referenced paper (the addition of the empty set requires only minor modifications).

Starting with a domain D , the construction of the weak powerdomain $\mathcal{P}(D)$ may be outlined as follows:

(1) Form the collection $\mathcal{F}(D)$ of finitely generable subsets of D . A set $X \subseteq D$ is finitely generable if (a) there exists an infinite finitely branching tree T , which may contain finite as well as infinite paths (Smyth forbids finite paths; the only effect of including them is to admit $\emptyset$ as finitely generable), whose nodes are labeled with elements of D in such a way that the labels of each path form a $\sqsubseteq$ -chain in D , and (b) X is the set of lub's of chains which label the infinite paths of T .

(2) Over $\mathcal{F}(D)$, define a preorder $\sqsubseteq_0$ as follows:

$$X \sqsubseteq_0 X' \text{ iff } \forall x' \in X', \exists x \in X. x \sqsubseteq x'.$$

(3) Define the equivalence $\approx_0$ over $\mathcal{F}(D)$ by

$$X \approx_0 X' \text{ iff } X \sqsubseteq_0 X' \text{ and } X' \sqsubseteq_0 X.$$

We then have a domain $(\mathcal{F}_0(D), \sqsubseteq) = (\mathcal{F}(D)/\approx_0, \sqsubseteq_0/\approx_0)$; that is, $\mathcal{F}_0(D)$ is the quotient of $\mathcal{F}(D)$ by the equivalence $\approx_0$. In fact the (equivalence classes containing) finite sets of isolated elements turn out to be exactly the isolated elements of $\mathcal{F}_0(D)$.

Now for the sake of concreteness we choose a canonical member of each $\approx_0$ equivalence class in $\mathcal{F}_0(D)$ as follows: Define, for each $X \in \mathcal{F}_0(D)$, its *right closure*

$$\mathcal{RC}(X) = \{y \in D; \exists x \in X. x \sqsubseteq y\}.$$

It is clear that $\mathcal{RC}(X) \approx_0 X$ and that $X \approx_0 Y$ iff $\mathcal{RC}(X) = \mathcal{RC}(Y)$; moreover, $X \sqsubseteq_0 Y$ iff $\mathcal{RC}(X) \supseteq \mathcal{RC}(Y)$. In conclusion, we may regard $\mathcal{F}_0(D)$ as the collection of right-closed finitely generable sets under the *superset* ordering ($X \sqsubseteq X'$ iff $X \supseteq X'$).

We shall choose $\mathcal{F}_0(D)$, so regarded, as our $\mathcal{P}(D)$. Plotkin's (strong) powerdomain $\mathcal{P}_m(D)$ is constructed with a stronger preorder $\sqsubseteq_M$, which causes fewer sets to be identified under the induced equivalence $\approx_M$. However, it is not so simply amenable to inclusion of the empty set; how to include $\emptyset$ in $\mathcal{P}_M(D)$ is still an open problem. We would have wished our results on processes to pertain equally well using $\mathcal{P}_M(D)$, but the addition of $\emptyset$ together with the superset ordering makes $\mathcal{F}_0(D)$ comparatively easy to handle, and we believe that for many applications $\mathcal{F}_0(D)$ may be adequate.

Intuitively, the loss of information by identification of elements in $\mathcal{F}_0(D)$ has two aspects. First, suppose that some $X \in \mathcal{F}_0(D)$ contains $\perp_D$ as an element; then it follows that $X \approx_0 \{\perp_D\}$, the minimum element of $\mathcal{F}_0(D)$. This means that any nondeterministic program which *may* fail to terminate is as bad as one which *never* terminates. Second, since $\emptyset \in \mathcal{F}_0(D)$ can be argued to correspond to deadlock, and since $X \cup \emptyset = X$, our model is insensitive to the possibility that a nondeterministic program *may* deadlock (though we can demonstrate that it *must* deadlock if it denotes $\emptyset$ in a suitable domain). We should emphasize that this insensitivity is particular to our present model; we are studying other interpretations of nets in which possible but not inevitable deadlock can indeed be represented faithfully.

We now collect together the properties of $\mathcal{P}(D)$ that we need.

PROPOSITION. *In weak powerdomains:*

- (1) $X \sqsubseteq X' \Leftrightarrow X \supseteq X'$.
- (2) D is minimum, $\emptyset$ is maximum, in $\mathcal{P}(D)$.
- (3) The functions $\cup$ and $\cap$ are continuous; in fact

$$X \cap Y = X \sqcup Y, \quad X \cup Y = X \sqcap Y.$$

- (4) For any directed set S of sets, $S \subseteq \mathcal{P}(D)$, $\sqcup S = \cap S$.

(5) Function extension. If $f \in D \rightarrow E$ is continuous, then so is $\hat{f} \in \mathcal{P}(D) \rightarrow \mathcal{P}(E)$, where $\hat{f}(X) = \mathcal{RC}\{f(x); x \in X\}$.

(6) Big union. The function $\bigcup: \mathcal{P}(\mathcal{P}(D)) \rightarrow \mathcal{P}(D)$ is continuous.

(7) Singleton. The function $\llbracket - \rrbracket \in D \rightarrow \mathcal{P}(D)$ is continuous, where $\llbracket x \rrbracket = \mathcal{RC}\{x\}$.

(8) Restriction. The function $\backslash \in \mathcal{P}(D) \times (D \rightarrow 2) \rightarrow \mathcal{P}(D)$ is continuous, where 2 is $\{\perp, \top\}$ with $\perp \sqsubseteq \top$, and

$$X \backslash f = \mathcal{RC}\{x \in X; f(x) = \perp\}.$$

OUTLINE OF PROOFS AND REMARKS. We omit proofs of (1)–(5); they are routine in the light of Smyth's work.

For (6), note that as each $X \in S$ is right-closed (since $X \in \mathcal{P}(D)$), $\bigcup S$ is automatically right-closed. A simple argument, via generating trees, shows that $\bigcup S$ is also finitely generable, hence in $\mathcal{P}(D)$; the proof of continuity of $\bigcup$ is then straightforward.

For (7), merely note that $\llbracket x \rrbracket$ is not of course a singleton set in general; in particular $\llbracket \perp_D \rrbracket = D = \perp_{\mathcal{P}(D)}$.

The purpose of (8), the function $\backslash$, is to provide a basis in terms of which to define the set forming construct

$$\llbracket \dots; \dots \rrbracket,$$

which has already been used informally. Intuitively, $X \backslash f$ is the result of first filtering out all $x \in X$ such that $f(x) = \top$, then applying right closure. $\backslash$ is easily shown to be monotonic in each argument. For continuity, it is sufficient to show that the corresponding function without the right closure operation is continuous over the preorder $(\mathcal{F}(\mathcal{D}), \sqsubseteq_0)$; that is, for

$$\backslash \in \mathcal{F}(D) \times (D \rightarrow 2) \rightarrow \mathcal{F}(D),$$

where $X \backslash f = \{x \in X; f(x) = \perp\}$, we require continuity in f and in X . The former is rather easy. For the latter, we first fix f , then proceed in outline as follows (but the reader should refer to [20] to fill in the gaps).

It is enough to show that if X has a generating tree T whose levels are the finite sets of finite elements $\{E_i; i \geq 0\}$, then the filtered sets $E_i \backslash f$ are the levels of a generating tree, which we may call $T \backslash f$, which generates $X \backslash f$. The details of this are routine. $\square$

We now justify certain more or less abbreviative uses of $\llbracket \dots; \dots \rrbracket$ to form sets.

COROLLARY 1. Let $X \in \mathcal{P}(D)$, $p \in D \rightarrow T$, $h \in D \rightarrow E$. Then the set

$$\llbracket h(x); p(x), x \in X \rrbracket \in \mathcal{P}(E)$$

varies continuously in h , p , and X , when interpreted as

$$\mathcal{RC}\{h(x); x \in X, p(x) \sqsubseteq \text{true}\}.$$

PROOF. It is just $\hat{h}(X \backslash (\lambda x. p(x) \rightarrow \perp, \top))$. $\square$

Our main concern in this paper is with certain functions from $\mathcal{P}(D)$ to $\mathcal{P}(E)$, where $D = \sum_{\lambda \in L} D_\lambda$ and $E = \sum_{\mu \in M} E_\mu$, which can be informally described as follows: For any $X \in \mathcal{P}(D)$ form $Y \in \mathcal{P}(E)$ by first rejecting all the members of some summands, then applying certain functions to the remaining elements and performing right closure.

COROLLARY 2. Let $D = \sum_{\lambda \in L} D_\lambda$, $E = \sum_{\mu \in M} E_\mu$, with L and M finite. Let $X \in \mathcal{P}(D)$ and $g_v \in D_v \rightarrow E_v$ for each $v \in N \subseteq L \cup M$. Then the set

$$\llbracket v: g_v(d); v \in N, v: d \in X \rrbracket \in \mathcal{P}(E)$$

varies continuously in each g_v and in X , when interpreted as

$$\mathcal{RC}(\{v: g_v(d); v \in N, v: d \in X\} \cup \{\perp_E; \perp_D \in X\}).$$

PROOF. It is a finite union of sets, typically

$$\llbracket h_v(x); p_v(x), x \in X \rrbracket \quad (\text{interpreted as in Corollary 1})$$

where

$$p_v(x) = \begin{cases} \text{true} & \text{if } x = v:d, \\ \text{false} & \text{if } x = \lambda:d, \lambda \neq v, \\ \perp & \text{if } x = \perp, \end{cases}$$

and

$$h_v(x) = \begin{cases} v:g_v(d) & \text{if } x = v:d, \\ \perp & \text{if } x = \lambda:d, \lambda \neq v \text{ or if } x = \perp. \end{cases}$$

The rest follows by Corollary 1 and the continuity of $\cup$. $\square$

For the remainder of the paper we shall normally use $\llbracket \dots; \dots \rrbracket$ in the form introduced in Corollary 2, with that interpretation. Sometimes we use a mild generalization, such as a set depending on two set parameters rather than one.

The main points to bear in mind with this construct are simply that it denotes a right closure, and that it denotes $\perp$ whenever the set parameter is $\perp$ (i.e. it is strict in the set parameter). Otherwise it can be treated just as $\{\dots; \dots\}$.

6. Flowgraphs and Flow Algebras

As promised in an earlier section, we now discuss more precisely the notion of *flowgraph* or *net* (we will use these terms indiscriminately). We have already used examples of nets informally to explain our process operations. What will emerge is that in a precise sense nets are a suitable programming language for processes, just as conventional flowcharts are a suitable language to describe serial computation. This analogy can be pursued further. Many conventional languages are in part a way of describing flowcharts—and indeed the same flowchart may be described by more than one program; in the same way we shall find that the same flowgraph may be described by more than one *flow expression*.

In serial programming we have become accustomed to ignoring the flowchart as an intermediary between a program and its semantics; this is justified partly by the fact that programs which use procedures (perhaps with recursion) are not readily charted, and partly by the fact that serial programs are often readily intelligible in themselves. It is likely that a programming language for concurrent computation will also contain features which are not readily translatable into nets, but we believe that for the subset of programs which *are* so translatable the concept of net is of great importance to the understanding of the program.

Consider in this context the language MODULA of Wirth [23], which is an experiment in language design for modular multiprogramming. The execution of a MODULA program can be divided into two phases. In the first phase what we could call a net is established; roughly, its nodes are described by *modules*, and their connection by *interface modules*. In the second phase, the processes denoted by the modules communicate via these connections; the net remains fixed.

In describing the action of such programs, Wirth (among others) finds it convenient to diagram his nets; it appears that however well structured our “multiprogram” is, the diagram—either drawn or imagined—is almost indispensable to its understanding. So the formal description of nets appears to be well motivated. We also believe that to describe nets as a distinguished member of a class of algebras which we call *flow algebras* promises to give a uniform framework within which to discuss their semantics. What follows is very much in the spirit of the initial algebra semantics of Goguen, Thatcher, Wagner, and Wright [4]—indeed it is a special case. We depend heavily on the algebraic approach to which these authors have strongly contributed, and which perhaps originates with Burstall and Landin [2].

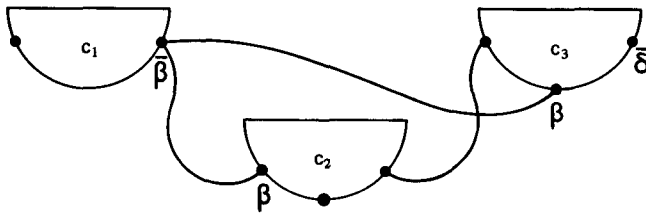
We should also mention here an important paper of Scott [18]. In that paper programs which can iterate or “recurse” are represented as infinite flow diagrams which may be described by finite expressions in a programming language; their semantics is done using

the fixed points of continuous functionals. Here, we are only concerned with finite nets, though an important extension of our work is to consider infinite nets—corresponding to programs which could use an unbounded number of processors concurrently in their execution.

We will not define nets completely formally here; we refer the reader to [13] where the formal definition is required for proofs. In this paper we only need to say enough to make it appear reasonable to the reader that the operations over nets satisfy a simple equational theory.

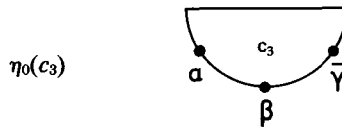
First, recall the notions of *name*, *label*, and *sort* introduced in Section 3. In addition to this, we need a set Γ of *generators*, which is partitioned into $\{\Gamma_L; L \text{ is a sort}\}$. We will use $c, d, \dots$ to range over Γ ; if $c \in \Gamma_L$ then c has sort L , and we write $c:L$. Generators may later be interpreted as processes (a basic set of processors, from which we wish to build others), but here they are left uninterpreted.

A *net* or *flowgraph* then has a set of *nodes*, to each of which is assigned a generator. Each node with generator $c:L$ has one *port* for each member of L . An *arc* may join a pair of ports belonging to distinct nodes. Each port may be assigned a *label*, in which case it is an *outer* port of the net; otherwise it is an *inner* port. Besides this, if an arc joins two ports then either both ports are inner, or they are both outer and have complementary labels. If c_1, c_2, c_3 have sorts $\{\alpha, \bar{\gamma}\}, \{\gamma, \delta, \bar{\epsilon}\}, \{\alpha, \beta, \bar{\gamma}\}$ then this is a net:



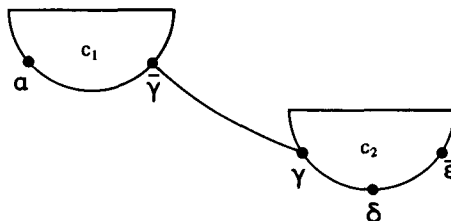
Here we have represented the bijection between the ports of a node and the sort of its generator by assuming some arbitrary ordering of labels and drawing the ports from left to right respecting this order, but note that the label of an outer port in the net need not be the label in the generator sort to which the port corresponds.

We are not interested in *all* nets, but only those which may be built from the generators by the operations (to be defined now over nets, not processes) of composition, restriction, and relabeling. First then, there is a natural injection η_0 from generators into nets, under which for example $c_3: \{\alpha, \beta, \bar{\gamma}\}$ becomes the net



which we also denote by c_3 . Now in general the *sort* L of a net g is the set of labels on its outer ports, and we write $g:L$. Let G_L be the set of generable nets of sort L .

(i) *Net Composition.* If $g:L$ and $g':L'$, the composite net $g|g':L \cup L'$ is formed by joining every outer port in g to every outer port of g' whose label complements that of g . Thus for c_1, c_2 with the sorts above mentioned, $c_1|c_2: \{\alpha, \gamma, \bar{\gamma}, \delta, \bar{\epsilon}\}$ is the net



(ii) *Net Restriction.* If $g:L$, the α -restricted net $g\backslash\alpha:L - \{\alpha, \bar{\alpha}\}$ is formed by dropping the labels α and $\bar{\alpha}$ from outer ports of g .

(iii) *Net Relabeling.* If $g:L$ and $S:L \rightarrow M$ is a relabeling (as defined in Section 3), then $g[S]$ is formed by replacing every label μ in g by $S(\mu)$. Continuing the example, the three-node net first illustrated is then

$$((c_1|c_2)[\beta/\gamma]\backslash\delta|c_3[\epsilon/\alpha, \delta/\gamma])\backslash\epsilon.$$

We have thus simultaneously defined the net operations and the sets G_L of generable nets. A characterization of the generable nets is given in [13] and we do not consider it here. But note that (as we said in Section 3, concerning processes) the same net may be built in many ways. What are the different ways? That is, considering the sorted algebra G of generable nets, with phyla G_L and operations

$$\begin{aligned} | &\in G_L \times G_{L'} \rightarrow G_{L \cup L'} && \text{(an operation for each pair } L, L' \text{ of sorts),} \\ \backslash\alpha &\in G_L \rightarrow G_{L - \{\alpha, \bar{\alpha}\}} && \text{(an operation for each sort } L \text{ and name } \alpha), \\ [S] &\in G_L \rightarrow G_{S(L)} && \text{(an operation for each sort } L \text{ and relabeling } S), \end{aligned}$$

which *derived* operations are equivalent?

PROPOSITION. *The following identities hold over nets, where we assume $x_i:L_i$ are arbitrary nets, α and β arbitrary names, and R and S arbitrary relabelings.*

LAWS OF COMPOSITION.

$$(C1) \ x_1|x_2 = x_2|x_1.$$

$$(C2) \ x_1|(x_2|x_3) = (x_1|x_2)|x_3.$$

LAWS OF RESTRICTION

$$(R1) \ x_1\backslash\alpha = x_1 \text{ when } \alpha, \bar{\alpha} \notin L_1.$$

$$(R2) \ x_1\backslash\alpha\backslash\beta = x_1\backslash\beta\backslash\alpha.$$

$$(R3) \ (x_1|x_2)\backslash\alpha = (x_1\backslash\alpha)|(x_2\backslash\alpha) \text{ when } \alpha, \bar{\alpha} \notin L_1 \cap \bar{L}_2.$$

LAWS OF RELABELING

$$(S1) \ x_1[\] = x_1 \quad ([\] \text{ is the identity relabeling}).$$

$$(S2) \ x_1[R][S] = x_1[S \circ R].$$

$$(S3) \ x_1\backslash\alpha[R] = x_1[R \cup \lambda/\alpha]\backslash\beta \text{ when } \alpha \in L_1 \cup \bar{L}_1, \text{ name}(\lambda) = \beta \notin \text{names}(R \text{ and } R).$$

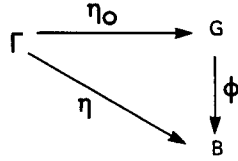
$$(S4) \ (x_1|x_2)[R] = x_1[R \upharpoonright L_1]|x_2[R \upharpoonright L_2]$$

It is a routine but tedious matter to verify these laws from a formal definition of nets. What is their importance? First, they are simple. Second, as is shown in [13], they allow any net expression, i.e. application of a derived operation to generators, to be expressed in a normal form in which all relabelings are innermost and all restrictions are outermost; that is, any generable net can be expressed in the form

$$(c_1[S_1]) \dots (c_n[S_n])\backslash\alpha_1 \dots \backslash\alpha_p.$$

Third and most important, the laws are *complete* for generable nets; any two net expressions which denote the same net can be proved equal by the laws. To see the full consequence of this fact, we need to define the notion of a *flow algebra*, or a Δ -flow algebra if we wish to make the set Δ of names explicit.

A Δ -flow algebra B consists of a phylum (set) B_L for each sort L , a binary operation $|_{L,M}^B \in B_L \times B_M \rightarrow B_{L \cup M}$ for each pair of sorts L and M , a unary operation $\backslash_L^B \alpha \in B_L \rightarrow B_{L - \{\alpha, \bar{\alpha}\}}$ for each sort L and name α , and a unary operation $[S]_L^B : B_L \rightarrow B_M$ for each pair of sorts L and M and relabeling $S:L \rightarrow M$, such that the nine laws hold (where the superfix B and sort suffices $L, M, \dots$ should strictly be supplied on each operation symbol). The completeness of the laws, which is shown in [13], then has the consequence that the flow algebra G of nets is free over Γ in the category of flow algebras, in which the morphisms are homomorphisms. This means that for any flow algebra B and sorted map $\eta:\Gamma \rightarrow B$, there is a unique homomorphism $\phi:G \rightarrow B$ for which the diagram



commutes (recall that η_0 is the injection of generators into nets). So if for example the generator c_i is given meaning $b_i = \eta(c_i)$ in B , then the net $(c_1 |^G c_2) [\beta/\gamma]^G$ is given, via ϕ , the unique meaning $(b_1 |^B b_2) [\beta/\gamma]^B$ in B .

Thus any algebra B of the same signature as G —that is, sorted on finite subsets of Λ and having operations $|$, $\backslash\alpha$, and $[S]$ at appropriate sorts—will be a suitable semantics for nets if we can show that it is a flow algebra, i.e. that it satisfies our laws, which we call the Laws of Flow. There are many flow algebras, but here we wish of course to show that processes (whose operations we have already given) constitute a flow algebra; this is the subject of Section 7.

7. Processes Are a Flow Algebra

The phyla P_L of the algebra P of processes were given in Section 3. We give an outline of the verification of the laws of flow, in which of course the superfix P should be understood on each operation symbol. The associativity of composition (C2) is the only nontrivial proof; a detailed case analysis is required, and may be found in full in the technical report [11], but here we shall just enumerate the cases and treat one of them in detail.

LAW (C1). $p_1 | p_2 = p_2 | p_1$.

Define $|^R$ by $p |^R p' = p' | p$. We require that $|^R = |$. Now by routine manipulation we can show that $|^R$ satisfies the defining equation for $|$, whence $| \subseteq |^R$. It follows that $p_1 | p_2 \subseteq p_2 | p_1$, and by symmetry (C1) is established.

For the next law we need two simple lemmas (proofs omitted).

LEMMA 1 (strictness of $|$). $\perp | p = p | \perp = \perp$.

LEMMA 2. $p_1 | p_2 = \perp \Rightarrow p_1 | (p_2 | p_3) = \perp$.

In what follows, we use the notation $|_i$ for the i th truncation of the function $|$. Thus if Ψ is the functional (given implicitly by eq. (4) in Section 3) whose least fixed point is $|$, then $|_0 = \perp$, $|_{i+1} = \Psi(|_i)$, and $| = \bigcup_i |_{i+1}$.

LAW (C2). $p | (q | r) = (p | q) | r$.

We show that

$$p | (q | r) \subseteq (p | q) | r \quad (1)$$

and

$$p | (q | r) \supseteq (p | q) | r. \quad (2)$$

To prove (1), show $\forall i, j, p, q, r$,

$$p |_{i+1} (q |_{j+1} r) \subseteq (p |_{i+1} q) |_{j+1} r \quad (3)$$

by induction on the sum of i and j . Then (2) follows from (1) by Law (C1).

Base. We prove (3) when $i = 0$ or $j = 0$. In fact it follows immediately using $|_0 = \perp$ and Lemma 1.

Induction Step. To prove (3) it is enough to show that

$$m \in (p | q) | r \Rightarrow m \in p |_{i+1} (q |_{j+1} r) \quad (4)$$

when $i, j > 0$, under the inductive assumption

$$\forall p, q, r, p |_k (q |_h r) \subseteq (p | q) | r \text{ with } k + h < i + j. \quad (5)$$

That is, assume (5) and show (4) by cases.

Now m can arise as a member of $(p|q)|r$ in three ways:

Case 1. $m \in [\mu: \langle v, \lambda x.(gx|r) \rangle; \mu: \langle v, g \rangle \in p|q]$.

Case 2. $m \in [\mu: \langle v, \lambda x.((p|q)|gx) \rangle; \mu: \langle v, g \rangle \in r]$.

Case 3. $m \in gv'|g'v$, where $\mu: \langle v, g \rangle \in p|q$ and $\bar{\mu}: \langle v', g' \rangle \in r$.

Considering case 1, $\mu: \langle v, g \rangle \in p|q$ can arise also in three ways:

Case 1.1. $\exists \mu: \langle u, f \rangle \in p$ such that $\mu: \langle v, g \rangle \supseteq \mu: \langle u, \lambda x.(fx|q) \rangle$

Case 1.2. $\exists \mu: \langle u, f \rangle \in q$ such that $\mu: \langle v, g \rangle \supseteq \mu: \langle u, \lambda x.(p|fx) \rangle$.

Case 1.3. $\exists v: \langle u, f \rangle \in p, \bar{v}: \langle u', f' \rangle \in q$ such that $\mu: \langle v, g \rangle \in fu'|f'u$.

Now let us treat only the last subcase in detail. From $\bar{v}: \langle u', f' \rangle \in q$ we have

$$\bar{v}: \langle u', \lambda x.(f'x|_{j-1}r) \rangle \in q|_j r,$$

whence, from $v: \langle u, f \rangle \in p$, it follows that

$$fu'|_{i-1}(f'u|_{j-1}r) \subseteq p|(q|_j r).$$

But by our induction hypothesis (5), and recalling that $\subseteq$ equals $\supseteq$, we know that

$$(fu'|f'u)|r \subseteq fu'|_{i-1}(f'u|_{j-1}r).$$

However, since $\mu: \langle v, g \rangle \in fu'|f'u$, we have

$$\mu: \langle v, \lambda x.(gx|r) \rangle \in (fu'|f'u)|r,$$

and we use the two preceding set inclusions, together with the assumption of case 1, to deduce that $m \in p|(q|_j r)$ as required. (The right closure of sets is also needed in this last deduction.)

We chose case 1.3 as an illustration of the proof method since it makes important use of the correspondence between the order relation $\sqsubseteq$ and set theoretic inclusion in the weak powerdomain.

LAW (R1). $p \setminus \alpha = p$ when $\alpha, \bar{\alpha} \notin L$, $p \in P_L$.

$p \setminus \alpha \sqsubseteq p$ may be proved by induction on the truncations $\setminus_i \alpha$. For the other half, we prove $p \setminus \alpha \supseteq \text{proc}_i(p)$ by induction, where $\text{proc} = \sqcup_i \text{proc}_i$ as defined in Section 3, eq. (2), is the identity function.

LAW (R2). $p \setminus \alpha \setminus \beta = p \setminus \beta \setminus \alpha$.

Induction on the truncations $\setminus_i \alpha$ or $\setminus_i \beta$ suffices.

LAW (R3). $(p_1|p_2) \setminus \alpha = (p_1 \setminus \alpha)|(p_2 \setminus \alpha)$ when $\alpha, \bar{\alpha} \in L_1 \cap L_2$, $p_1 \in P_{L_1}$, $p_2 \in P_{L_2}$.

Again, induction on the truncations $\setminus_i \alpha$ or $|_i$ suffices.

LAWS (S1)–(S4). All these laws may be proved by inductions similar to those for Laws (R1)–(R3).

8. Directions for Further Study

We have developed one flow algebra in some detail to support our belief that the concept of flow algebra, and in particular the algebra of processes, will be useful in the study of communication among computing agents and in the formation of a theory of such communications. Further investigation is needed to discover whether processes provide a tractable model of such agents as operating systems and computer networks, and to compare processes with other flow algebras in this and other respects. In particular we need to know what properties of concurrency are modeled more faithfully in one algebra than in another, and what proof techniques are available in each algebra.

We already know of the existence of other flow algebras; in one the objects are a kind of automaton, in another they are a kind of execution tree, and a third involves processes using the stronger form of powerdomain due to Plotkin. We must therefore examine the morphisms that exist among these algebras, particularly to determine which features of concurrency are preserved by the morphisms, and which are lost.

In our Introduction we suggested that the algebra of nets, i.e. flow algebra, could form a subset of future programming languages for concurrency. We would like to justify this

suggestion by designing a very simple programming language, including as little as possible beyond the flow operations, which will serve as a canonical programming language for concurrency—much in the same way that recursion equations, with a few primitive operations (e.g. the conditional function), may be regarded as canonical for sequential programming. At present it is not clear what is required beyond the flow operations; a satisfactory solution to this problem will, in our view, yield a strong justification for the present work.

ACKNOWLEDGMENTS. We would like to thank Gordon Plotkin and Mike Smyth for helpful discussions of this work. We are indebted to Dana Scott and the referees for their critical comments on an earlier version; we hope that as a result the presentation is clearer, and the work more clearly motivated, than it was previously. We are also grateful to Dorothy McKie for her patience in typing (and retyping!) a difficult manuscript.

REFERENCES

- 1 BEKIC, H Towards a mathematical theory of processes Tech Rep TR25.125, IBM Lab, Vienna, Dec 1971
- 2 BURSTALL, R M., AND LANDIN, P.J Programs and their proofs In Machine Intelligence, Vol 4, B Meltzer and D Michie, Eds., Edinburgh U Press, Edinburgh, Scotland, 1969, pp 17–43.
- 3 DIJKSTRA, E.W. The structure of the “THE”—multiprogramming system *Comm ACM* 11, 5 (May 1968), 341–346
- 4 GOGUEN, J A, THATCHER, J W, WAGNER, E.G, AND WRIGHT, J B Initial algebra semantics and continuous algebras. *J. ACM* 24, 1 (Jan 1977), 68–95
- 5 GREIF, I On proofs of programs for synchronisation Proc 3rd Int Colloquium Automata Languages and Programming, Edinburgh U Press, Edinburgh, Scotland, 1976, pp 494–507
- 6 HOARE, C A R An axiomatic basis for computer programming *Comm ACM* 12, 10 (Oct 1969), 576–580, 583.
- 7 HOARE, C A R Communicating sequential processes *Comm ACM* 21, 8 (Aug 1978), 666–677
- 8 KAHN, G., AND MACQUEEN, D Co-routines and networks of parallel processes. Res Rep 202, IRIA, Paris, 1976.
- 9 LUCAS, P, AND WALK, K On the formal description of PL/1 *Ann Rev. in Autom. Prog* 6, 3 (1969), 105–182.
- 10 MILNE, G A mathematical model of concurrent computation. Ph D Th, U. of Edinburgh, Edinburgh, Scotland, 1978.
- 11 MILNE, G, AND MILNER, R Concurrent processes and their syntax Rep CSR-2-77, Comptr Sci. Dept, U of Edinburgh, Edinburgh, Scotland, 1977
- 12 MILNER, R Processes, a mathematical model of computing agents Proc Logic Colloquium, Bristol, North-Holland Pub Co, Amsterdam, 1973, pp 157–174
- 13 MILNER, R Flowgraphs and flow algebras Rep CSR-5-77, Comptr Sci Dept, U of Edinburgh, Edinburgh, Scotland, 1977
- 14 MOSSES, P The mathematical semantics of ALGOL 60 Tech Mono DRG-12, Programming Res Group, Comptng Lab, Oxford U, Oxford, England, 1974.
- 15 OWICKI, S Axiomatic proof techniques for parallel programs TR 75-251, Cornell U, Ithaca, N Y, 1975
- 16 PDP11 Peripherals and Interfacing Handbook Digital Equipment Corp, Maynard, Mass, 1971
- 17 PLOTKIN, G. A powerdomain construction *SIAM J Comptng* 5, 3 (1976), 452–487
- 18 SCOTT, D The lattice of flow diagrams In Symposium on Semantics of Algorithmic Languages, E Engeler, Ed., Springer Lecture Note Series No 188, Springer-Verlag, Heidelberg, 1971, pp 311–366
- 19 SCOTT, D Lattice theoretic models for various type-free calculi Proc 4th Int Congress Logic, Methodology and Philosophy of Sci, Bucharest, Romania, 1972, North-Holland Pub Co, Amsterdam, 1973, pp 157–187
- 20 SMYTH, M Powerdomains Proc 5th Math. Foundations of Comptr Sci Symposium, Gdansk, Poland, Lecture Notes in Computer Science 45, Springer-Verlag, 1976, pp 537–543
- 21 TENNENT, R The denotational semantics of programming languages *Comm ACM* 19, 8 (Aug 1976), 437–453
- 22 VAN WIJNGAARDEN, A., ET AL Revised report on the algorithmic language ALGOL 68 *Acta Informatica* 5 (1975), 1–236
- 23 WIRTH, N MODULA A language for modular multiprogramming Res Rep 18, Inst für Informatik, Zurich, Switzerland, 1975
- 24 ZAVE, P On the formal definition of processes Proc Int Conf on Parallel Processing, 1976 (available from IEEE, New York)

RECEIVED JUNE 1977, REVISED MARCH 1978