DENOTATIONAL MODELS FOR PARALLEL PROGRAMS WITH INDETERMINATE OPERATORS

by

Robert M. Keller

UUCS - 77 - 103

To appear in Proceedings of the
IFIP Working Conference on
Formal Description of Programming Concepts,
August 1977.

This work was supported in part by the National Science Foundation through grant GJ-42627.

DENOTATIONAL MODELS FOR PARALLEL PROGRAMS WITH INDETERMINATE OPERATORS

Robert M. Keller
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Several approaches to networks of concurrently-operating modules involving indeterminacy are discussed. Techniques for representing the denotational semantics of such networks, and for verifying properties of them, are presented, including an oracle approach, an axiomatic approach, a data-type reduction approach, and a partial-ordering approach.

INTRODUCTION

The motivating context for this study is the desire to formally model and prove properties of systems, such as computer operating systems, with one or more of the following characteristics:

1. The system is composed of concurrently-operating sub-systems.
2. The system, or some of its sub-systems, may communicate with its environment via possibly-infinite input or output streams.
3. Sub-systems are modular, in the sense that they are specified in terms of operators on abstract data types (such as the streams mentioned above).
4. The system itself, or its subsystems, may be indeterminate, in the sense that several or many outputs may be possible in response to a single given input.

The motivation for points 1 to 3 is by now well-established. To motivate point 4, we note that certain real systems are indeed indeterminate. For example, the set of passengers assigned seats on a given flight by an airline reservation system is not purely a function of passengers requesting seats. Also, an operating system can be viewed as a program which calls user tasks with indeterminate results, namely the user's resource requests. Furthermore, some determinate systems may be constructed of indeterminate subsystems, as will be seen. One good reason for doing so is to increase the utilization of certain components.

We will be concerned with "denotational" modeling of such systems. By this term, we mean that operators are described by action on domains over the entire (possibly infinite) computation period, rather than a step at a time. The contrasting approach is termed "operational" modeling, and is signalled by references to terms such as "states", "transitions", and "computation sequences."

This work was supported in part by the National Science Foundation through grant GJ-42627.

Current proof methodologies for concurrent programs, e.g. [Keller 76], [Lamport 77], [van Lamsweerde and Sintzoff 76] tend to deal with what we will call second order properties, such as invariants, deadlock-freedom, livelock-freedom, etc. The difficulty here is that one has to make a case that these properties really do constitute correctness. However, a first order definition of correctness should involve only a precisely-stated input/output relation which the system is expected to satisfy, and which hopefully can be proved. Of course, not too much is yet known about how to even state such relations for examples as complex as operating systems, but the denotational approach appears useful in that the definition of the program is closer to the domain in which the statement of the correctness relation is stated. The traditional approaches to partial or total correctness, e.g. [Floyd 67a], [Hoare 69], [Manna 69], etc. are not applicable because of the fact that the systems we wish to model are not generally terminating intentionally.

It is not our suggestion that one should try to describe every last detail of a system by the denotational approach. Although this can be done, there are certain very definite trade-offs between the succinctness and ease in proof of the two approaches. A complete system analysis will thus likely involve both. The investigation on which this paper is based attempts to see how far the denotational approach can be pushed. Reported here are some preliminary results and observations.

HISTORICAL

The continuous analog of the denotational approach has been in use by system engineers for a long time. That is, one deals with boxes, the behaviors of which are specified by "transfer functions." Boxes can be interconnected in series, parallel, with feedback loops, etc., and there is a calculus for dealing with such combinations, at least in the domain of Laplace or z- transforms [Zadeh and Desoer 63].

On the other hand, studies of concurrent systems of the type we have in mind have been relatively few. [Patil 70] (see also [Brinch Hansen 73]) discusses the interconnection of boxes representing functions on sequence domains. In particular, it was shown that an interconnection of such boxes yields a system, the behavior of which could also be represented as a function, i.e. that determinate systems are closed under composition. Hence a network of such boxes can be analyzed in a step-wise fashion. A similar proof for general systems was also given in [Zadeh and Desoer 63].

From an operational standpoint, similar results also appeared in [Karp and Miller 66], [Adams 68], [Rodriguez 68], [Keller 75]. In particular, [Patil 67] and [Adams 68] added the concept of recursion to such network models. Also relevant are [Patil 67] and [Seror 70] which added functional arguments. The use of such concepts for machine design has been additionally discussed in [Miller and Cocke 74] and [Dennis 74], for operating system designs in [Ritchie and Thompson 75] and [Stoy and Strachey 76], and for programming language features in [Conway 63], [Burge 75], and [Kahn and MacQueen 76].

It was the work reported in [Kahn 74] which was most inspirational in the present investigation. This work showed that networks of operators on sequence domains and the "closure properties" of the type described above can be represented elegantly in terms of a fixed point theory. Furthermore, Kahn indicated that recursion could also be included without great difficulty. A related approach to dealing with networks appeared in [Böhm 66].

In the next section, we will re-present Kahn's result, except we will generalize it to domains other than sequence domains. The added usefulness of this generalization will be illustrated later in the paper.

NETWORKS OF OPERATORS ON DATA TYPES

Following [Vuillemin 73], and [Scott 76], a data type is a set D with a partial ordering $\sqsubseteq$ on D and a least element $\perp$ in D, such that for any chain of elements in D,

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$$

there is a unique least upper bound, denoted

$$\sqcup\{d_0, d_1, d_2, \ldots\}$$

If $\hat{d}$ denotes the above element, then its defining property is that

$$(\forall i)\ d_i \sqsubseteq \hat{d}$$

and if d is any element such that

$$(\forall i)\ d_i \sqsubseteq d$$

then necessarily $\hat{d} \sqsubseteq d$.

Some examples of data types are the following:

PS: (Powerset data-type) Let S be a set, and P(S) be the set of all subsets of S. Then D is P(S), $\sqsubseteq$ is $\subseteq$ and $\perp$ is $\emptyset$, the empty set.

FD: (Flat data-type) Let S be some set, with "?" (read "undefined") an element not in S. Then D is S U {?}, $\perp$ is ?, and $\sqsubseteq$ is defined by

$$x \sqsubseteq y \quad \text{iff} \quad x = ? \text{ or } x = y$$

SD: (Sequence domain data-type) Let S be some set. Let $\hat{S}$ denote the set of all finite or countably-infinite sequences of elements in S, including the null sequence, Λ. Then D is $\hat{S}$, $\perp$ is Λ, and $\sqsubseteq$ is $\leq$ ("is a prefix of"), i.e.

$$x \leq y \quad \text{iff} \quad x = y \text{ or } (\exists u)\ xu = y$$

where juxtaposition denotes concatenation.

BD: (Bag-domain data-type) Let S be some set. Let D = B(S) be the set of all bags over S. A bag is like a set, except that repeated elements are allowed, with any countable number of repetitions. Equivalently, a bag is a map $B: S \to \omega \cup \{\infty\}$, where ω is the set of all natural numbers and ∞ is a special infinity symbol. The element $\perp$ is the function $\emptyset$ such that $\emptyset(x) = 0$ for all $x \in S$. If

$$b_0 \sqsubseteq b_1 \sqsubseteq b_2 \sqsubseteq \ldots$$

is a chain of bags, then

$$\sqcup\{b_0, b_1, b_2, \ldots\} = b$$

where $(\forall x \in S)$ b(x) is the numeric least upper bound of $\{b_0(x), b_1(x), b_2(x), \ldots\}$.

PO: (Partial Order data-type) Let S be some set. Let D = Π(S) be the set of partial orders over S. Then $\sqsubseteq$ is $\subseteq$ and $\perp$ is I, the identity relation.

TD: (Tree-domain data-type) Let S be some set. Let $\tilde{S}$ denote the set of all finite or countably infinite "trees" of elements in S, including the null tree, Λ. That is, all elements of S are trees, and any finite or countably-infinite sequence $\langle t_0, t_1, t_2, \ldots \rangle$ of trees $t_0, t_1, t_2, \ldots$ is a tree, where we include the null sequence Λ as well. Then, of course, D = $\tilde{S}$ and $\perp$ is Λ. The ordering

$\sqsubseteq$ is given by

(i) $(\forall x \varepsilon \tilde{S})\ \Lambda \sqsubseteq x$

(ii) If $x \varepsilon S$, then $x \sqsubseteq y$ iff $x = y$.

(iii) If $x_0, x_1, \ldots, y_0, y_1, \ldots,$ are trees and $(\forall i)\ x_i \sqsubseteq y_i$, then
$\langle x_1, x_2, \ldots \rangle \sqsubseteq \langle y_1, y_2, \ldots \rangle$

For example, if S = {a}, then a tiny segment of the ordering is shown in Figure 1, where any Λ can be replaced by an a to get another element $\sqsupset$ the one shown.

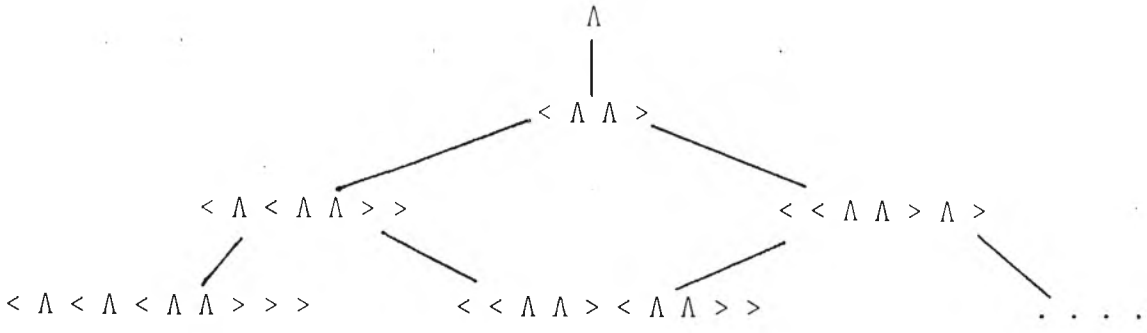


Figure 1 A segment of the tree domain.

It is also possible to construct data types from other data types. For example, if $(D_i, \perp_i, \sqsubseteq_i)$, i = 1, 2, ..., n are data types, then so is $(\underset{i=1}{\overset{n}{X}} D_i, \perp, \sqsubseteq)$ where $\perp = (\perp_1, \perp_2, \ldots, \perp_n)$ and $\sqsubseteq$ is given by

$$(x_1, x_2, \ldots, x_n) \sqsubseteq (y_1, y_2, \ldots, y_n)$$

iff

$$(\forall i)\ \ x_i \sqsubseteq y_i$$

Data types can be constructed from "continuous functions" on data types as well. A function from one data type into another,

$$f: \ D_1 \to D_2$$

is <u>continuous</u> if for any chain in D_1

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$$

we also have a chain in D_2 (this is called the <u>monotonicity</u> condition)

$$f(d_0) \sqsubseteq f(d_1) \sqsubseteq f(d_2) \sqsubseteq \ldots$$

and furthermore

$$f(\sqcup_1\{d_0, d_1, d_2, \ldots\}) = \sqcup_2\{f(d_0), f(d_1), f(d_2), \ldots\}$$

For example, in the case of sequence domains, the continuous functions are those functions which are monotonic and whose values on infinite arguments can be <u>defined</u> from values on finite arguments by taking limits. That is, extending

the input argument of a function cannot diminish the output, and any finite segment of output must be produced by some finite input. One important consequence of the continuity of a function is that its properties can be proved from its properties when restricted to finite inputs, thereby admitting inductive proofs. See [Kahn 74] for several examples.

It can be shown that the set of all continuous functions from D_1 into D_2, denoted $[D_1 \rightarrow D_2]$, is a data type, where the ordering is

$$f \sqsubseteq g \quad \text{iff} \quad (\forall d \in D_1) \quad f(d) \sqsubseteq g(d)$$

To see this, we need only observe that

$$f = \sqcup\{f_0, f_1, f_2, \ldots\}$$

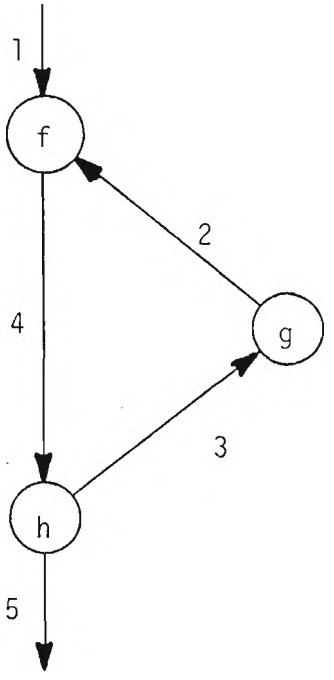
is determined by

$$(\forall d \in D_1) \quad f(d) = \sqcup_2\{f_0(d), f_1(d), f_2(d), \ldots\}$$

The preceding discussion gives a way of computing a semantic function for any "network" of operators on data types. By a network, we mean a directed graph, the arcs of which correspond to data types and the nodes of which correspond to operators. A node with several arcs, say $x_1, x_2, \ldots, x_n$, directed into it is viewed as a set of functions

$$f_j: \quad D_1 \times D_2 \times \ldots \times D_n \rightarrow D_j$$

where there is one such function for each arc directed out of the node, with D_j being the domain associated with that arc. An example is shown in Figure 2.



$f: D_1 \times D_2 \rightarrow D_4$

$g: D_3 \rightarrow D_2$

$h_1: D_4 \rightarrow D_5$

$h_2: D_4 \rightarrow D_3$

Figure 2 Example of a network of operators

A function for a network can be determined from the functions of its constituent operators. Namely, the network's function is a map from the domains of the unconnected arcs directed into the network to the domains of the unconnected arcs directed out of the network. Let

$$D = D_1 \times D_2 \times \ldots \times D_n$$

be the product of all domains. Then the network function is determined as follows: let $d \in D$ be such that the input arcs have the values to which the network is to be applied, and let values of other arcs in d be $\perp$ for their respective domains. We let

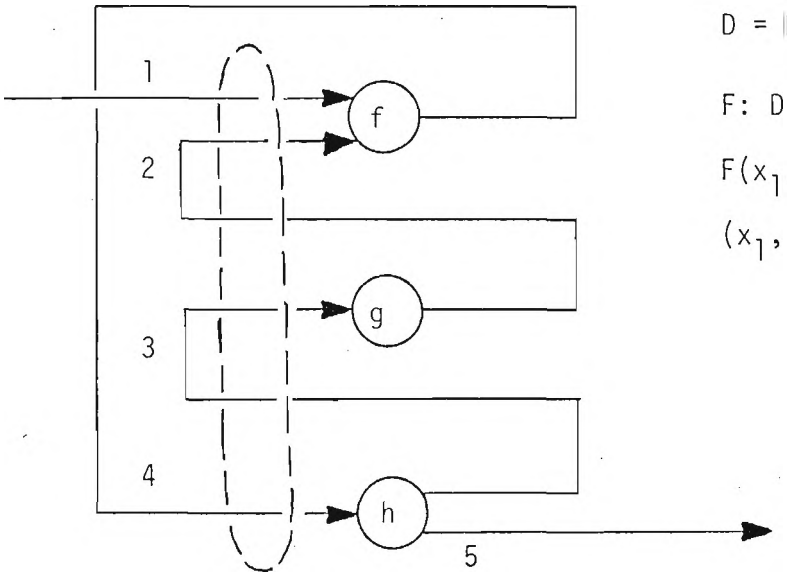
$$F: \quad D \to D$$

be the function obtained by "bundling" the functions for the constituent nodes together in the obvious way (see Figure 3 for an example). Then the output values for the network produced by the given inputs are obtained as the output components of

$$\bigsqcup \{d, F(d), F^2(d), \ldots\}$$

the latter being given the more convenient notation

$$F^*(d)$$



$D = D_1 \times D_2 \times D_3 \times D_4 \times D_5$

$F: D \to D$ is defined by

$F(x_1, x_2, x_3, x_4, x_5) =$

$(x_1, g(x_3), h_2(x_4), f(x_1, x_2), h_1(x_4))$

Figure 3 Bundling

An example is shown in Figure 4.

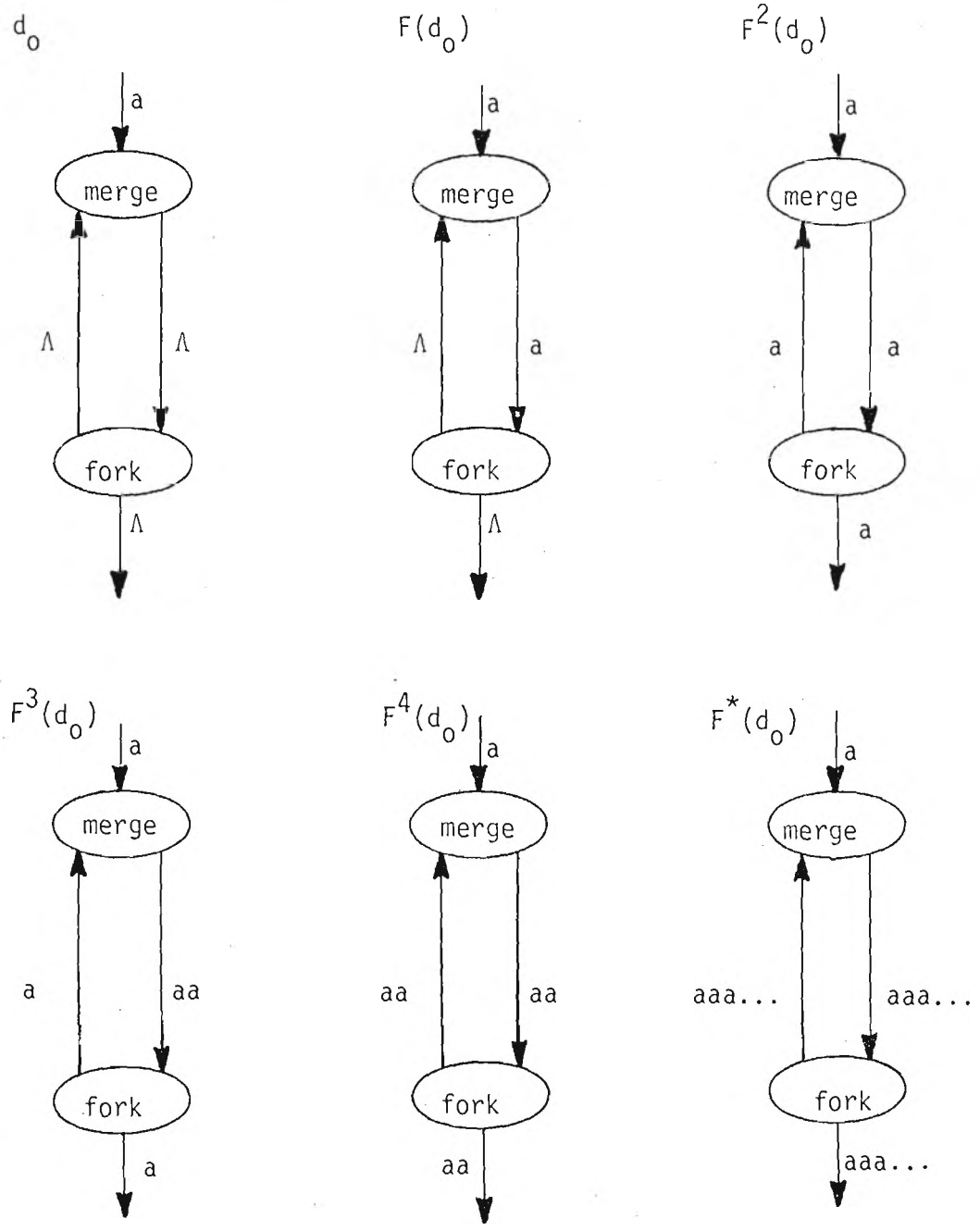


Figure 4 Computation of $F^*(d_0)$, where all domains are $\{\hat{a}\}$.

The module definitions for this case are merge(x, y) = xy and $fork_1(x) = fork_2(x) = x$.

As follows from analogy with Kleene's first recursion theorem [Kleene 52], $F^*(d)$ is the least solution (i.e. least fixed point) d^* of the equation

$$d^* = d \sqcup F(d^*)$$

since

$$\begin{aligned} d \sqcup F(F^*(d)) &= d \sqcup F(\bigsqcup\{d, F(d), F^2(d), \ldots\}) \\ &= d \sqcup \bigsqcup\{F(d), F^2(d), F^3(d), \ldots\} \text{(since F is continuous)} \\ &= \bigsqcup\{d, F(d), F^2(d), F^3(d), \ldots\} = F^*(d) \end{aligned}$$

(the above establishing that $F^*(d)$ is a solution) and, for any solution d^*

$d^* \sqsupseteq d$ (from the equation)

$d^* \sqsupseteq F(d^*) \sqsupseteq F(d)$ (from the equation, the above, and monotonicity of F)

$d^* \sqsupseteq F(d^*) \sqsupseteq F(F(d))$ (from the above, and monotonicity of F)

$\vdots$

$d^* \sqsupseteq F(d^*) \sqsupseteq F^i(d)$ for each i

Therefore

$$d^* \sqsupseteq F^*(d)$$

which establishes that $F^*(d)$ is the least solution.

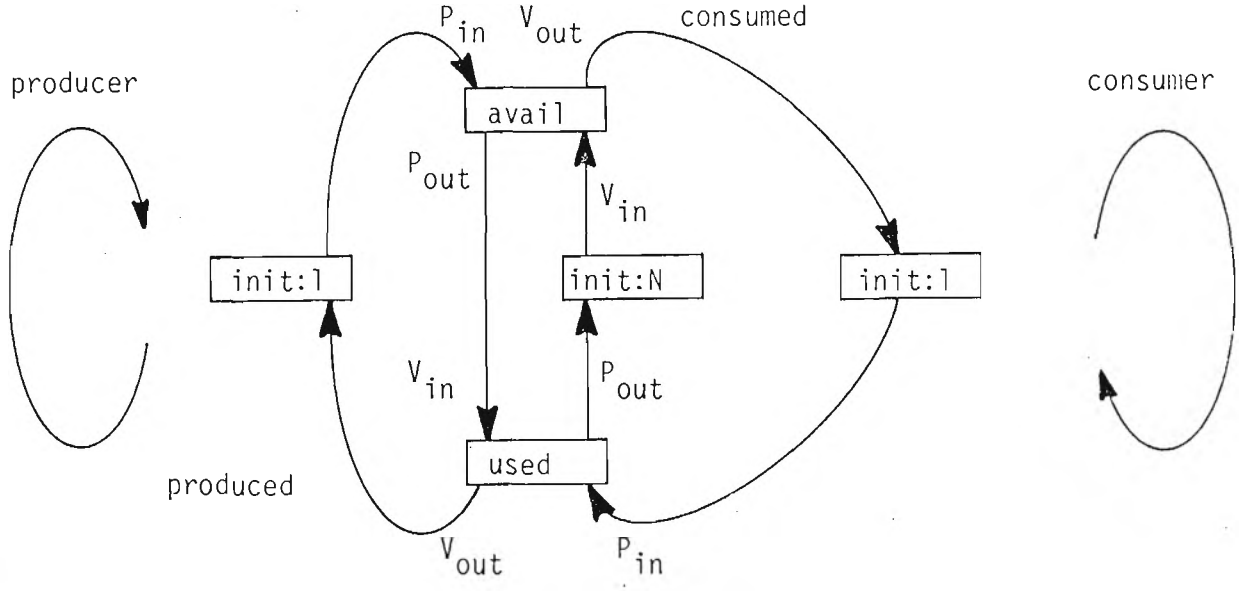


Figure 5

Semantics for unshared semaphores avail and used: $V_{out} = V_{in}$, $P_{out} = \min(P_{in}, V_{in})$

Semantics for initializing modules init:n: init:n(x) = n

All domains are the natural numbers plus ∞ and all arcs are initially 0.

The least solution to the system gives

$$(\text{produced}, \text{consumed}) = \begin{cases} (0, 0) & \text{if } N = 0 \\ (\infty, \infty) & \text{if } N > 0 \end{cases}$$

The least solution viewpoint is demonstrated in Figure 5 on an abstract system involving two unshared semaphores [Dijkstra 68]. The conclusion to derived from the least solution is that for certain initial values, the system is completely "deadlocked," whereas for others, it never becomes deadlocked, both producer and consumer running forever.

The preceding result has a special interpretation. It says that a network of continuous functions is a function, in the sense that there is a unique least (with respect to the data type ordering) output for any given input. In other-words, a network of continuous functions, if implemented in a manner which guarantees that the "least defined" output is actually produced, is "determinate." It can be further shown without difficulty that the network function is itself continuous, which gives the following important property:

The class of networks of continuous functions

is closed under arbitrary network composition.

Similar reasoning allows one to obtain the function for a network with recursion, i.e. one in which a node is given a name which is the name of the network itself. In this instance, we can view the network as a functional, in the sense that the name which is used recursively is treated as a function variable. Thus, G is the network functional, such that for any f

$$G(f)$$

represents the network with function f substituted for the node in question. If we let $\perp\!\!\perp$ denote the constant function whose value is always $\perp$, then

$$G^*(\perp\!\!\perp) = \bigsqcup\{\perp\!\!\perp, G(\perp\!\!\perp), G^2(\perp\!\!\perp), \ldots\}$$

can be taken as the function computed by the network, and similar to the previous reasoning, $G^*(\perp\!\!\perp)$ is the least function Φ which satisfies

$$\Phi(d) = G(\Phi(d))$$

An example is shown in Figure 6, where $G^*(\perp\!\!\perp)$ is seen to be a generally-infinite network obtained by repeated substitution of $G^i(\perp\!\!\perp)$ for f in G(f).

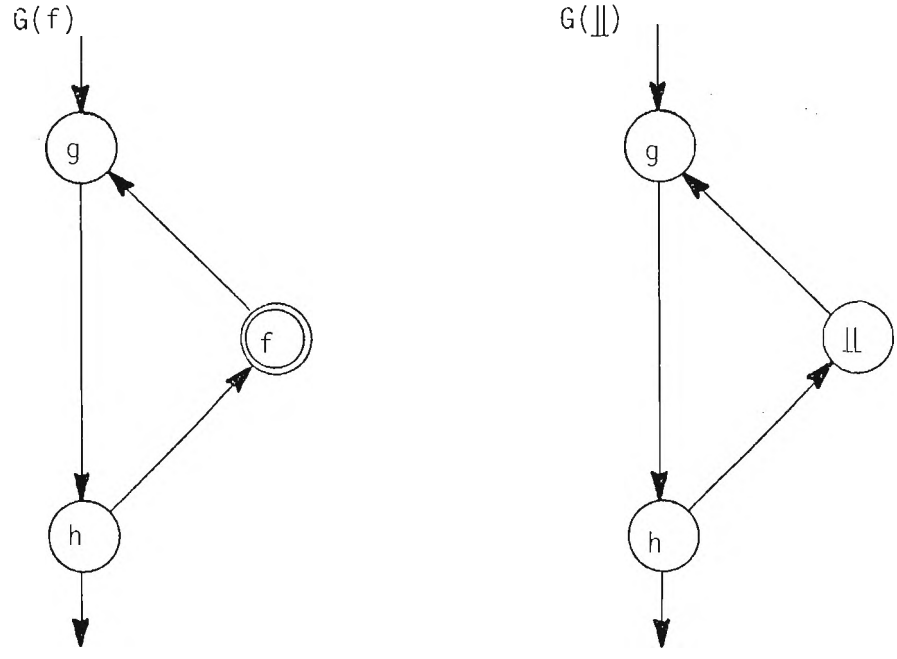


Figure 6 Construction of $G^*(\perp\!\!\perp)$ (continued on next page)

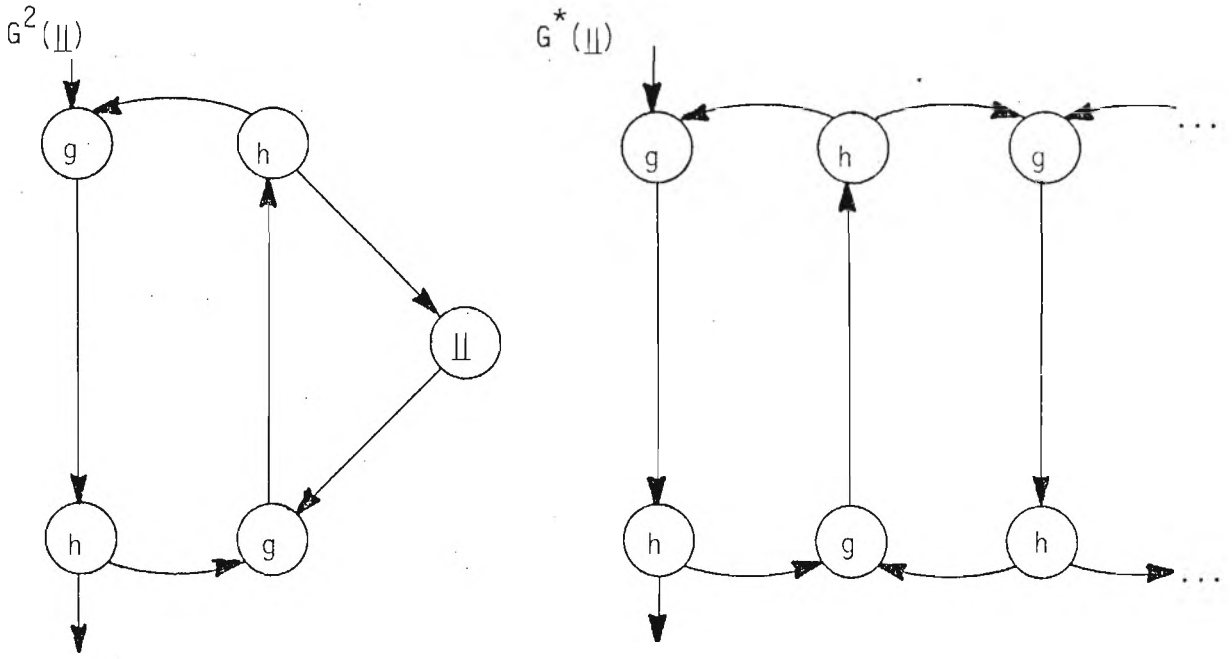
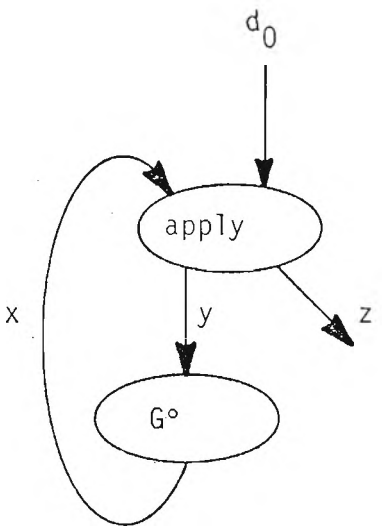


Figure 6 Construction of $G^*(\perp\!\!\perp)$ (continued)

To see that the same reasoning applies to networks with recursion, we can view the network G with recursion as the alternative network shown in Figure 7 (without recursion) where $(\forall\Phi)(\forall d)$ apply$(\Phi, d) = (\Phi, \Phi(d))$ and $(\forall\Phi)$ $G\circ(\Phi)$ is the function such that $(\forall d)$ $(G\circ(\Phi))(d) = G(\Phi(d))$. We summarize by stating:

The class of networks of continuous functions is closed under recursion.



x	y	z
$\perp\!\!\perp$	$\perp\!\!\perp$	$\perp$
$G(\perp\!\!\perp)$	$\perp\!\!\perp$	$\perp\!\!\perp(z)=\perp$
$G(\perp\!\!\perp)$	$G(\perp\!\!\perp)$	$G(\perp\!\!\perp)(d_0)$
$G(G(\perp\!\!\perp))$	$G(\perp\!\!\perp)$	$G(\perp\!\!\perp)(d_0)$
$G(G(\perp\!\!\perp))$	$G(G(\perp\!\!\perp))$	$G(G(\perp\!\!\perp))(d_0)$
$G(G(G(\perp\!\!\perp)))$	$G(G(\perp\!\!\perp))$	$G(G(\perp\!\!\perp))(d_0)$
⋮		

Figure 7 Functional network without recursion

The results stated in this section suggest that in a system constructed of continuous functions, computation may be performed incrementally. That is, if the ultimate input to the system function F is y, but we currently know only x, where $x \sqsubseteq y$, then we may begin computing F(x), without fearing that the computational effort will be wasted, since it is guaranteed that $F(x) \sqsubseteq F(y)$. As pointed out in [Patil 70] and [Kahn 74], this form of computation is particularly relevant for sequence domains.

NETWORKS OVER SEQUENCE DOMAINS

Sequence domains are an important class of data types. For example, many operating system functions can be viewed as mappings on sequence domains, as many of them involve buffering processes between producers and consumers. One can envision a network composed of intercommunicating modules, with each module containing some local storage (even an unbounded amount thereof) and executing an internal sequential program. The sequential program can perform any operations or tests on its internal variables, but can communicate with other modules only via certain commands which involve the links (arcs) interconnecting the modules. We classify the commands performable by a sequential module program as follows:

Types of Commands:

1. Internal command - These are commands which operate only on variables internal to a module. Any standard, sequential, instructions are acceptable.
2. Write command - A write command transmits a symbol from an internal variable out through a specified output link. The notation is

   write <variable name> on <link>

3. Read command - A read command waits for a symbol to appear on a particular input link, then copies the symbol into an internal variable. The notation is

   read <variable name> from <link name>

4. Poll command - A poll command checks to see if there is currently an un-read symbol on a specified input link. The command returns a value true if there is, and false otherwise. Thus, the notation is as if a Boolean procedure were being called:

   non-empty <link name>

5. Choice command - Permits a non-deterministic control branch to occur. The notation is as if a Boolean procedure were being called:

   choice

6. Extended read command - Like the read command, except that if there is currently no symbol on the link, then the symbol read is taken to be a special indicator "?".

We mention 6 above only to relate it to other schemes. Since allowing 3 and 4 is equivalent to allowing 3 and 6, we will use 4 in lieu of 6.

Previous work of Kahn [Kahn 74] dealt only with modules whose programs computed continuous functions from input sequence domains to output sequence domains. Kahn observed that restricting commands to be internal, read, or write insured this property. In this paper, we wish to examine the extension to poll and choice commands as well. Such modules can provide an alternative to "monitors" [Hoare 74] in which, rather than entering a module themselves, processes effectively transfer control to an internal process which can be dormant inside the module until a request occurs. Many operating system supervisors work in this way. One advantage over Hoare's monitors is that the scheduling rules can be tailored by the designer, rather than being built into a language construct. The use of such modules in achieving secure systems is also known [Lampson 69]. This approach is also used in some of the modules of [Keller 74].

As will be seen, inclusion of either poll or choice introduces the possibility that the semantics of a module be indeterminate. As we shall also see, different types of indeterminacy result when only poll or only choice commands are used alone.

Using 1, 2, 3, and 4 for example, we can implement a merge module (Figure 8a). This module has two input links, x and y, and an output link z. One possible internal program for it is

```
loop
    if non-empty(x)
        then
            read σ from x;
            write σ on z
            fi
    if non-empty(y)
        then
            read σ from y;
            write σ on z   fi
end
```

We will discuss the denotational semantics for this module later. To give a better idea of what it does, we state an axiom for it.

merge axiom: $z \in x \Delta y$

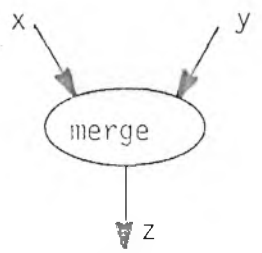
Here x and y are finite or infinite strings and $x \Delta y$ (read "x shuffle y") is a set of strings, where Δ is defined recursively:

$x \Delta y =$ if $(x = \Lambda$ or $y = \Lambda)$
then
$\{xy\}$
else
letting $x = \sigma u$, $y = \tau v$ $(\sigma, \tau \in \Sigma)$
$\{\sigma\}(u \Delta y) \cup \{\tau\}(x \Delta v)$
fi

For example,

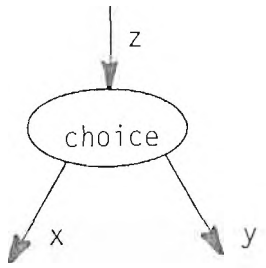
$ab \Delta cd = \{abcd, acbd, acdb, cabd, cadb, cdab\}$

(a)



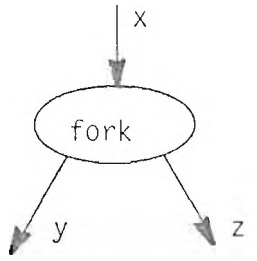
$z \in x \Delta y$

(b)



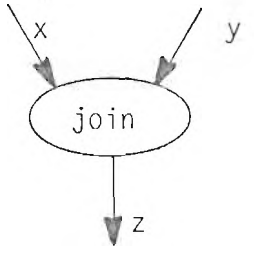
$z \in x \Delta y$

(c)



$x = y = z$

(d)



$z \varepsilon \min(\ln(x), \ln(y))$

Figure 8 Graphical representations and axioms for some modules

Merge is indeterminate, in the sense that for a given pair of input sequences, there may be any one of a set of several or many output sequences. An example of the use of the merge module appears in Figure 9. Here all domains except the output are sequences of trees over a set S. The output is a sequence of

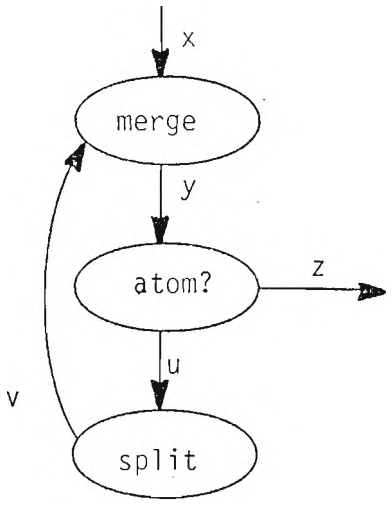


Figure 9 A network which counts the leaves in a sequence of trees

elements of S. The system shown is supposed to output all leaves of trees in the sequence. The module marked "atom?" splits its input stream into atoms, i.e. trees consisting only of a leaf, and non-atoms. The module marked "split" splits its input stream, which is assumed to be a sequence of non-atoms, into the sequence of sub-trees of those non-atoms. The example is similar to those in [Burstall 74] and [Manna and Waldinger 76], except that it is more succinct, and the proof, to be presented later, is straightforward, not involving any induction.

We now give an example of a module realizable with internal operations of type 1, 2, 3, and 5. This module (shown in Figure 8b) is, in a certain sense, the dual of the merge. It is called choice and has one input link, z, and two output links, x and y. A possible program is:

```
loop
    read σ from z
    if choice
        then
            write σ on x
        else
            write σ on y
        fi
    end
```

(The name "choice" is used both for the module and the internal command.)

An axiom for this module is given by the following:

choice axiom: $z \in x \Delta y$

The reader is justified in questioning whether the consideration of choice commands is at all worthwhile. We answer that programs with choice commands have been used in representing programs with backtracking, e.g. [Floyd 67a]. Choice commands are also useful when the choice is deterministic, but depends on some factor which we wish to leave variable, in the spirit of schema theory, for example, wherein one quantifies over all "interpretations."

The main contribution of this paper involves the examination of methods for dealing with indeterminate modules over sequence domains and related data types. We will consider three approaches:

1. Oracle approach: Representing indeterminacy by an oracle which selects one of a set of possible determinate behaviors.
2. Axiomatic approach: Expressing axioms for indeterminate modules, which may characterize important aspects of their behavior without necessarily being representationally complete.
3. Data-type reduction approach: By viewing certain indeterminate modules as operating on different data types which are "reductions" of the original, we can in some cases get determinate operators over the new data types.

Accompanying the discussion of these approaches will be some indication of the difficulty in trying to express precise semantics of indeterminate modules. An attempt at giving precise semantics for the merge is given in the Appendix.

THE ORACLE APPROACH

So far, the oracle approach has been successful in representing the semantics of general modules with choice commands, but not with general modules involving poll commands. Thus networks of determinate modules and choice modules can be represented. To the author, this indicates that the types of indeterminacy due to poll versus choice commands are fundamentally different.

In the oracle approach, the requirement that nodes of a network be continuous functions is relaxed to allow nodes which are relations of a special type, namely unions of continuous functions. Before computation begins, an "oracle" selects one function from the union for each node. By the arguments of the section on Networks of Operations on Data Types, each such selection guarantees a unique continuous function for the entire network. Furthermore, since a selection of one functon for each node can be thought of as a corresponding selection for the network, the network's semantics is also a union of continuous functions. So we have the following desirable closure property:

Networks in the oracle model
are closed under composition

Since recursion is essentially repeated composition, as shown before, we also have:

Networks in the oracle model

are closed under recursion.

Oracle approaches have also been discussed in different settings in [Milner 73], [Cadiou and Levy 73], and [Cohen 75]. The discussion in [Plotkin 76] may also be relevant.

To see why the oracle approach does not apply directly to modules which use polling commands, such as merge, we list the operation of merge on selected input sequences:

input	possible outputs
(Λ, Λ)	$\{\Lambda\}$
(a, c)	{ac, ca}
(ab, c)	{abc, acb, cab}

With the oracle approach, the three different outputs in response to (ab, c) would be produced by three different continuous functions. Now consider the continuous function f_1 such that $f_1(ab, c) = abc$. What is $f_1(a, c)$? Since f_1 is continuous, it must be a prefix of abc. It therefore must be a, because b is not in either input a or c. But this contrary to the definition of merge, e.g. the implementation given earlier would demand that $f_1(a, c) = ac$.

The obvious conclusion of the above is that prefix orderings are the "wrong" ordering. That is, we should be looking at a different data type. However, finding an appropriate data-type for merge is not trivial. For example, the sub-sequence ordering of strings does not work well. Although merge is monotonic with respect to the sub-sequence ordering, least upper bounds do not exist for arbitrary chains ordered by subsequence. Furthermore, it is unlikely that other modules in a system will be monotonic with respect to the subsequence ordering.

There turn out to be more obstacles to the application of the incremental approach to semantics of the merge. Of course, the denotational approach is only meaningful if its results agree with an operational approach. However, if we attempt to construct the output of the network shown in Figure 10, an anomaly appears. Ignoring problems of the monotonicity of merge, let us just consider that it is a function which produces for each set of pairs of inputs a set of outputs. Pursuing the incremental approach, initially we have

$$\{(x, y, z)\} = \{(ab, \Lambda, \Lambda)\}$$

At the next step, we would have

$$\{(ab, ab, \Lambda)\}$$

and at the next

$$\{(ab, ab, c)\}$$

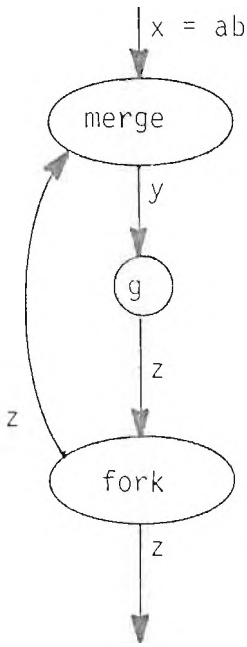
which implies that the output y of the merge could be any element of

$$ab \Delta c$$

Given that these inputs were presented externally, this would certainly be the case. However the element

$$cab$$

of this set cannot be an output operationally for the network of Figure 10,



$\Sigma = \{a, b, c\}$

Definition of g:

$g(\Lambda) = \Lambda$

$g(ay) = c$ for any $y \in \hat{\Sigma}$

$g(\sigma y) = \sigma$ for $\sigma \in \Sigma - \{a\}$, $y \in \hat{\Sigma}$

Figure 10 A merge anomaly

as the symbol c was produced, according to the definition of g, in response to the symbol a.

To summarize the above, we now see that the merge cannot be represented as any function of pairs of input sequences, even one which maps each pair into a set of possible outputs. Instead, there must be additional information in the input to a merge which indicates whether any symbol was produced in response to some other symbol, and therefore must be assimilated after it.

Our approach to giving a complete denotational semantics for the merge involves a much more elaborate data type which partially orders "events" represented by inputs to the merge. Because it is somewhat unwieldy and does not readily generalize to other modules with polling, we defer its presentation to the Appendix.

THE AXIOMATIC APPROACH

Although we have observed difficulties with representing complete semantics of modules such as merge, we can write axioms for such modules, as described in an earlier Section. It is then possible to derive axioms for networks using axioms of their constituent modules. We illustrate with two examples.

The first example, shown in Figure 11, is a simple control network for asynchronous modules, cf. [Keller 74]. As is typical with such networks, most "data processing" operations are abstracted out, leaving only the control portion.

In this case, we take as the property to be proved

$$z \in \ln(x)$$

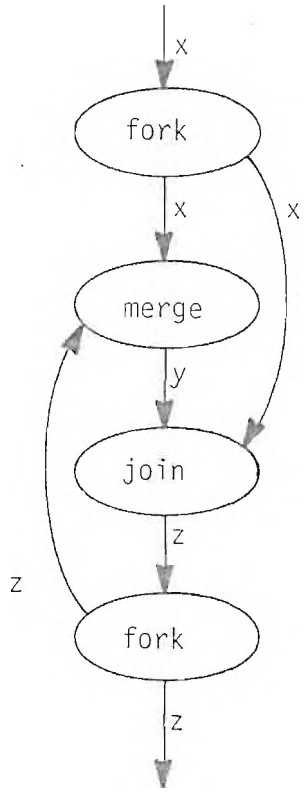


Figure 11 An asynchronous control network

That is, the output z is a string of symbols which is a member of the set of strings having the same length as the input string x. Interpreted in an operational sense, this means that the network does not "hang up" on any input, but instead produces output control signals in one-to-one correspondence with input signals (a "signal" being the occurrence of a symbol in a string).

We have taken a short cut and marked the inputs and outputs with each fork module with the same symbol, according to the axioms for fork given in Figure 8c. We then apply the axioms for merge and join:

$$y \in x \Delta z$$

$$z \in \min(\ln(x), \ln(y))$$

From these, infer

$$z \in \min(\ln(x), \ln(x \Delta z))$$

and since the length of x is $\leq$ that of any string in $x \Delta z$,

$$z \in \ln(x)$$

A corresponding operational proof of the same property will undoubtedly be more complicated.

For the second example, we again refer to the system of Figure 9. We assume that the trees are over a single-letter alphabet {a}. We wish to show that

$$z = \text{leaves}(x)$$

That is, z is a string of a's whose length is equal to the number of leaves in x. For simplicity, we identify the string itself and length of such a string, as if a string of n a's were the number n (n = ∞ is permissible).

We take as an axiom for any stream α:

0. leaves (α) = atoms (α) + leaves (non-atoms (α))

where an atom is a tree consisting of a single leaf by itself. We take as axioms for the modules:

1. z = atoms(y)
2. u = non-atoms(y)
3. y $\in$ v :: x
4. leaves(v) = leaves(u)

We then proceed to derive, without induction, the desired property of the network.

(from 1, 3) 5. z = atoms(v) + atoms(x)

(from 0, 5) 6. z = leaves(v) - leaves(non-atoms(v)) + atoms(x)

(from 4, 6) 7. z = leaves(u) - leaves(non-atoms(v)) + atoms(x)

(from 2, 7) 8. z = leaves(non-atoms(y)) - leaves(non-atoms(v)) + atoms(x)

(from 3, 8) 9. z = leaves(non-atoms(x)) + atoms(x)

(from 0, 9) 10. z = leaves(x)

It is interesting to note again that, although the concept of continuous function gives an inductive method for proving properties of such functions, in many cases induction is not necessary. Contrast this to the observation that induction is almost always necessary in operational proofs.

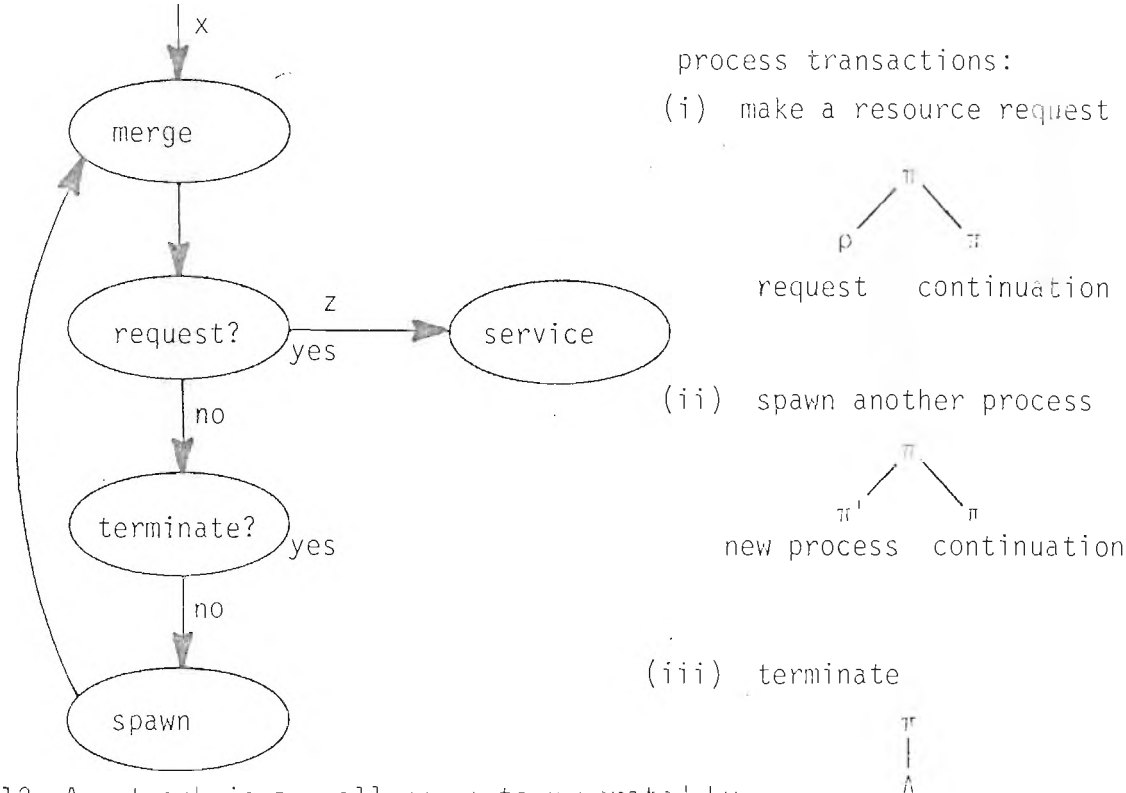


Figure 12 A network in z = all requests generated by processes in x and their descendants

An almost-isomorphic example to the previous one is shown in Figure 12. This example was suggested by a related example in [Manna and Waldinger 76]. The idea is that the network represents an "operating system" which processes a stream of requests where each request by a process is either

(i) a request for a resource

(ii) a request to spawn another process

(iii) a request to terminate this process

The idea is to show that each request ultimately gets answered.

THE DATA TYPE REDUCTION APPROACH

The idea of "data type reduction" is that we can sometimes ignore irrelevant aspects of a data type to end up with cleaner proofs. For example, if we ignore the ordering of symbols within a string, we have a bag. As shall be seen, certain operators which are indeterminate on one data type may become determinate on a reduction of that data type. One may consider this to be, in a very loose sense, a denotational analogue to the reduction described in [Lipton 75] and [Kwong 77] which applies to operational semantics.

Suppose that D_i, $\perp_i$, $\sqsubseteq_i$ $(i = 1, 2)$ are data types. We say that D_2 is a reduction of D_1 if there is an order-preserving homomorphism from D_1 onto D_2; i.e.

$$\eta: \ D_1 \to D_2$$

such that

$$(\forall x, y \in D_1)\ x \sqsubseteq_1 y \text{ implies } \eta(x) \sqsubseteq_2 \eta(y)$$

For example, if Σ is a set, then $B(\Sigma)$, the set of bags over Σ, is a data type reduction of $\hat{\Sigma}$, the sequence domain over Σ. If x is a string in $\hat{\Sigma}$, then $\eta(x)$ is a bag in which each symbol in Σ occurs the same number of times as it occurs in x.

A module which is a relation, but not necessarily a function, on its domains, say D_1, D_2, D_3, can sometimes be represented as a function on reductions of those domains. That is, if

$$f: \ D_1 \times D_2 \to P(D_3)$$

then there may be reductions η_1, η_2, η_3 and a function

$$g: \ \eta_1(D_1) \times \eta_2(D_2) \to \eta_3(D_3)$$

such that $(\forall (x, y, z) \in D_1 \times D_2 \times D_3)$

$$z \in f(x, y) \text{ implies } g(\eta_1(x), \eta_2(y)) = \eta_3(z)$$

For example, when we reduce the sequence domains of merge to the bag domain, merge becomes a function, namely bag addition. Similarly, the functions split and atom? can be represented as functions on the bag domain.

With the above reductions, the network of Figure 9 becomes a determinate network, and we can show that the output z is precisely the bag of all atoms in the input bag x of trees, even if the tree is over an alphabet with more than one symbol.

Of course, the approach described in this section will be useful only in those cases where we really can ignore the information discarded. This in turn suggests that one major difficulty in applying the denotational approach to examples such as the merge is that all relevant aspects of operational behavior must be represented in the chosen data types.

CONCLUSIONS

We have presented a discussion of networks of operators on data types and examined the extension of previously published work to a variety of data types and to indeterminate operators. Several approaches to proving correctness of systems involving these extensions were presented.

While those aspects of a system which are best handled by the network approach are still not well understood, the approach itself still demands attention. We point out that with the extensions introduced here, we can easily model standard sequential programs, our axiomatic approach thereby including the axiomatic approach of [Hoare 69]. In this case, the data type is a stream of state vectors. By using the merge, we can also represent arbitrary interleaving of processes, which in turn means that any operational model for concurrent programs can be represented. The work of [Mazurkiewicz 76] is relevant, in that it corresponds to partitioning state vectors into sub-vectors which can be operated on independently.

Networks also allow modeling of variety of recursive program rules, e.g. parallel if-then-else, etc. as described in [Manna and Vuillemin 72] (for Flat data types). In fact the networks readily display exploitable parallelism, showing promise for attacking problems discussed, e.g., in [Keller 73], [Urchsler 73]. It also appears that, by using our tree data type, a formal semantics for the "suspended" operations described in [Friedman and Wise 76] can be easily provided. Hence generality is to be gained, rather than lost, in further pursuit of such network models.

ACKNOWLEDGMENT

The author benefitted from discussing various aspects of this research with Jack Dennis, Gilles Kahn, Yat-Sang Kwong, Leslie Lamport, Zohar Manna, Suhas Patil, William Riddle, and John Smith. The careful typing assistance of Karen Evans is also appreciated.

APPENDIX - Denotational Semantics for the Merge

We describe here one approach to a denotational semantics for the merge. Part of the reason that this approach seems unwieldy is that it, and any other approach as well, must be capable of representing all relevant behavioral aspects in the chosen data type.

The data type chosen is the set of partially ordered events, a type similar, but not identical, to the PO data type in the main text. We know that the operation of a merge will involve the following events:

(1) Production of a symbol on an input to the merge.

(2) Assimilation of an input symbol by the merge.

(3) Production of an output symbol by the merge.

(4) Assimilation of the output symbol produced by the merge.

As far as a particular merge module itself is concerned, only (2) and (3) are relevant.

We use $\overline{\sigma}$ to denote the event corresponding to the production of symbol σ and $\underline{\sigma}$ the event corresponding to its assimilation. Of course, the "same" symbol may

be produced many times during the operation of a network, so we use a superscript, as in

$$\sigma^i \quad \bar{\sigma}^i \quad \underline{\sigma}^i$$

to distinguish to which copy of symbol σ we are referring.

Each symbol which is produced is produced on one of the arcs of the network, and assimilated by one of the modules to which that arc is input. So we write

$$e \text{ on } i$$

to designate the arc with which event e is associated. We now introduce the partial ordering relation

$$\longrightarrow$$

such that

$$e_1 \longrightarrow e_2$$

means that event e_1 must precede event e_2 in an operational sense, although in a denotational sense $\longrightarrow$ can be viewed simply as an abstract relation.

The first axiom about $\longrightarrow$ is one which holds for all arcs i,

A1: $\bar{\sigma}$ on i iff $\underline{\sigma}$ on i

In other words, the assimilation of a symbol must be associated with the same arc as its production.

The second axiom relates the fact that a symbol is assimilated only after it is produced,

A2: $\bar{\sigma} \longrightarrow \underline{\sigma}$

The third axiom states that the order of production of two symbols on an arc determines the order of their assimilation

A3: $(\bar{\sigma}_1$ on i) and $(\bar{\sigma}_2$ on i) and $(\bar{\sigma}_1 \longrightarrow \bar{\sigma}_2)$
imply $\underline{\sigma}_1 \longrightarrow \underline{\sigma}_2$

The fourth and fifth axioms are particular to the merge. The fourth says that for each input symbol assimilated, a new copy of that symbol is produced at the output, which is ordered with respect to $\longrightarrow$,

A4: ($\underline{\sigma}$ on input to merge) implies
$(\exists\bar{\sigma}')(\underline{\sigma} \longrightarrow \bar{\sigma}')$ and ($\bar{\sigma}'$ on output to merge)

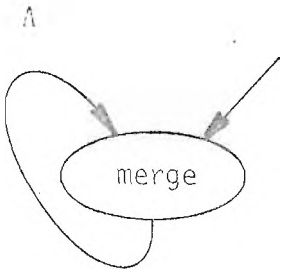
The fifth axiom says that if input symbols to a merge are ordered in $\longrightarrow$, then the corresponding output symbols are ordered in $\longrightarrow$,

A5: ($\underline{\sigma} \longrightarrow \underline{\tau}$ on input of merge) implies
($\bar{\sigma}' \longrightarrow \bar{\tau}'$on output of merge)

The merge is represented as a function on a set of partially-ordered events. Specifically, it maps one partial order x into another by adding the events representing the production of new symbols and their assimilation, corresponding to assimilation of symbols already in x, in accordance with the axioms stated.

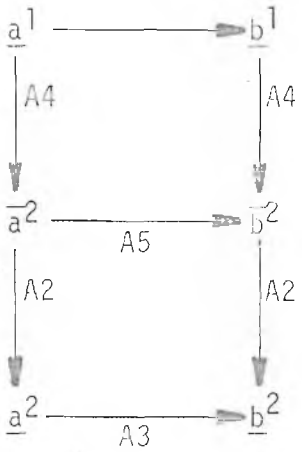
Figure 13 shows the repeated application of the merge mapping in an example related to that of Figure 10. The least fixed point of the mapping is also shown.

The corresponding behavioral output is roughly those sequences of symbols consistent with the partial ordering in the least fixed point. We say "roughly"



d_0: $\underline{a}^1 \longrightarrow \underline{b}^1$ (assimilation of a^1 precedes that of b^1)

$F(d_0)$



(labels on arcs denote axioms which assert arcs' existence)

$F^*(d_0)$:

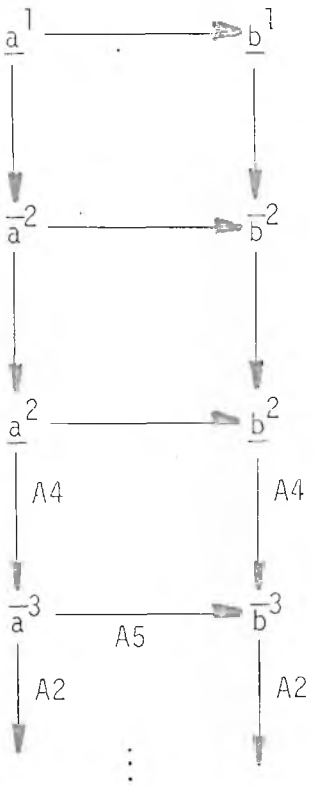


Figure 13 Part of partial order mapping for merge

$F^*(d_0)$:

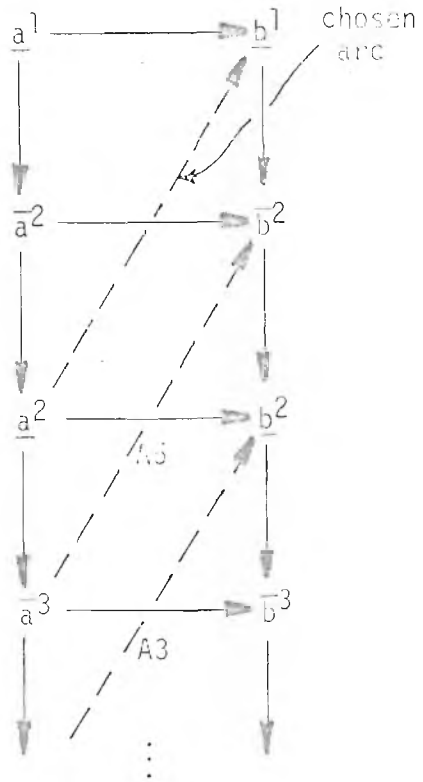


Figure 14 Additional arcs implied by choice of output ordering

because further qualification is necessary. The set mentioned for Figure 13 would be

$$ab \Delta ab \Delta ab \Delta \ldots$$

which is just

$$\{x \in \{a, b\}^{\omega} \mid \#_a(x) = \#_b(x) \text{ and } (\forall y < x)\ \#_a(y) \geq \#_b(y)\}$$

Here we are using $\#_a(x)$ to represent the number of a's which occur in x. However certain sequences in the set cannot be outputs of the merge, for example:

$$a\ a\ b\ a\ a\ b\ a\ a\ b \ldots$$

This can be verified by simulating the network operationally.

The further qualification needed is indicated by our use of the phrase "sequence of symbols consistent with." The term "sequence" implies a linear ordering $\Longrightarrow$. When we choose arbitrarily to include a pair of events in $\Longrightarrow$ which are not in $\longrightarrow$, we may have to include other pairs as well, to insure that axioms A1 - A5 are valid. For example, the arbitrary choice to include

$$\underline{a}^2 \longrightarrow \underline{b}^1$$

would imply

$$\begin{aligned}
\bar{a}^3 &\longrightarrow \bar{b}^2 && \text{(by A5)}\\
\underline{a}^3 &\longrightarrow \underline{b}^2 && \text{(by A3)}\\
\bar{a}^4 &\longrightarrow \bar{b}^3 && \text{(by A5)}\\
\underline{a}^4 &\longrightarrow \underline{b}^3 && \text{(by A3)}\\
&\vdots
\end{aligned}$$

(see Figure 14) so that the only output sequence which begins

$$a\ a\ b \ldots$$

is

$$a\ a\ b\ a\ b\ a\ b\ a\ b \ldots$$

We can now appeal again to the oracle approach described in the text to state that the output of a merge is any linear sequence consistent with axioms A1 to A5, which for this example is observed to be

$$\{aa^n b(ab)^{\omega} \mid n \in \omega\}$$

To summarize, the denotational semantics for networks involving the merge is specified in terms of mappings on partially-ordered events. When other types of modules are involved as well, we can either try to get partial order semantics for them, or we can linearize the partial order and appeal to the oracle approach. The best way to integrate this type of semantics into such a network is left as an open question.

We wish to note the similarity of the event ordering approach to the work in [Grief 75], which axiomatizes properties of "cells" for example. It is easy to see that a cell can be implemented using a merge and determinate modules (but cannot be implemented only with determinate modules, as cells are not functions on streams). Our approach is slightly more structured than Grief's in terms of the way in which messages are sent between modules. The exact relationship between Grief's approach and ours remains for future investigation. [Lamport 76] also discusses the use of partial orderings in distributed concurrent systems.

We also note that Riddle's concept of "synchronization symbols" [Riddle 76] is essentially a way of representing partial orderings in a linear string notation. The use of this concept for representing merge semantics therefore merits further investigation.

REFERENCES

[Adams 68] D.A. Adams, A computation model with data flow sequencing. Stanford University, Computer Science Dept., Tech. Rept. CS117 (1968).

[Brinch Hansen 73] P. Brinch Hansen, Operating system principles. Prentice-Hall (1973).

[Böhm 66] C. Böhm, The CUCH as a formal description language, in T.B. Steel (ed.) Formal language description languages, North-Holland (1966).

[Burge 75] W.H. Burge, Recursive programming techniques. Addison-Wesley (1975).

[Burstall 74] R.M. Burstall, Program proving as hand simulation with a little induction. IFIP '74, 308-312.

[Cadiou and Levy 73] J.M. Cadiou and J.J. Levy, Mechanizable proofs about parallel processes. IEEE Fourteenth Switching and Automata Theory Symposium, 34-48 (Oct. 1973).

[Cohen 75] E.S. Cohen, A semantic model for parallel systems with scheduling. Second ACM Symposium on Principles of Programming Languages, 87-94 (Jan. 1975).

[Conway 63] M.E. Conway, Design of a separable transition-diagram compiler. CACM, 6, 396-408 (1963).

[Dennis 74] J.B. Dennis and D.P. Misunas, A preliminary architecture for a basic data-flow processor. MIT Project MAC Computation Structures Group Memo. 102 (August 1974).

[Dijkstra 68] E.W. Dijkstra, Cooperating sequential processes, in F. Genuys (ed.), Programming languages, Academic Press (1968).

[Floyd 67a] R.W. Floyd, Non-deterministic algorithms. JACM 14, 4, 636-644 (Oct. 1967).

[Floyd 67b] R.W. Floyd, Assigning meanings to programs. Proc. Symp. in Appl. Math., 19, 19-32, AMS (1967).

[Friedman and Wise 76] D.P. Friedman and D.S. Wise, The impact of applicative programming on multiprocessing. Tech. Rept. 52, Computer Science Dept., Indiana University (July 1976).

[Ginsburg 66] S. Ginsburg, The mathematical theory of context-free languages. McGraw-Hill (1966).

[Greif 75] I. Greif, Semantics of communicating parallel processes. MIT Project MAC TR-154 (Sept. 1975).

[Hoare 69] C.A.R. Hoare, An axiomatic basis for computer programming. C. ACM, 12, 10, 576-580, 583 (Oct. 1969).

[Hoare 74] C.A.R. Hoare, Monitors: an operating system structuring concept. CACM, 17, 10, 54-557 (Oct. 1974).

[Kahn 74] G. Kahn, The semantics of a simple language for parallel programming. Proc. IFIP '74, 471-475 (1974).

[Kahn and MacQueen 76] G. Kahn and D. MacQueen, Coroutines and networks of parallel processes. IRIA Rept. 202 (Nov. 1976).

[Karp and Miller 66] R.M. Karp and R.E. Miller, Properties of a model for parallel computations: Determinacy, termination, queueing. SIAM J. Appl. Math, 14, 6, 1390-1411 (Nov. 1966).

[Karp and Miller 68] R.M. Karp and R.E. Miller, Parallel program schemata. J. Comp. and Syst. Sci., 3, 2, 147-195 (May 1969).

[Keller 72] R.M. Keller, On the decomposition of asynchronous systems. Proc. IEEE Thirteenth Annual Symposium on Switching and Automata Theory, 78-89 (Oct. 1972).

[Keller 73] R.M. Keller, Parallel program schemata and maximal parallelism. J. ACM, 20, 3, 514-537 (July 1973) and 20, 4, 696-710 (Oct. 1973).

[Keller 74] R.M. Keller, Towards a theory of universal speed-independent modules. IEEE Trans. on Computers, C-23, 1, 21-33 (Jan. 1974).

[Keller 75] R.M. Keller, A fundamental theorem of asynchronous parallel computation. In T.Y. Feng (ed.), Parallel Processing, 102-112, Springer (1975).

[Keller 76] R.M. Keller, Formal verification of parallel programs. C. ACM, 19, 7, 371-384 (July 1976).

[Kimura 76] T. Kimura, An algebraic system for process structuring and interprocess communication. ACM Eighth Annual Symposium on the Theory of Computing, 92-100 (May 1976).

[Kleene 52] S.C. Kleene, Introduction to Metamathematics. Van Nostrand (1952).

[Kwong 77] Y.S. Kwong, On reduction of asynchronous systems. To appear in Theoretical Computer Science (1977).

[Lamport 76] L. Lamport, Towards a semantics of multiprocess programs. Massachusetts Computer Associates Rept. (Dec. 1976).

[Lamport 77] L. Lamport, Proving the correctness of multiprocess programs. IEEE Trans., SE-3, 2, 125-143 (March 1977).

[Lampson 69] B.W. Lampson, Dynamic protection structures, AFIPS Proc., 35, 27-38 (Fall 1969).

[van Lamsweerde and Sintzoff 76] A. van Lamsweerde and M. Sintzoff, Formal derivation of strongly correct parallel programs. MBLE Report R338 (Oct. 1976).

[Lipton 75] R. J. Lipton, Reduction: A method of proving properties of parallel programs. CACM, 18, 12, 717-721 (Dec. 1975).

[Manna 69] Z. Manna, The correctness of programs. JCSS, 3, 2, 119-127 (May 1969).

[Manna and Vuillemin 72] Z. Manna and J. Vuillemin, Fixpoint approach to the theory of computation. C. ACM, 15, 7, 528-536 (July 1972).

[Manna and Waldinger 76] Z. Manna and R. Waldinger, Is sometimes better than always? IEEE Software Engineering Conference (Oct. 1976).

[Mazurkiewicz 76] A. Mazurkiewicz, Invariants of concurrent programs. International conference on data processing, 3-31 Warsaw, Poland (1976).

[Miller and Cocke 74] R.E. Miller and J. Cocke, Configurable computers: A new class of general-purpose machines. In Ershov and Nepomniaschy (eds.),

International symposium on theoretical programming, 285-298, Springer (1974).

[Milner 73] R. Milner, An approach to the semantics of parallel programs. Proc. Convergno di Informatica Teorica. Pisa (1973).

[Patil 67] S. Patil, Parallel evaluation of lambda-expressions. Unpublished manuscript (Jan. 67).

[Patil 70] S. Patil, Closure Properties of interconnections of determinate systems. Proc. Project MAC Conference on Concurrent Systems and Parallel Computation, 107-116 (June 1970).

[Plotkin 76] G.D. Plotkin, A power domain construction. SIAM J. Comp. 5, 3, 452-487 (Sept. 1976).

[Riddle 76] W.E. Riddle, An approach to software system modelling, behavior specification and analysis. Dept. of Computer and Communication Sciences, RSSM/25, University of Michigan (July 1976).

[Ritchie and Thompson 75] D.M. Ritchie and K. Thompson, The Unix time-sharing system. C. ACM, 17, 7, 365-381 (July 1975).

[Rodriguez 69] J.E. Rodriguez, A graph model for parallel computation. MIT Project MAC Tech. Rept. TR-64 (1969).

[Scott 76] D. Scott, Data types as lattices. SIAM J. Comput., 5, 3, 522-587 (Sept. 1976).

[Seror 70] D. Seror, DCPL: A distributed control programming language. Tech. Rept. UTEC-CSc-70-108, Dept. of Computer Science, University of Utah (Dec. 1970).

[Stoy and Strachey 72] J.E. Stoy and C. Strachey, OS6 - An experimental operating system for a small computer. Computer J. 15, 3, 195-203 (1972).

[Urchsler 73] G. Urchsler, The transformation of flow diagrams into maximally parallel form. Proc. 1973 Sagamore Conference on Parallel Processing, 38-46.

[Vuillemin 73] J.E. Vuillemin, Proof techniques for recursive programs, Stanford Rept. CS-73-393 (Oct. 1973).

[Zadeh and Desoer 63] L.A. Zadeh and C.A. Desoer, Linear system theory. McGraw-Hill (1963).