JOURNAL OF COMPUTER AND SYSTEM SCIENCES 19, 290–308 (1979)

Semantics of Nondeterminism, Concurrency, and Communication*

NISSIM FRANCEZ

Department of Computer Science, The Technion, Haifa, Israel

C. A. R. HOARE

Programming Research Group, Oxford University, Oxford, OX2 6PE, England

DANIEL J. LEHMANN

Department of Computer Science, the Hebrew University, Jerusalem, Israel

AND

WILLEM P. DE ROEVER

Department of Computer Science, University of Utrecht, Utrecht, Holland

Received September 13, 1978; revised July 25, 1979

I. INTRODUCTION

1. *Background and Motivation*

One of the more important and active areas in the theory of programming languages is that of concurrent programs, specifically their design, definition, analysis, and verification. Due to recent developments in the technology of microprocessors, there is a trend toward languages supporting distributed activities involving communication rather than concurrent activities on some shared resources, mainly memory. Thus, it becomes very important to supply adequate tools for the definition and analysis of such programs and programming languages.

One recent attempt to design such a language was done by Hoare [9], where the language CSP (communicating sequential processes) was presented informally. This is a language for the expression of nondeterministic, concurrent, and communicating programs.

* Based on work carried out by Francez, Hoare, and de Roever at Queen's University, Belfast, during the academic year 1976–1977, supported by SRC Grant B/RG/74082.

The completion of the paper was supported by NSF Grants MCS-78-673 and MCS-78-7461, during the stay of Francez and Lehmann at the University of Southern California, Los Angeles, and of de Roever at the University of California, Berkeley, 1977–1978.

0022-0000/79/060290-19$02.00/0

The main features which distinguish CSP are:

a. Processes are disjoint, do not have any shared variables. The only contact between processes is by means of communication. Concurrency is explicit on the process level.

b. Communication is achieved by means of input and output operations, which are expressed by primitives of the language. Communication plays a double role of both message passing and synchronization. Communication is always directed, having syntactically specified source and target processes, and a strongly typed message.

c. Processes are nondeterministic, and the language allows one to distinguish between two kinds of nondeterminism, discussed in the sequel.

Syntax and informal meaning of CSP are described by way of example:

$$P::[P_1 \| P_2 \| P_3],$$

where

$$\begin{aligned} &P_1::A_1;[P_2?x \to T_1 \Box P_3!y \to T_2];\\ &\qquad *[P_2!u \to T_3 \Box P_3?v \to T_4];\\ &P_2::*[P_1?s \to T_5 \Box P_1!t \to T_6 \Box P_3?w \to T_7];\\ &P_3::A_2;*[B_1 \to T_8 \Box B_2 \to T_9]; \end{aligned}$$

—"$\|$" is the parallel composition operator.

—A_i's are elementary operations such as assignment, skip, abort, etc.

—T_i's are unspecified (for abbreviating the example) program sections.

—$P_j?x$ (in P_i) is an input command, expressing an input request of P_i from P_j, and assignment of the input value to the (local) variable x. Such a command is to be executed only when P_j is ready to execute a corresponding output command $P_i!y$, meaning a request to output the value of y to P_i. Either i/o command waits until the corresponding one is ready.

—"$\Box$" is the guard separator. Guards may be Boolean (B_i's), passable when true, or i/o commands, passable when a corresponding i/o command in the addressed process is ready.

—"$*$" denotes repetition as long as there exists a passable guard.

—";" is sequential composition.

—All processes have disjoint sets of local variables, the only ones to be assigned.

Thus the language is essentially different from various other attempts to consider concurrent programs, e.g., Concurrent Pascal (Brinch-Hansen [2]), which uses monitors [10] to control access to shared variables and procedures, or the language used by Owicki [15] with critical sections.

In this paper, we

(i) Define a formal (denotational) semantics for the main constructs of CSP.

(ii) Clarify, by means of semantical analysis, the relationship between nondeterminism, concurrency, and communication.

(iii) Suggest a rigorous framework for dealing with termination and deadlock of communicating processes.

(In this paper communication is understood as interaction between disjoint processes.)

A natural question that arises is: What can be learned from the formal semantics (of CSP), that would not be apparent already from any informal semantics? To the rather stale debate concerning this issue, we add an observation arising from our personal experience on struggling with the CSP language. During the process of work toward proof rules for this language, to our own amazement we did not succeed in devising a satisfactory intuitive semantics (we tried as hard as we could). It became clear that first a firm, formal foundation for the semantics of CSP had to be found, before any progress on an operational semantics could be made. This was caused by the unrestricted use of i/o guards, which resulted in subtleties concerning nondeterminism, termination, scheduling, waiting, and other operational phenomena. Once the basic intuition was captured formally, the operational semantics became clear as well.

The denotational approach to the definition of the semantics of programming languages originated from a pioneering paper by Scott and Strachey [18], who have shown that a relatively small number of basic semantic constructions are needed for an adequate modeling of the realm of meanings of sequential, deterministic programs. The main idea in this approach is to attach to each program some mathematical object as its meaning, or denotation; see Strachey and Milne [22] for a survey of such a characterization of various programming language constructs. The domain of these objects is called a semantic domain. This attachment enables a mathematical proof of properties of the program, and supplies a justification for various inductive proof rules. The process of attaching a meaning to a program uses induction on the syntactic structure of the program. Because of the presence of circular definitions, e.g., recursive procedures, a mathematical theory had to be developed (Scott [19]) in order to prove the existence of the required denotations. According to this theory, a program denotes a partial function from one domain to another. In case of circular definitions (e.g., recursive ones), it can be shown that there exists a unique partial function which satisfies this definition and is a limit of a sequence of partial functions, each of which is at least as well defined as its predecessor. Using tools from lattice algebra and topology, Scott was able to give the appropriate foundations needed for this approach.

More recently, Plotkin [16] extended this approach to cover also nondeterministic programs. By a construction of power domains, which are domains of certain sets of elements from the base domain ordered in an appropriate approximation ordering (see also Egli [6]), he was able to supply denotations to nondeterministic programs by using set-valued partial functions. Another power domain construction appears in Smyth [20].

Milner [14] suggests a construction, called renewals (or resumptions), to give denotations to concurrent programs. However, the programs he has in mind in [14] involve highly interleaved actions on shared variables. Thus, an action by a process may either deliver a result, or give rise to a new process yet to be interleaved with other processes. Since process interaction in CSP is by means of communication rather than by means of sharing memory, a different description of the semantics is enabled. We suggest another construction which is adequate for communicating processes in CSP (compare also Milne and Milner [13]) and which reduces the degree of interleaving.

The importance of this work is twofold:

(1) The formal semantics for CSP clarifies many of the complex issues which are needed in order to formulate and validate proof rules for CSP; compare [7].

(2) The clearer relationship between concurrency, nondeterminism, and communication suggests a way for both the design of language constructs which diminish the danger of deadlock, and the construction of terminating programs.

In the next section, we specify some new contributions of the paper, which expand on (1) and (2) above.

2. *What is New?*

2.1. *A Priori Semantics*

We regard a single process (taken out of a set of communicating processes) by itself to be a ***semantically meaningful entity***, which deserves a denotation of its own. Therefore, we are led to a definition of semantics which attributes a separate meaning to each component P_i of P::$[P_1 \| \cdots \| P_n]$. This kind of semantics is called ***a priori*** because it denotes ***all*** the communication capabilities of P_i when confronted with ***any*** environment, i.e., all other processes in P. At the next level we introduce a binding operator $\mathscr{B}$, which combines the set of all separate a priori meanings of all P_i's to a joint meaning of P. This n-ary binding will be compared with Milne and Milner's binary binding in Section III.1.

Since every process has its own (disjoint) local memory, the only contact with other processes being via communication, the degree of interleaving is much smaller than with shared variables, as in each process local computation, involving no communication, does not influence in any way similar computations in other processes. Therefore, the semantics of each separate process is determined by (1) providing an initial state for the local variables and (2) describing its reaction to every possible message requested. Since the values of these messages may vary, each value specifies a different possibility for continuing the computation. These possibilities will be expressed as branches of a tree. This leads to the construction of a new semantic domain, which we call the domain of ***history trees***. The histories in question are histories of ***communication*** (i.e., traces of records of communications that might have taken place). We show that these histories are sufficient for the description of deadlock since deadlock can be caused ***only*** by some communications failing to happen. With these histories we provide a uniform alternative to mythical ("ghost") variables (e.g., Clint [3] and Owicki [15]), since these variables are used to capture parts of such histories.

In this context, communication involves a message passing from a source process to a target process. Other approaches are, e.g., Milne and Milner [13], where emphasis is put on *exchange* of values, or Kahn [11], where message transmission is *buffered*, not synchronizing.

2.2. *Nondeterminism, Concurrency, and Communication*

In our semantic domain we distinguish between *two kinds of nondeterminism*, which are expressed in CSP by means of two kinds of guarded commands [4] (this may be one of the innovations of CSP). These two types differ in the way nondeterminism is resolved, and have a different impact on achieving successful communication and deadlock freedom.

The one kind, using Boolean guards, we call local nondeterminism, and is the "old" notion introduced by Dijkstra [4]. Examined in connection with communication, it occurs when a process P_i can communicate with any of $P_{i_1} \cdots P_{i_n}$, and decides *on its own* for which communication to wait, i.e., independent of any consultation with the other processes.

The second kind, using i/o guards, we call global nondeterminism, and is resolved by inspecting the other processes w.r.t. to their willingness to communicate. Only mutual willingness to communicate may result in a de facto communication.

Mixtures of Boolean guards and i/o guards are not considered in this paper. As a simple example, consider the difference between the following two programs:

$$[P_1::[\text{true} \rightarrow P_2?x \square \text{true} \rightarrow P_2!0] \parallel P_2::P_1!1] \tag{1}$$

and

$$[P_1::[P_2?x \rightarrow \text{skip} \square P_2!0 \rightarrow \text{skip}] \parallel P_2::P_1!1]. \tag{2}$$

In the first, P_1 may choose the second alternative and cause a deadlock. In the second, successful termination is guaranteed.

In our semantic domain of history trees, this distinction is reflected by letting the history trees have two kinds of nonleaves, on which the binding operator operates differently. This difference reflects also the different implications of the presence of the two kinds of nondeterminism on freedom from deadlock, and will underlie a future proof rule [7].

2.3. *The Use of End-Signaling for Termination*

CSP [9] enables a neat handling of loops which depend on communication guards, and also enables the abortion of the corresponding selection. Upon termination, a process P_i reaches a final state, which may be sensed by all processes communicating with P_i. A guard consisting of a communication request is regarded as false iff the target process has already terminated! Otherwise, waiting occurs, because the communication may take place in the future. Thus, a loop depending on guards communicating with $P_{i_1} \cdots P_{i_n}$ is exited only if *all* of these processes have terminated. Correspondingly, a selection depending on such guards aborts.

Note that termination is in general *not* a property of a single process. As a typical example, consider a "service process" which responds to requests until it receives a signal meaning "terminate!" If presented with an infinite sequence of requests, it should produce an infinite sequence of responses. Only the pair consisting of user-process and service-process may provably terminate. For a more interesting example, displaying how intricate the termination of such programs may be, see Dijkstra [5]. Thus, although every possibility for (non)termination is already present in the a priori meaning of a single process, actual (non)termination of the combination of all processes is determined only on the level of the binding operator $\mathscr{B}$.

Dealing with *terminating* processes is an essential feature of the conception of CSP. This may be a distinguishing trait of our formalism compared to Milne and Milner's, in its present form [13]. They consider nonterminating processes, and the problem of termination has never arisen in their work.

II. A Domain $\mathscr{T}_i$ and Semantic Equations for $\mathscr{M}[\![P_i]\!]$, the a Priori Semantics of P_i

The distinction between local and global nondeterminism implies that a domain of ordinary trees (whose arcs are possibly labeled by records of communication) is not sufficiently structured to reflect this distinction. For, whenever local nondeterminism is resolved, a number of independent alternatives are created, each of which has to be independently confronted with the environment. However, global nondeterminism postpones resolution until the moment of binding since it looks for mutual consent with the environment and can therefore only be resolved during binding.

Therefore, a more refined structure of (finite and infinite) trees with *two* kinds of nodes, called *local* nodes and *global* nodes, is needed.

Since at any stage in its computation, at the level of elementary statements and operations, a process can make only a finite number of nondeterministic choices, any local node has only a finite, positive, number of outgoing arcs.

A global node signals willingness to communicate. Therefore, any arc outgoing a global node is labeled by a target process identifier. Willingness to communicate means either willingness to output a value or to input one of the appropriate type.

At any instant, a process may have a global nondeterministic choice to communicate with a finite number of processes, and therefore the corresponding node will have a finite number of outgoing edges, specifying these processes.

There will be also nodes corresponding to a single input command, and these may have an infinite number of outgoing branches, each labeled by a record of communication corresponding to one of the possible input values. An output command will be represented by a node with a single outgoing edge, labeled by the record of communication corresponding to the output value.

We now proceed with the formal definition of this domain.

Definition. Let A be any nonempty set, called the *Communication Alphabet.*

Members of A represent messages passed via i/o from one process to other. In this paper we shall assume that $A = \{n \mid n \geqslant 0\}$.

DEFINITION. A record of communication (roc) is a triple $\sigma = \langle a, i, j \rangle$, $a \in A$.

The intended interpretation of σ is that of the message a passed from P_i to P_j. Let $1 \leqslant i, j \leqslant n$. Then $\Sigma_i^j = \{\langle a, i, j \rangle \mid a \in A\}$, where $i \neq j$, and $\Sigma_i^i = \varnothing$. Also, let $\Gamma_i = \{1, \ldots, n\} - \{i\}$. $\Sigma_i = \bigcup_{j \in \Gamma_i} \Sigma_i^j$ and $\Sigma^j = \bigcup_{i \in \Gamma_j} \Sigma_i^j$.

DEFINITION. Let V_i denote the set of (local) variables of P_i. $S_i = [V_i \to A] \cup \{fail\}$ is the set of *states* of P_i.

We consider a state to be mapping from variables to values. We avoid the consideration of "environments" [18] since these do not change in the restricted language we consider; *fail* is a special state denoting a failing computation.

For $s \in S_i$, $s \neq fail$, $x \in V_i$, and $a \in A$, $s_a^x = \lambda y.\ if\ y = x\ then\ a\ else\ s(y)$.

Next, we define the complete partial order (cpo) $\mathscr{T}_i$ as the least solution (in the category of cpo's) of a domain equation. The reader unfamiliar with the technicalities of this kind of equations could consult [21, 22]. $\mathscr{T}_i$ is the domain of history trees corresponding to P_i, and will be used as the range of the semantic function $\mathscr{M}[\![P_i]\!]$, characterizing the a priori semantics of P_i.

$$\mathscr{T}_i = \left(S_i \cup \left(\bigcup_{j \in \Gamma_i} [\Sigma_j^i \to \mathscr{T}_i]\right) \cup \left(\bigcup_{j \in \Gamma_i} (\Sigma_i^j \times \mathscr{T}_i)\right) \cup (\Gamma_i \times \mathscr{T}_i)^+ \times S_i \cup \mathscr{T}_i^+\right)_\perp . \quad (1)$$

Here:

$$X^+ = \mu F(X) \qquad \text{and} \qquad F(X) = \lambda Y \cdot X \oplus X \otimes Y,$$

i.e., X^+ is the domain of finite (nonempty) sequences over $X - \{\perp\}$ with the following ordering: there is one bottom element, sequences of different lengths are not comparable, and sequences of the same length are ordered coordinatewise by the ordering inherited from X.

Equation (1) defines a (finite or infinite) tree in $\mathscr{T}_i$ to be either bottom or belonging to one of five addends.

Formally: If A is a partially ordered set then $A_\perp$ is obtained by adding to A a new bottom element. The union symbol $\bigcup$ denotes disjoint union of partially ordered sets (no bottom element is added); union is thus associative. (For instance, an element $A \cup B \cup C$ is either in A or in B or in C and corresponds therefore with three cases; this is in contrast to the customary usage of the disjoint sum $A + B + C$ which adds more bottom elements and therefore creates more partially defined objects (and is not associative)).

If A is a set and B a cpo, $[A \to B]$ is the cpo of all total functions from A to B with the obvious ordering.

The symbol $\times$ denotes Cartesian product μ denotes the least fixed point operator.

The coalesced sum $\oplus$ and the coalesced product $\oplus$ have been defined in [12, 21]. They are used to avoid the introduction in X^+ of partially undefined and infinite objects, as explained in [12] (warning: In [12] the coalesced sum is denoted by $+$).

Intuitive explanation: The solution to Eq. (1) can be thought of as the domain of all finite and infinite trees, which have the following nodes:

a. *Leaves* are labeled by $S_i \cup \{\perp\}$ and have no outgoing arcs.

b. *Input nodes* have a (possibly infinite) number of outgoing arcs, each labeled by some $\sigma \in \Sigma^i$.

c. *Output nodes* have one outgoing arc, labeled by some $\sigma \in \Sigma_i$.

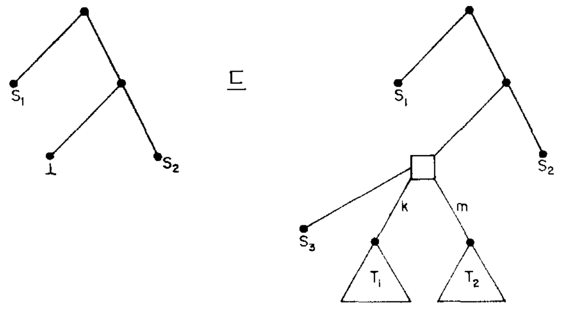
d. *Global nodes* have a finite, positive, number of outgoing arcs, labeled by Γ_i and an additional unlabeled arc (will be denoted in figures as □).

e. *Local nodes* have a finite, positive, number of unlabeled arcs (will be denoted in figures as ●).

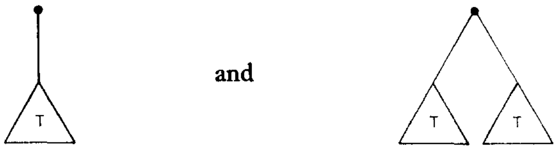
Note that each kind of node corresponds to a particular addend in (1). Equation (1) has been designed so as to induce the following ordering on $\mathcal{T}_i$ (see [12, 21]):

$T_1 \sqsubseteq T_2$ iff T_2 may be obtained from T_1 by replacing some $\perp$-labeled leaf by some $T' \in \mathcal{T}_i$.

Thus,



This is very refined ordering, which, e.g., distinguishes between



Such distinctions are natural in a context in which histories (and not only terminal values) matter. One could envisage a semantics of the whole program reflecting the difference of such incomparable trees, e.g., by counting nondeterministic choices involved in producing a given final value. Our specific semantics ignored this difference at the binding level. Both trees will produce the same results while bound to the same environment. The smallest element in this ordering is $\perp$. We shall denote by i_S, i_I, i_O, i_G, and i_L the corresponding injections from the components to their corresponding copies in $\mathcal{T}_i$; cf. [18].

Thus,

$$i_{\mathrm{S}}\colon S_i \to \mathscr{T}_i\,, \qquad i_{\mathrm{I}}\colon \bigcup_j [\Sigma_j^i \to \mathscr{T}_i] \to \mathscr{T}_i\,, \qquad \text{etc.}$$

Before defining the semantic function $\mathscr{M}[\![P_i]\!]$, we define an auxiliary function $\mathscr{R}$ (replacement), which generalizes functional composition from the sequential case.

When defining the meaning of S_1 ; S_2 , S_1 has already produced an (intermediate) history tree. Thus, the meaning of S_2 has to be applied to *all* possible leaves of that tree, and will in general depend on the states labeling the leaves of this tree; e.g., if S_2 starts with some Boolean selection, the state will determine the selected branch(es).

$$\mathscr{R}\colon \mathscr{T}_i \times [S_i \to \mathscr{T}_i] \to \mathscr{T}_i$$

and the meaning of $\mathscr{R}[T, F]$ is the tree obtained by replacing every leaf labeled s by the tree $F(s)$. (We assume $F(\mathit{fail}) = \mathit{fail}$.)

$\mathscr{R}$ is defined recursively by structural induction on T:

$$\begin{aligned}
\mathscr{R}(T,F] &= \bot, && \text{if} \quad T = \bot,\\
&= F(T), && \text{if} \quad T \in i_{\mathrm{S}}(S_i),\\
&= i_{\mathrm{I}}(\lambda\sigma \cdot \mathscr{R}[T(\sigma), F]), && \text{if} \quad T \in i_{\mathrm{I}}([\Sigma_j^i \to \mathscr{T}_i]),\\
&= i_{\mathrm{O}}(\langle T\downarrow 1, \mathscr{R}[T\downarrow 2, F]\rangle), && \text{if} \quad T \in i_{\mathrm{O}}(\Sigma_i^j \times \mathscr{T}_i), \qquad\qquad (2)\\
&= i_{\mathrm{G}}(\langle\langle\langle T'\downarrow 1\downarrow 1, \mathscr{R}[T'\downarrow 1\downarrow 2, F]\rangle, \ldots, \langle T'\downarrow k\downarrow 1, \mathscr{R}[T'\downarrow k\downarrow 2, F]\rangle\rangle, F(s)\rangle),\\
&\quad \text{where } s = T\downarrow 2 \text{ and } T' = T\downarrow 1, && \text{if} \quad T \in i_{\mathrm{G}}(\Gamma_i \times \mathscr{T}_i)^k \times s),\\
&= i_{\mathrm{L}}(\langle \mathscr{R}[T\downarrow 1, F], \ldots, \mathscr{R}[T\downarrow k, F]\rangle, && \text{if} \quad T \in i_{\mathrm{L}}(\mathscr{T}_i^k)
\end{aligned}$$

($\downarrow i$ denotes the projection to the ith component of an n-tuple).

Note that $\mathscr{R}$ does not affect infinite paths in T. Clearly, $\mathscr{R}$ is continuous. The apparent (notational) complexity of the definition of $\mathscr{R}$ is due to the fact that a domain equation for denoting our history trees had to be employed. Had we considered such trees as self-explanatory, a notationally simpler definition of $\mathscr{R}$ could have arisen; compare Hoare [23].

Next, we proceed with the formal definition of $\mathscr{M}[\![P_i]\!]\colon S_i \to \mathscr{T}_i$, and the informal explanation of each clause in the definition.

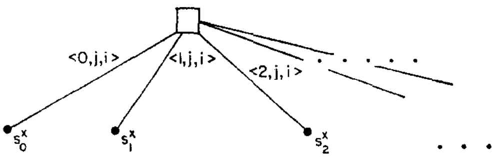
(1) $\mathscr{M}[\![Q]\!](\mathit{fail}) = \mathit{fail}$ for all Q. In the sequel, we assume $s \neq \mathit{fail}$!

(2) *Input*

$$\mathscr{M}[\![P_j\,?x]\!] = \lambda s \cdot i_{\mathrm{I}}(\lambda\sigma \cdot i_{\mathrm{S}}(s^x_{\sigma\downarrow 1})).$$

We get a new function, returning for each s the modified state, which records the side effect of input the value component $\sigma \downarrow 1$ of the input (roc) σ.

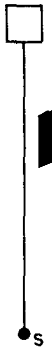
Thus, $P_j\,?x$ creates the following tree, to be called an input node.



(3) *Output*

$$\mathcal{M}[\![P_j!x]\!] = \lambda s \cdot i_O(\langle\langle s(x), i, j\rangle, i_S(s)\rangle).$$

We get a tree with one arc to be called an output node.



The subtree labeled s indicates that output has no side effect, and the output value will have to be matched by $\mathcal{B}$ to an input arc in the tree corresponding to P_j.

(4) *Assignment*

$$\mathcal{M}[\![x := e]\!] = \lambda s.\ \textit{if}\ \mathcal{V}(e, s) = \textit{fail then fail}$$
$$\textit{else}\quad i_S(s^x_{\mathcal{V}(e,s)}).$$

$\mathcal{V}(e, s)$ is an auxiliary function which computes the value of an expression e in state s. We assume $\mathcal{V}(e, s) = \textit{fail}$ if e is undefined. We do not consider recursive functions here, so the evaluation of expressions always terminates, and yields *fail* in cases like division by 0. The meaning of assignment is to update the state s. Note that it does not create any new arcs in the tree.

(5) *Skipping*

$$\mathcal{M}[\![\text{skip}]\!] = \lambda s \cdot i_S(s). \qquad \text{Obvious.}$$

(6) *Sequential composition*

$$\mathcal{M}[\![S_1\ ;\ S_2]\!] = \lambda s \cdot \mathcal{R}[\mathcal{M}[\![S_1]\!](s), \mathcal{M}[\![S_2]\!]].$$

For a given state s, we first apply $\mathcal{M}[\![S_1]\!]$ to s, obtaining a tree, say T_s. Then, we apply the replacement operator $\mathscr{R}$ to T_s and the function $\mathscr{M}[\![S_2]\!]$. This reflects the fact that the operation of S_2 depends upon the final state of $\mathscr{M}[\![S_1]\!]$, which it continues.

Note that $\mathscr{M}[\![S_1]\!](\mathit{fail}) = \mathit{fail}$ by assumption (case (1)), and $\mathscr{R}$ may therefore be applied with $\mathscr{M}$ as an argument. Also, if S_1 has a nonterminating computation, $\mathscr{M}[\![S_1]\!](s)$ will have an infinite path, which will not be affected by $\mathscr{R}$.

(7) *Boolean selection.* We treat here the case of two guards only. The extension to any number of guards should be clear. We assume guards are always defined.

$$\mathscr{M}[\![[B_1 \rightarrow S_1 \square B_2 \rightarrow S_2]]\!] = \lambda s \cdot T_s,$$

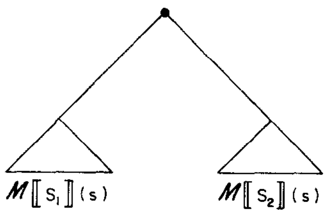
where

$$\begin{aligned} T_s = {} & \mathit{case}\ \langle \mathscr{V}(B_1, s), \mathscr{V}(B_2, s)\rangle\ \mathit{of} \\ & \langle \mathit{ff}, \mathit{ff}\rangle \colon i_{\mathrm{S}}(\mathit{fail}); \\ & \langle \mathit{ff}, \mathit{tt}\rangle \colon \mathscr{M}[\![S_2]\!](s); \\ & \langle \mathit{tt}, \mathit{ff}\rangle \colon \mathscr{M}[\![S_1]\!](s); \\ & \langle \mathit{tt}, \mathit{tt}\rangle \colon i_{\mathrm{L}}(\langle \mathscr{M}[\![S_1]\!](s), \mathscr{M}[\![S_2]\!](s)\rangle). \end{aligned}$$

In case both guards are false, computation is aborted. In case exactly one guard is true, then $\mathscr{M}$ of the corresponding guarded statement is applied to s.

In case both guards are true, a *local* node is created, reflecting in its two unlabeled subtrees the two independent continuations, thus recording local nondeterminism, which will cause independent binding of each subtree.

The picture for this case is:



(8) *i/o directed selection.* Again, we shall describe the semantics of a particular case, involving only two guards, both being input guards. The description of $\mathscr{M}$ for more (or less) than two guards, and for output guards, should be clear.

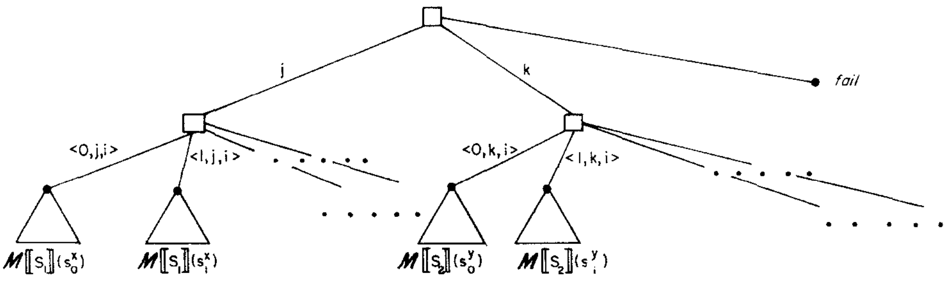
$$\mathscr{M}[\![[P_j ?x \rightarrow S_1 \square P_k ?y \rightarrow S_2]]\!] = \lambda s \cdot T_s,$$

where

$$\begin{aligned} T_s = i_{\mathrm{G}}(\langle\langle\langle & j, i_{\mathrm{I}}(\lambda\sigma \cdot \mathscr{M}[\![S_1]\!](s^x_{\sigma\downarrow 1}))\rangle, \\ & \langle k, i_{\mathrm{I}}(\lambda\sigma \cdot \mathscr{M}[\![S_2]\!](s^y_{\sigma\downarrow 1}))\rangle\rangle \\ & i_{\mathrm{S}}\,(\mathit{fail})\rangle). \end{aligned}$$

An input can be accepted from either P_j or from P_k, and then the corresponding guarded statement will be executed. The decision as to which continuation to take is postponed to the binding time, when the state of P_j and P_k will be available, thus reflecting the global nondeterminism.

The picture corresponding to this case is:


j
k
fail
<0,j,i>
<1,j,i>
<0,k,i>
<1,k,i>
$\mathcal{M}[\![S_1]\!](s_0^x)$
$\mathcal{M}[\![S_1]\!](s_1^x)$
$\mathcal{M}[\![S_2]\!](s_0^y)$
$\mathcal{M}[\![S_2]\!](s_1^y)$

The *fail* subtree will be used if both P_j and P_k have terminated, a fact to be noticed at binding.

Since all the auxiliary functions applied so far are continuous, the definition of $\mathcal{M}$ for loops by means of least fixed points is justified.

(9) *Boolean repetition*

$$\mathcal{M}[\![*[B_1 \to S_1 \Box B_2 \to S_2]]\!] = \mu(\lambda F \cdot \lambda s \cdot T_s),$$

where

$$\begin{aligned} T_s = {} & \textit{case}\ \langle \mathcal{V}(B_1, s), \mathcal{V}(B_2, s)\rangle\ \textit{of} \\ & \langle ff, ff\rangle\colon i_S(s); \\ & \langle ff, tt\rangle\colon \mathcal{R}[\mathcal{M}[\![S_2]\!](s), F]; \\ & \langle tt, ff\rangle\colon \mathcal{R}[\mathcal{M}[\![S_1]\!](s), F]; \\ & \langle tt, ff\rangle\colon i_L(\langle \mathcal{R}[\mathcal{M}[\![S_1]\!](s), F], \mathcal{R}[\mathcal{M}[\![S_2]\!](s), F]\rangle). \end{aligned}$$

If both guards are false, the loop is exited. In case exactly one guard is true, the corresponding guarded statement is executed and the whole guarded command is attempted again.

In case both guards are true, a local node is created, and both continuations are recorded as subtrees of this node.

(10) *i/o directed repetition*

$$\mathcal{M}[\![*[P_j ?x \to S_1 \Box P_k ?y \to S_2]]\!] = \mu(\lambda F \cdot \lambda s \cdot T_s),$$

where

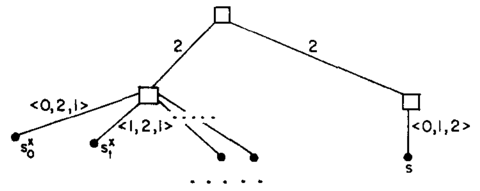
$$\begin{aligned} T_s = i_G(\langle\langle\langle & j, i_I(\lambda\sigma \cdot \mathcal{R}[\mathcal{M}[\![S_1]\!](s^x_{\sigma\downarrow 1}), F])\rangle, \\ & \langle k, i_I(\lambda\sigma \cdot \mathcal{R}[\mathcal{M}[\![S_2]\!](s^y_{\sigma\downarrow 1}), F])\rangle\rangle, \\ & i_S(s)\rangle). \end{aligned}$$

If both processes end, the loop is exited. Otherwise an input is selected (again, postponing the decision from which process), the corresponding statement S executed, and the whole loop attempted again.

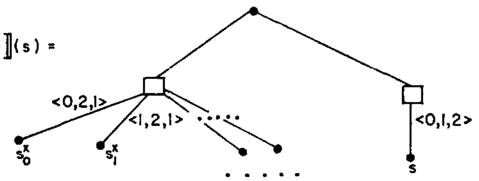
This completes the definition on $\mathcal{M}[\![P_i]\!]$ by means of semantic equations. As a simple example, reconsider the program presented in the Introduction.

Let $s \in S_1$, $s' \in S_2$ be two initial local states.

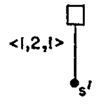
$\mathcal{M}[\![P_1]\!](s) =$



$\mathcal{M}[\![P_1']\!](s) =$



$\mathcal{M}[\![P_2]\!](s') =$



Anticipating the binding function, one can see intuitively that the second program can never fail, since it will choose that alternative in P_1 which will match P_2 (i.e., the first). On the other hand, the first program *may* fail, if P_1' chooses its second guard as a "wrong" independent choice, thereby causing deadlock.

III. The Binding Operator $\mathcal{B}$

1. The purpose of the binding operator is to attach a joint meaning to a concurrent command $P{::}[P_1 \| \cdots \| P_n]$ in an initial state $\langle s_1, ..., s_n \rangle$ by using the history trees $\mathcal{M}[\![P_1]\!](s_1), ..., \mathcal{M}[\![P_n]\!](s_n)$.

We restrict ourselves to "closed" concurrent commands since the semantics for, and the proper restrictions to be imposed upon, nonclosed concurrent commands are still under scrutiny; cf. Hoare [9]. (By "closed" we mean that no component P_i of P communicates with any process not among $P_1, ..., P_{i-1}, P_{i+1}, ..., P_n$.)

The computation of P starts in $\langle s_1, \ldots, s_n \rangle$ and may produce a set of such n-tuples as final states. While binding, all the histories of communication are "forgotten" and only final states are left; cf. also Milne and Milner [13]. Any occurrence of an infinite computation in any process P_i is recorded by a single formal value $\perp$ standing for the "undefined" n-tuple. We also include two other states, *deadlock*, which records a deadlock situation, and *fail*, which records abortion in any component. The meaning of P is given by $\mathscr{B}(\mathscr{M}[\![P_1]\!](s_1), \ldots, \mathscr{M}[\![P_n]\!](s_n))$.

In Milne and Milner [13], a binary binding of processes is introduced. Thus, $[P_1 \| P_2 \| P_3]$ is interpreted as, e.g., $[(P_1 \| P_2) \| P_3]$, where "(,)" correspond to the binary binding. The binary binding is associative. This operation leads to intermediate trees which describe a mixture of global *and* local nondeterminism on the same level. This mixture arises from internalization of communication between the bound processes. Consider the following example:

$$P::[P_1 \| P_2 \| P_3],$$

where

$$\begin{aligned} &P_1::[P_2?x \rightarrow P_3?y \square P_3?y \rightarrow P_2?x],\\ &P_2::P_1!z,\\ &P_3::P_1!u. \end{aligned}$$

Then, $[(P_1 \| P_2) \| P_3]$ is equivalent to

$$[[\text{true} \rightarrow x := z;\ P_3?y \square P_3?y \rightarrow x := z] \| P_1!u].$$

Such mixed trees introduce subtleties, which on the level of program constructs, as seen from the above example, introduce guarded commands with a mixture of Boolean and i/o guards. These subtleties are our reason for excluding such mixtures from the present paper, and considering instead n-ary binding of closed commands, which avoids the introduction of such trees.

1.1. *On the Egli-Milner Order*

Before defining the binding operator $\mathscr{B}$ we first need to introduce the concept of the Egli-Milner order to describe the value domain of $\mathscr{B}$—a certain collection of subsets of $S_1 \times \cdots \times S_n \cup \{\perp, \textit{fail}, \textit{deadlock}\}$ and its underlying structure (since $\mathscr{B}$ is defined recursively, $\mathscr{B}$'s existence follows from the usual continuity considerations with respect to this order).

The concept of the Egli-Milner order (Egli [6], Plotkin [16]) dates back to 1975, and constituted a breakthrough in the semantics of nondeterminism, and a fortiori, of concurrency; its application in de Roever [17] resulted in the first comprehensive model—i.e., including a characterization of nondeterminism—of Dijkstra's predicate transformer; and it is based on a powerful intuition which is best explained in Egli's unpublished paper, extensively cited in, e.g., de Bakker [1].

Let D denote any nonempty set with $\perp \notin D$. $P_{\text{E-M}}(D \cup \{\perp\})$ denotes the collection C of all nonempty subsets V of $D \cup \{\perp\}$ satisfying: If $V \in C$ and V is infinite, then $\perp \in V$. Order $P_{\text{E-M}}(D \cup \{\perp\})$ as follows: For $V_1, V_2 \in P_{\text{E-M}}(D \cup \{\perp\})$.

$$V_1 \sqsubseteq_{\text{E-M}} V_2 \text{ iff } \textit{either } \perp \in V_1 \text{ and } V_1 - \{\perp\} \subseteq V_1$$
$$\text{(set-theoretical containment)}$$
$$\textit{or } \perp \notin V_1 \text{ and } V_1 = V_2.$$

$\langle P_{\text{E-M}}(D \cup \{\perp\}), \sqsubseteq_{\text{E-M}}\rangle$ is a complete partial order, called the Egli–Milner order.

We shall use the property that if $V_i \in P_{\text{E-M}}(D \cup \{\perp\})$ and $V_i \sqsubseteq_{\text{E-M}} V_{i+1}$, for $i \in N$, then $y \in \text{l.u.b.}_i\, V_i$ iff $\exists j \cdot y \in V_j$. This concept is related to nondeterminism in that functions with values in $P_{\text{E-M}}(D \cup \{\perp\})$, such as $\mathscr{B}$, describe nondeterministic program behavior. The extension of this concept to subsets of any complete partial order $\langle L, \sqsubseteq_{\text{L}}\rangle$ has been described by Plotkin [16]. Smyth [20] provides a masterful account how this order can be more simply described for the general case.

1.2. *The Definition of $\mathscr{B}$*

The functionality of $\mathscr{B}$ for any $n \geqslant 2$ is given by

$$\mathscr{B}\colon \mathscr{T}_1 \times \cdots \times \mathscr{T}_n \to P_{\text{E-M}}((S_1 \times \cdots \times S_n) \cup \{\perp, \textit{fail}, \textit{deadlock}\}).$$

First, we give a recursive definition of $\mathscr{B}(T_1, \ldots, T_n)$ accompanied by an informal explanation of the role of each clause in the definition. We shall number the successive clauses in the margin, for convenience.

$$\mathscr{B}(T_1, \ldots, T_n) = \{\perp \mid \exists i\, (1 \leqslant i \leqslant n).\ T_i = \perp\}. \tag{1}$$

$\{\perp\}$ is the bottom element of $P_{\text{E-M}}$ mentioned above, and denotes undefined information. A $\perp$-node in $\mathscr{T}_i$ is used to describe approximations to elements in $\mathscr{T}_i$, and will appear in the approximations to the a priori semantics of loops within P_i. This clause is needed for the continuity of $\mathscr{B}$.

(For all subsequent clauses, assume $\forall i\, (1 \leqslant i \leqslant n).\ T_i \neq \perp$.)

$$\bigcup \{\textit{fail} \mid \exists i\, (1 \leqslant i \leqslant n).\ T_i = i_{\text{S}}(\textit{fail})\}. \tag{2}$$

Fail is a formal value denoting some machine-detectable error, such as a selection with false guards only, etc. It is preserved under the a priori semantics (see clause 1 in the definition of $\mathscr{M}$), and could be used to issue an error message (Goguen [8]). Once such an error occurs within any $\mathscr{M}[\![P_i]\!](s_i)$, it will be reflected in the value of $\mathscr{B}$. Note that $\perp \sqsubseteq \textit{fail}$.

(For all subsequent clauses, assume $\forall i\, (1 \leqslant i \leqslant n).\ T_i \neq i_{\text{S}}\,(\textit{fail})$.)

$$\bigcup \{\langle s_1, \ldots, s_n\rangle \mid \forall i\, (1 \leqslant i \leqslant n).\ T_i = s_i \in i_{\text{S}}(S_i)\}. \tag{3}$$

This is the case of successful termination of all P_i's, each reaching a final state $s_i \in S_i$. Then we add the tuple $\langle s_1, ..., s_n \rangle$ to the set of final values of $\mathscr{B}$.

$$\bigcup \mathscr{B}(T_1, ..., T_{i-1}, T_i(T_j \downarrow 1), ..., T_{j-1}, T_j \downarrow 2, ..., T_n) \quad \text{if} \quad T_i \in i_{\mathrm{I}}([\Sigma_j{}^i \rightarrow \mathscr{T}_i]) \quad \text{and} \quad T_j \in i_{\mathrm{O}}(\Sigma_j{}^i \times \mathscr{T}_j). \tag{4}$$

This is the case of successful communication, where T_i contains an input node (from P_j) and T_j contains a corresponding output node (to P_i), with matching roc's. Then, T_i is replaced by the subtree corresponding to this matching roc, which is obtained by applying T_i, which is a function, to this roc, and T_j is replaced by its (unique) subtree; then $\mathscr{B}$ is called recursively.

$$\bigcup \{\mathit{fail} \mid T_i \in i_{\mathrm{I}}([\Sigma_j{}^i \rightarrow \mathscr{T}_i]) \text{ and } T_j \in i_{\mathrm{S}}(S_j), \text{ or } T_i \in i_{\mathrm{O}}(\Sigma_i{}^j \times \mathscr{T}_i) \text{ and } T_j \in i_{\mathrm{S}}(S_j)\} \tag{5}$$

This is the case of unsuccessful communication, where T_i is an input node or an output node, and $T_j \in S_j$ is a final state of P_j, meaning that P_j has already terminated. A communication attempt with a terminated process is interpreted as failure, and the value is $\{\mathit{fail}\}$.

$$\bigcup \mathscr{B}(T_1, ..., T_{i-1}, T_i{}^j, ..., T_n) \quad \text{if} \quad T_i = \langle T_i{}^1, ..., T_i{}^m \rangle \in i_{\mathrm{L}}(\mathscr{T}_i{}^+), \qquad 1 \leqslant j \leqslant m, \tag{6}$$

This is the case of local nondeterminism in P_i involving selection with Boolean guards. As already noted, the meaning is that *any* of the subtrees of T_i can be chosen, and bound to the other T_j's. Thus, we pick an arbitrary j, $1 \leqslant j \leqslant m$, and replace T_i by its subtree $T_i{}^j$ in the recursive call. Since the union is taken over all possibilities, each $T_i{}^j$ will be considered.

$$\begin{aligned}&\bigcup \mathscr{B}(T_1, ..., T_{i-1}, T_i{}^j, ..., T_{j-1}, T_j, ..., T_n)\\ &\qquad \text{if} \quad T_i = \langle\langle\langle k_1, T_i{}^1\rangle, \langle k_2, T_i{}^2\rangle, ..., \langle k_m, T_i{}^m\rangle\rangle, s\rangle,\\ &\text{and}\\ &\qquad \text{either} \quad T_i{}^j \in i_{\mathrm{I}}([\Sigma_{k_j}^j \rightarrow \mathscr{T}_i]), \text{ and } T_{k_j} \in i_{\mathrm{O}}(\Sigma_{k_j}^i \times \mathscr{T}_{k_j})\\ &\qquad \text{or} \quad T_i{}^j \in i_{\mathrm{O}}(\Sigma_i^{k_j} \times \mathscr{T}_i) \text{ and } T_{k_j} \in i_{\mathrm{I}}([\Sigma_i^{k_j} \rightarrow \mathscr{T}_{k_j}]), \qquad \text{for} \quad 1 \leqslant j \leqslant m,\end{aligned} \tag{7}$$

This is a case of global nondeterminism in P_i. T_i is a global node, with subtrees corresponding to communication with $P_{k_1}, ..., P_{k_m}$. For some j, $1 \leqslant j \leqslant m$, $T_i{}^j$ is an input node (corresponding to an input guard), addressing P_{k_j}, and T_{k_j} is an output node addressing P_i. Thus the global nondeterminism can be successfully resolved. Note that this binding step does not reflect the establishment of the corresponding communication. This communication will be detected at the next level of recursion, when the input node $T_i{}^j$ is confronted with the output node T_{k_j}. A similar case arises if $T_i{}^j$ is an output node (to P_{k_j}) and T_{k_j} is an input node.

$$\begin{aligned}&\mathscr{B}(T_1,\ldots,T_{i-1},T_i^p,\ldots,T_{j-1},T_j^q,\ldots,T_n)\\&\quad\text{if}\quad T_i=\langle\langle\langle k_i,T_i^1\rangle,\ldots,\langle k_m,T_i^m\rangle\rangle,s\rangle,\\&\quad\phantom{\text{if}}\quad T_j=\langle\langle\langle l_1,T_j^1\rangle,\ldots,\langle l_r,T_j^r\rangle\rangle,s'\rangle\\&\quad\text{and for some } p,q,\ k_p=j,\ l_q=i,\ \text{and}\\&\quad\text{either}\quad T_i^p\in i_{\mathrm{I}}([\Sigma_j^i\to\mathscr{T}_i]),\qquad T_j^q\in i_{\mathrm{O}}(\Sigma_j^i\times\mathscr{T}_j),\\&\quad\phantom{\text{eit}}\text{or}\quad T_i^p\in i_{\mathrm{O}}(\Sigma_i^j\times\mathscr{T}_i),\qquad T_j^q\in i_{\mathrm{I}}([\Sigma_i^j\to\mathscr{T}_i]).\end{aligned}\tag{8}$$

This is another case of resolvable global nondeterminism in both P_i and P_j. Each of them has an i/o guard addressing the other with matching roc's. Again, the actual communication will be detected at the next recursive call.

$$\begin{aligned}&\bigcup\mathscr{B}(T_1,\ldots,T_{i-1},s,T_{i+1},\ldots,T_n)\\&\text{if } T_i=\langle\langle\langle k_i,T_i\rangle,\ldots,\langle k_m,T_i\rangle\rangle,s\rangle\\&\text{and } \forall j\ (1\leqslant j\leqslant m).\ T_{k_j}\in i_{\mathrm{S}}(S_{k_j}).\end{aligned}\tag{9}$$

This is the case of unresolvable global nondeterminism in P_i, and then T_i is replaced by the "escape" state s, which accompanies the guards. The state s is a proper state if this node was generated in an i/o directed loop (clause 10 in the definition of $\mathscr{M}$) or equals *fail* in case of selection (clause 7 in the definition of $\mathscr{M}$). The global nondeterminism is unresolvable only if all addressed processes $P_{k_i},\ldots,P_{k_m}$ have terminated.

$$\bigcup\{\textit{deadlock}\mid\text{if none of the other clauses is applicable}\}.\tag{10}$$

This case arises when a group of processes are involved in some cyclic communication, while all the rest have terminated. This is a deadlock state, and is recorded as such in the value of $\mathscr{B}$.

Note that we are able to detect a *nondeterministically possible* deadlock state. Compare Milne and Milner [13].

In the example at the end of the last section, one would get

$$\mathscr{B}(\mathscr{M}[\![P_1]\!](s),\mathscr{M}[\![P_2]\!](s'))=\{\langle(s)_1^\infty,s'\rangle\}$$

whereas

$$\mathscr{B}(\mathscr{M}[\![P_1']\!](s),\mathscr{M}[\![P_2]\!](s'))=\{\langle(s)_1^\infty,s'\rangle,\textit{deadlock}\}.$$

2. The first thing that has to be done is to show that if the equation is written as $\mathscr{B}=\tau(\mathscr{B})$, the functional τ is continuous in $\mathscr{B}$. Let $\langle\mathscr{B}^i\rangle_{i=0}^\infty$ be a sequence of partial functions from $\mathscr{T}_1\times\cdots\times\mathscr{T}_n$ to $P_{\mathrm{E-M}}(S_1\times\cdots\times S_n\cup\{\bot,\textit{fail},\textit{deadlock}\})$, $\mathscr{B}_0\sqsubseteq\mathscr{B}_1\sqsubseteq\cdots\sqsubseteq\mathscr{B}_i\sqsubseteq\cdots$, and $\mathscr{B}^\infty=\text{l.u.b.}_i\mathscr{B}^i$; then $\tau(\mathscr{B}^\infty)(T_1,\ldots,T_n)$ is obtained by replacing all occurrences of $\mathscr{B}$ in τ by $\mathscr{B}^\infty$; from $\mathscr{B}_0\sqsubseteq\mathscr{B}_1\sqsubseteq\cdots\sqsubseteq\mathscr{B}_i\sqsubseteq\cdots$ it follows that $\mathscr{B}^\infty(T_1',\ldots,T_n')=\text{l.u.b.}_i\mathscr{B}^i(T_1',\ldots,T_n')$ for arbitrary T_i'; by continuity of $\bigcup$ one obtains $\tau(\mathscr{B}^\infty)(T_1,\ldots,T_n)=\text{l.u.b.}_i\tau(\mathscr{B}^i)(T_1,\ldots,T_n)$.

One of the cardinal principles of denotational semantics being that all semantically meaningful functions are continuous, one would like to show next that $\mathscr{B}$ is continuous in its arguments. $\mathscr{B}$ being the least upper bound of the sequence $\Omega \sqsubseteq \tau(\Omega) \sqsubseteq \tau^2(\Omega) \cdots \sqsubseteq \tau^i(\Omega) \cdots$ (where Ω is the completely undefined function), it is enough to show that $\tau^i(\Omega)$ is continuous for every i; it suffices to show that if A is continuous then so is $\tau(A)$ since Ω is obviously continuous.

The first step is to see that if A is monotone then $\tau(A)$ is monotone. Suppose $T_1 \sqsubseteq T_1'$: Then case analysis shows that if $a \in (S_1 \times \cdots \times S_n) \cup \{\textit{fail}, \textit{deadlock}\}$ and $a \in \tau(A)(T_1, T_2, \ldots, T_n)$, then $a \in \tau(A)(T_1', T_2, \ldots, T_n)$ and if $\perp \in \tau(A)(T_1', T_2, \ldots, T_n)$ then $\perp \in \tau(A)(T_1, T_2, \ldots, T_n)$; the case analysis is tedious but standard, and therefore omitted. The last step is to show that if A is continuous and $T_1^0 \sqsubseteq T_1^1 \sqsubseteq \cdots \sqsubseteq T_1^i \sqsubseteq \cdots$ is an ascending sequence whose l.u.b. is T_1^∞ then, if $a \in (S_1 \times \cdots \times S_n) \cup \{\textit{fail}, \textit{deadlock}\}$ and $a \in \tau(A)(T_1^\infty, T_2, \ldots, T_n)$ then there are a, i such that $a \in \tau(A)(T_1^i, T_2, \ldots, T_n)$ and that if for every i, $\perp \in \tau(A)(T_1^i, T_2, \ldots, T_n)$ then $\perp \in \tau(A)(T_1^\infty, T_2, \ldots, T_n)$. Both properties are checked by case analysis.

We give a detailed proof of one case.

Assume A is continuous, and let $T_1^0 \sqsubseteq T_1^1 \sqsubseteq \cdots \sqsubseteq T_1^i \sqsubseteq \cdots$, $T_1^\infty = \text{l.u.b.}_i(T_1^i)$. Let $y \in \tau(A)(T_1^\infty, T_2, \ldots, T_n)$, where $y = \langle s_1, \ldots, s_n \rangle$. We want to show that $\exists\, i$ s.t. $y \in \tau(A)(T_1^i, T_2, \ldots, T_n)$. From the form of τ's definition, there are six clauses due to which this y could be generated (y is a tuple of final states!). These are clauses 3, 4, 6, 7, 8, 9. Since, by assumption, A is continuous in its arguments (and application and projection are continuous as well), we have that

$$A(T_1^\infty, T_2, \ldots, T_n) = \underset{i}{\text{l.u.b.}}\ A(T_1^i, T_2, \ldots, T_n).$$

By a property of $P_{\text{E-M}}$ of a flat domain D, $d \in \text{l.u.b.}_i\ V_i$ implies that $\exists j.\ d \in V_j$, for $V_i \in P_{\text{E-M}}(D)$. This implies the claim for clauses 4, 6, 7, 8, 9.

For clause 3, we have that

$T_1^\infty = s_1$ (S_1 has no $\perp$!), and therefore $\exists\, i.\ T_1^i = s_1$; again the claim follows.

Similar arguments can be given for the remaining arguments $T_2, \ldots, T_n$.

A Semantic Variant

According to the semantics presented, a possible outcome of a program is the set $\{\perp, \textit{fail}\}$. This represents a nondeterministic situation, where there is a nonending computation and a failing computation. This could be interpreted operationally as terminating (actually aborting) the *whole* concurrent program once a local failure is detected.

An alternate semantics could be that in the presence of nontermination the result is $\{\perp\}$, and any failure is disregarded. In order to achieve this semantics, one has to restrict the application of the "negative" clauses 2 and 5 only if no other, "positive" clause, is applicable.

Acknowledgments

We are grateful for helpful remarks from D. Albert, M. Clint, E. W. Dijkstra, D. Harel, R. Milner, G. Plotkin, and A. Pnueli. Special thanks are due to Robert Milne, who discussed in detail previous drafts and helped to improve both contents and presentation. We are grateful to SRC of the United Kingdom and to NSF of the United States for providing funds for living and traveling.

References

1. J. W. de Bakker, Semantics and termination of nondeterministic recursive programs, *in* "Proceedings, 3rd Coll. Automata, Languages and Programming," Edinburgh Univ. Press, Edinburgh, 1976.
2. P. Brinch-Hansen, The programming language Concurrent Pascal, *IEEE Trans. Software Engrg.* **1**, **2** (1975), 199–207.
3. M. Clint, Program proving: Coroutines, *Acta Informatica* **2**, No. 1 (1973), 50–63.
4. E. W. Dijkstra, "A Discipline Programming," Prentice-Hall, Englewood Cliffs, N. J., 1976.
5. E. W. Dijkstra *et al.*, An elephant inspired by the Dutch National Flag, EWD 608, Burroughs-Nuenen, 1977; see also EWD 607.
6. H. Egli, "A Mathematical Model for Nondeterministic Computations," Technological University, Zurich, 1975.
7. K. R. Apt, N. Francez, and W. P. de Roever, A proof system for communicating sequential processes, *TOPLAS*, submitted for publication.
8. J. Goguen, Abstract errors for abstract data types, *in* "Proceedings, IFIP Working Conference on Formal Description of Programming Concepts, 31 July to 5 August 1977, New Brunswick."
9. C. A. R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (1978).
10. C. A. R. Hoare, Monitors: An operating systems structuring concept, *Comm. ACM.* **17** (1974), 549–557.
11. G. Kahn, The semantics of a simple language for parallel programming, *IFIP*, 1974.
12. D. J. Lehmann and M. B. Smyth, Algebraic specifications of data types: A synthetic approach, *Math. Systems Theory*, in press. (Summary in "Proceedings, 18th Annual Symposium on F.O.C.S. Providence, R. I., Oct. 1977," pp. 7–12.)
13. G. Milne and R. Milner, Concurrent processes and their syntax, *J. Assoc. Comput. Mach.* **26** 2 (1979).
14. R. Milner, Processes: A mathematical model of computing agents, *in* "Logic Colloquium 1973," North-Holland, Amsterdam, 1973.
15. S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, I, *Acta Informatica* **6** (1976), 319–340.
16. G. D. Plotkin, A power domain construction, *SIAM J. Comput.* **5**, No. 3 (September 1976
17. W. P. de Roever, Dijkstra's predicate transformer, nondeterminism, recursion, and termination, *in* "Proceedings, Conference on Mathematical Foundation of Computer Science, 1976," Lecture Notes in Computer Science, Springer-Verlag, New York/Berlin, 1976.
18. D. Scott and C. Strachey, Towards a mathematical semantics for computer languages, *in* "Proceedings, Symposium on Computers and Automata," Microwave Research Institute, 1971.
19. D. Scott, Outline of mathematical theory of computation, *in* "Proceedings, 4th Princeton Conf. on Info. Sci and Sys., 1970."
20. M. Smyth, Power domains, *J. Comput. System Sci.* **16** (1978), 23–36.
21. J. Stoy, "Denotational Semantics of Programming Languages: The Scott-Strachey Approach," MIT Press, Cambridge, Mass., 1977.
22. C. Strachey and R. Milne, "A Theory of Programming Language Semantics," Chapman & Hall, London, 1977.
23. C. A. R. Hoare, "A Model for Communicating Sequential Processes," Oxford, December 1978.