A Theory of

Communicating Sequential Processes.

C.A.R. Hoare, S.D. Brookes, A.W. Roscoe.

Technical Monograph PRG-16
May 1981

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford. OX2 6PE

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford. OX2 6PE

Abstract

A mathematical model for communicating sequential processes is given, and a number of its interesting and useful properties are stated. The possibilities of non-determinism are fully taken into account.

CONTENTS

A THEORY OF COMMUNICATING SEQUENTIAL PROCESSES

0 Introduction.

In the last decade there has been a remarkable growth in general understanding of the design and definition of computer programming languages. This understanding has been based upon a recognition that the text of each program expressed in the language should be given a mathematically defined meaning or denotation, in the same way as any other notational system of logic or mathematics. For a conventional sequential programming language, the simplest mathematical domain suitable for this purpose is the space of partial functions which map from an abstract machine state before execution of a command to the state of the machine afterwards. For a programming language with jumps, the appropriate mathematical domain is slightly more complex, involving continuations. For a programming language in which subprograms are themselves assignable components of the abstract machine state, the appropriate reflexive domain of continuous functions has been discovered by Dana Scott [4]. His techniques have been applied to a variety of familiar and novel programming languages. The concept on which all these developments rest is the familiar mathematical concept of a partial function; and its familiarity has undoubtedly contributed to the widespread acceptance and success of the approach. However, there are two features of certain new experimental programming languages involving concurrency which are not so simply treated as mathematical functions.

(1) In the parallel execution of commands of a program, the effect of each command can no longer be modelled as a function from an initial state to a final state of an abstract machine; it is also necessary also to model the continuing interactions of a command with its environment.

(2) In the execution of parallel programs, it is desirable to abstract from the relative rates of progress of the commands being executed in parallel. In general, this will give rise to non-determinacy in the behaviour and outcome of the program.

Both these problems arise in acute form in the treatment of a language like that of Communicating Sequential Processes [1].

It is the purpose of this monograph to construct a mathematical domain which should play the same role in defining the semantics of communicating processes as the domain of partial functions does for sequential and deterministic programming languages. Every effort has been made to keep the domain simple, and to ensure that the necessary operators over objects in the domain have elegant and intuitively valid properties.

The first section of the monograph contains a definition of the required domain of processes. Following the lead of [2], we first introduce the concept of a *transition* which is a ternary relation between

(1) the *initial* state of a process

(2) a sequence describing its interactions with its environment *during* its execution

(3) a possible state of the process *after* those interactions.

Next, we note that the internal states of a process are not observable from its environment. We therefore define the concept of an *observation* of a process, which is a finitely describable experiment to which a process

can be subjected. We then postulate that two processes are identical if they cannot be distinguished by any such finite observation. This reasoning leads directly to the construction of our proposed mathematical space of processes.

The next section shows that this space has the usual ordering properties required of a semantic domain. The relevant partial ordering is simply set inclusion in the *reverse* of the normal direction, so that one process is an approximation to another if it is *less* deterministic. We show that this is a chain-complete partial ordering over the space of all processes. The important consequence of this is that every set of recursive equations in process-valued variables has a least solution; and this permits the use of recursion both in a programming language and in its formal definition.

The third section defines a wide range of operators over the domain of processes; these include sequential composition, conditional composition, two forms of parallel composition, and (perhaps the most crucial) a concealment operator, which permits abstraction from the details of internal communications between processes connected in a network. These operators enjoy a number of elegant and useful algebraic properties. We hope that this range of defined operators will be a sufficient basis in terms of which to define all other operations required in the semantics of a parallel programming language, without any further concern for the details of the underlying mathematical model. Thus these operators should play the same role as the basic operators defined by Scott for the LAMBDA calculus, which shield the practising user of the calculus from the complexities of the construction of the underlying domain.

The fourth section gives some examples of the application of the model, by showing how it can be used to define some complex but useful programming language constructs, and to describe some simple but interesting parallel algorithms.

The fifth section discusses the prospects for the development of formal methods in increasing reliability of implementation and use of a programming language which includes parallelism.

1 Definition of a process

The ultimate unit in the behaviour of a process is an *event*. Events are regarded as instantaneous: if we wish to represent an activity with duration, we must introduce two events to represent its start and its finish, so that other events can occur between them. We shall not be interested in the length of the time interval which separates the events, but only in the relative order in which they occur. We let A stand for the set of all events with which we shall be concerned. The behaviour of a process up to some moment in time can be recorded as the sequence of all events in which it has participated; this is known as a *trace* and the set of all possible traces is denoted by A*.

Let s be a trace and let P and Q be processes. A *transition* is a proposition

$$P \xrightarrow{s} Q$$

which means that s is a possible trace of the behaviour of P up to some moment in time, and that the subsequent behaviour of P may be the same as that of Q. Thus if t is a possible trace of Q, after which it may behave like R, then clearly st (s followed by t) is also a possible trace of P, after which it can behave like R.

This fact is formalised as a general *law*:

$$P \xrightarrow{s} Q \;\&\; Q \xrightarrow{t} R \Rightarrow P \xrightarrow{st} R. \qquad \text{(L1)}$$

Conversely, if $P \xrightarrow{st} R$, then there must exist some intermediate process Q, which behaves exactly like P would behave after doing s but before starting on t. This is expressed in the law:

$$P \xrightarrow{st} R \Rightarrow \exists\, Q.\ P \xrightarrow{s} Q \;\&\; Q \xrightarrow{t} R. \qquad \text{(L2)}$$

The empty trace <> is the sequence with no events. It describes the behaviour of a process which has not yet engaged in any externally recordable event. We adopt the convention that after doing nothing a process may remain unchanged:

$$P \xrightarrow{<>} P. \qquad (L3)$$

If Q ≠ P, then the possibility of the transition $P \xrightarrow{<>} Q$ means that P may make internal progress, which cannot be observed from outside, after which it can behave like Q rather than P. Since a process is in general nondeterministic, its internal progress will require making of arbitrary choices, which are wholly uncontrollable and invisible from outside. Such a choice can only reduce the range of possible future behaviours of P, by excluding behaviours which would have remained possible if some alternative choice had been taken. This fact is expressed by the theorem:

$$P \xrightarrow{<>} Q \ \& \ Q \xrightarrow{s} R \Rightarrow P \xrightarrow{s} R.$$

The *initials* of a process P are those events in which it can engage on its very first step; they are defined

$$\text{initials}(P) = \{a \mid \exists\ Q.\ P \xrightarrow{<a>} Q\}$$

where <a> is the sequence containing only "a". The choice of which of these events will actually occur will depend (at least in part) on the environment in which the process is placed. Let X be the set of events which are possible for that environment. Then the event that actually occurs must be in the intersection (X ∩ initials(P)). If this intersection is empty, then nothing further can happen; the process and its environment remain locked forever in deadly embrace [3]. Unfortunately, if P is non-deterministic, deadly embrace is still possible even when the intersection is non-empty. This occurs when P can progress invisibly to become Q, and the intersection (X ∩ initials(Q)) is empty. In such

a case, we say that X is a possible *refusal* of P, and that P can *refuse* X.

We want to be able to distinguish between processes by observing their behaviour in *finite* environments. It will be possible to distinguish between P and Q if and only if there is a finite sequence s of events possible for P and Q, and a *finite* set of events X, such that P can refuse X after doing s but Q cannot (or *vice versa*). We adopt this view of distinguishability because we consider a *realistic* environment to be one which is at any time capable of performing only a finite number of events. Bearing these remarks in mind, we define the set of all P's refusals as:

$$\begin{aligned}\text{refusals}(P) = \{X \mid\ & X \text{ is finite } \& \\ & \exists\ Q.\ P \xrightarrow{<>} Q\ \& \\ & (X \cap \text{initials}(Q) = \{\})\}\end{aligned}$$

where { } is the empty set.

From this definition it follows that

(1) { } ∈ refusals(P)

(2) if Y ∈ refusals(P) and X ⊆ Y then X ∈ refusals(P).

(3) if X ∈ refusals(P) and Y is a finite subset of (A − initials(P)) then (X ∪ Y) ∈ refusals(P).

(A − initials(P)) is the set of events that P *cannot* perform. The third theorem above states that P can refuse these events, together with any other set of events it can refuse.

A *trace* of a process is a sequence of events in which it may engage up to some moment in time. The set of all such traces is defined:

$$\text{traces}(P) = \{s \mid \exists\ Q.\ P \xrightarrow{s} Q\}.$$

From this definition it follows that

$$\langle\rangle \in traces(P)$$

$$st \in traces(P) \Rightarrow s \in traces(P).$$

The second theorem states that any prefix (i.e. initial subsequence) of a trace of P is also a trace of P.

If s is a trace of P, and if, after engaging in the events of s, P can refuse the finite set X, we say that the pair (s, X) is a *failure* of the process P. The set of all such failures is defined:

$$failures(P) = \{(s,X) \mid \exists Q.\ P \xrightarrow{s} Q\ \& \\ X \subseteq refusals(Q)\}.$$

From this definition it follows that the set F = failures(P) has the properties:

(P1) $(s,X) \in F \Rightarrow s \in A^* \ \&\ X \subseteq A \ \&\ X$ is finite

(P2) $(\langle\rangle,\{\}) \in F$

(P3) $(st,\{\}) \in F \Rightarrow (s,\{\}) \in F$

(P4) $X \subseteq Y \ \&\ (s,Y) \in F \Rightarrow (s,X) \in F$

(P5) Let $U = \{a \mid (s\langle a\rangle,\{\}) \in F\}$ and let Y be a finite subset of $(A - U)$; then
$$(s, X) \in F \Rightarrow (s, (X \cup Y)) \in F.$$

The failures of a process represent possible externally observable aspects of its behaviour. The fact that (s, X) ∈ failures(P) means that it is possible for P to do s and then refuse to do any more, in spite of the fact that

its environment allows any of the events of X. Our next postulate states that there exists a process corresponding to any possible set of failures.

If F satisfies the five properties of the previous paragraph then there exists a process P such that failures(P) = F. (L4)

Finally, we postulate that the failures of a process are the only externally observable aspects of its behaviour. Thus two processes that fail in exactly the same circumstances are indistinguishable by external observation. Since we deliberately choose to ignore the details of the internal construction of processes, it is reasonable to adopt the principle of identity of indiscernables:

$$failures(P) = failures(Q) \Rightarrow P = Q. \qquad (L5)$$

Postulates L4 and L5 together state that a process is uniquely defined by its failure set. In future, we shall *identify* a process with its failure set, and *define* the transition relation thus:

$$P \xrightarrow{s} Q = (\forall t, X.\ (t, X) \in Q \Rightarrow (st, X) \in P).$$

From this definition we deduce (using conditions P1 - P5):

$$P \xrightarrow{st} Q \equiv \exists R\ (P \xrightarrow{s} R \ \&\ R \xrightarrow{t} Q)$$

$$P \xrightarrow{\langle\rangle} Q \equiv Q \subseteq P$$

$$traces(P) = \{s \mid (s,\{\}) \in P\}$$

$$initials(P) = \{a \mid (\langle a\rangle,\{\}) \in P\}$$

$$refusals(P) = \{X \mid (\langle\rangle,X) \in P\}$$

$$failures(P) = P.$$

Since transitions can be defined in terms of failure sets and failure sets in terms of transitions, it is permissible to use either method in the definition of any particular process. It will be found convenient to give an intuitive explanation of the intended behaviour of a process by giving the laws governing its transitions, followed by a formal definition in terms of refusal sets. Usually, the laws will give only sufficient conditions for the transitions of the process being defined. Then the formal definition will specify the *smallest* refusal set which satisfies the laws; i.e., the one with the least failures.

(1) The simplest process is STOP, a process that never does anything. The only law which it obeys is:

$$STOP \xrightarrow{\langle\rangle} STOP.$$

The process that obeys only this law is defined:

$$STOP = \{(\langle\rangle, X) \mid X \subseteq A \;\&\; X \text{ is finite}\}.$$

Clearly, it refuses to do whatever the environment may offer.

(2) If Q is a process and "a" is an event, then the process (a → Q) is a process which first does "a" and then behaves like Q:

$$Q \xrightarrow{s} R \Rightarrow (a \to Q) \xrightarrow{\langle a\rangle s} R.$$

We also permit Q to make internal progress while waiting for "a"

$$Q \xrightarrow{\langle\rangle} Q' \Rightarrow (a \to Q) \xrightarrow{\langle\rangle} (a \to Q').$$

The smallest process which satisfies these laws is:

$$\begin{aligned}(a \to Q) = {} & \{(\langle\rangle, X) \mid X \subseteq (A - \{a\}) \;\&\; X \text{ is finite}\} \\ & \cup \{(\langle a\rangle s, X) \mid (s, X) \in Q\}.\end{aligned}$$

Clearly, it cannot refuse to do "a" if offered; but may (indeed must) refuse everything else.

Examples: $P_a \triangleq (a \to STOP)$

$P_b \triangleq (b \to STOP)$.

(3) Let B be a subset of A, and let F(x) be a process for each x in B. Then (x:B → F(x)) is a process which first does any event x in B and then behaves like F(x).

$$F(b) \xrightarrow{s} R \Rightarrow (x{:}B \to F(x)) \xrightarrow{\langle b\rangle s} R) \text{ for all b in B.}$$

$$(\forall x.\; x \in B \Rightarrow (F(x) \xrightarrow{\langle\rangle} F'(x))) \Rightarrow (x{:}B \to F(x)) \xrightarrow{\langle\rangle} (x{:}B \to F'(x)).$$

The smallest definition satisfying these laws is

$$\begin{aligned}(x{:}B \to F(x)) = {} & \{(\langle\rangle, X) \mid X \subseteq A - B \;\&\; X \text{ is finite}\} \\ & \cup \{(\langle x\rangle s, X) \mid x \in B \;\&\; (s, X) \in F(x)\}.\end{aligned}$$

Note that x is a bound variable of this construction, so that

$$(x{:}B \to F(x)) = (y{:}B \to F(y)).$$

Example: $P_{ab} = (x:\{a, b\} \longrightarrow STOP)$.

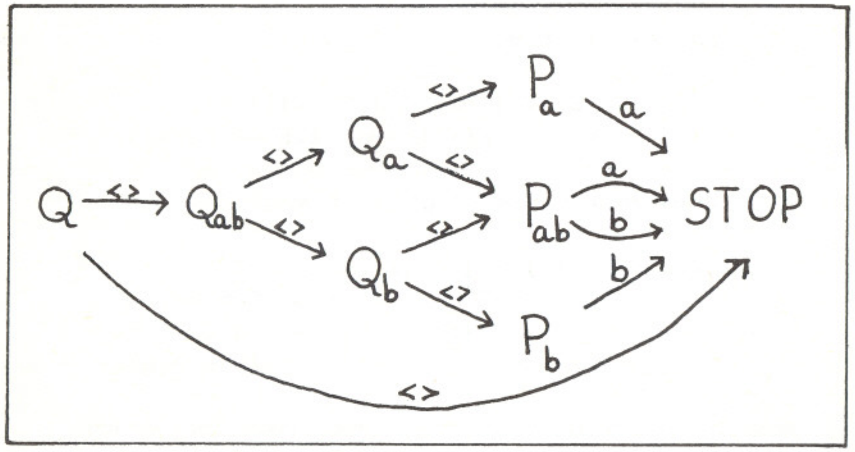
(4) Let $Q_a \cong P_a \cup P_{ab}$

$Q_b \cong P_b \cup P_{ab}$

$Q_{ab} \cong P_a \cup P_b$

$Q \cong Q_{ab} \cup STOP$.

Figure 1 shows the transitions between these processes (other than those deducible by transitivity).



$$\begin{tikzcd}
 & & & P_a \arrow[rd, "a"] & \\
 & & Q_a \arrow[ru, "<>"] \arrow[rd, "<>"] & & \\
Q \arrow[r, "<>"] \arrow[rrrr, bend right, "<>"] & Q_{ab} \arrow[ru, "<>"] \arrow[rd, "<>"] & & P_{ab} \arrow[r, bend left, "a"] \arrow[r, bend right, "b"] & STOP \\
 & & Q_b \arrow[ru, "<>"] \arrow[rd, "<>"] & & \\
 & & & P_b \arrow[ruu, "b"] &
\end{tikzcd}$$

Figure 1

If $A = \{a,b\}$, figure 2 shows the initials and refusals of each of these processes, proving that they are distinct.

process	initials	refusals
Q	$\{a,b\}$	$\{\},\{a\},\{b\},\{a,b\}$
Q_{ab}	$\{a,b\}$	$\{\},\{a\},\{b\}$
Q_a	$\{a,b\}$	$\{\},\{b\}$
Q_b	$\{a,b\}$	$\{\},\{a\}$
P_{ab}	$\{a,b\}$	$\{\}$
P_a	$\{a\}$	$\{\},\{b\}$
P_b	$\{b\}$	$\{\},\{a\}$
STOP	$\{\}$	$\{\},\{a\},\{b\},\{a,b\}$

Figure 2.

(5) RUN is a process which will always do anything offered by the environment. Thus it satisfies the laws:

$RUN \xrightarrow{s} RUN \quad$ for all s in A^*.

The required definition is

$$RUN = \{(s,\{\}) \mid s \in A^*\}$$

Clearly, RUN can never refuse anything.

(6) CHAOS is a process that can do anything at all; but in contrast to RUN, it can also at any time refuse to do anything at all.

$$CHAOS \xrightarrow{s} STOP \quad \text{for all } s \text{ in } A^*.$$

The required definition is

$$CHAOS = \{(s,X) \mid s \in A^* \ \&\ X \subseteq A \ \&\ X \text{ is finite}\}.$$

2 Nondeterminism.

This section investigates the properties of nondeterminism. It uses the methods of lattice theory to show how every recursive equation uniquely defines a process; the mathematics required is not difficult, and is fully explained.

2.1 Nondeterministic composition.

If P and Q are processes, the combination $(P \sqcap Q)$ is a process which behaves exactly like P or like Q; but the choice between them is wholly nondeterministic; it is made autonomously by the process (or by its implementor), and cannot be influenced or even observed by the environment. Thus $(P \sqcap Q)$ can do (or refuse to do) everything that P or Q can:

$$P \xrightarrow{s} R \vee Q \xrightarrow{s} R \Rightarrow (P \sqcap Q) \xrightarrow{s} R.$$

The smallest process which satisfies this law is simply:

$$P \sqcap Q = P \cup Q.$$

This operation is clearly associative, commutative, and idempotent, and has CHAOS as its zero:

$$(P \sqcap Q) \sqcap R = P \sqcap (Q \sqcap R) \qquad \text{(associative)}$$

$$(P \sqcap Q) = (Q \sqcap P) \qquad \text{(commutative)}$$

$$(P \sqcap P) = P \qquad \text{(idempotent)}$$

$$CHAOS \sqcap P = CHAOS \qquad \text{(zero)}$$

2.2 Distributivity.

One of the main reasons for specifying a nondeterministic process such as $(P \sqcap Q)$ is to allow an implementor the freedom to select and implement either P or Q, whichever of them is cheaper, or gives better performance. Suppose F is some function from processes to processes. "F( )" may be regarded as an assembly with a vacant slot into which an arbitrary component may be plugged, e.g. F(P) or F(Q). The behaviour of the assembly is then a function of the behaviour of this component. Suppose that an implementor has to implement $(F(P) \sqcap F(Q))$. The straightforward way of doing this is to implement F(P) *and* F(Q) and then select between them. An alternative way is first to select the component, and plug in just that one. This alternative is the same as the standard way of implementing $F(P \sqcap Q)$. We would like to ensure that both implementations give the same result, i.e.

$$F(P \sqcap Q) = F(P) \sqcap F(Q).$$

A function F which satisfies this condition for all processes P and Q is said to be *distributive*. Another reason for preferring distributive functions is that they simplify proofs of the properties of processes, by case analysis of the alternative behaviours.

As an example, the construction $(a \rightarrow P)$ is distributive in P, since:

$$(a \rightarrow (P \sqcap Q)) = (a \rightarrow P) \sqcap (a \rightarrow Q).$$

This means that there is no discernible difference whether the choice between P and Q is made before or after the occurrence of "a". A function of two or more arguments is distributive if it is distributive in

each argument separately. Thus nondeterministic composition is itself distributive, because

$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R)$$

$$\text{and} \quad (Q \sqcap R) \sqcap P = (Q \sqcap P) \sqcap (R \sqcap P).$$

Further, the construction $(x{:}B \rightarrow F(x))$ is distributive in F(x) for all x in B:

$$(x{:}B \rightarrow (F(x) \sqcap G(x))) = (x{:}B \rightarrow F(x)) \sqcap (x{:}B \rightarrow G(x)).$$

Thus all operators introduced so far are distributive, and we shall make this a requirement for all operators introduced hereafter.

2.3 Limits.

The relation $P \xrightarrow{<>} Q$ means that the process P may, as a result of internal progress, transform itself automatically to the process Q. A *chain* of processes is an infinite sequence $\{P_i \mid i \geqslant 0\}$, each of which may transform itself into its successor; thus it satisfies the laws:

$$P_i \xrightarrow{<>} P_{i+1} \qquad \text{for all } i.$$

For each such chain there exists a *limit* process $(\bigsqcup_i P_i)$, which can do (or refuse) anything that *every* member of the chain can do (or refuse):

$$(\forall i.\ P_i \xrightarrow{<>} P_{i+1} \ \&\ P_i \xrightarrow{s} Q) \Rightarrow (\bigsqcup_i P_i) \xrightarrow{s} Q.$$

The definition of the smallest process which satisfies these laws is:

$$(\bigsqcup_i P_i) = \bigcap_i P_i \quad \text{provided that } \forall i \,.\, P_i \xrightarrow{<>} P_{i+1}$$

This operator is distributive:

$$\bigsqcup_i (P_i \sqcap Q_i) = (\bigsqcup_i P_i) \sqcap (\bigsqcup_i Q_i)$$

provided that $\{P_i | i \geqslant 0\}$ and $\{Q_i | i \geqslant 0\}$ are chains.

Furthermore:

$$P_i \xrightarrow{<>} (\bigsqcup_i P_i) \text{ for all i.}$$

and for all processes Q

$$(\forall i.\ P_i \xrightarrow{<>} Q) \Rightarrow (\bigsqcup_i P_i) \xrightarrow{<>} Q.$$

The relation $P \xrightarrow{<>} Q$ means simply that the set Q is contained in the set P. Thus everything that Q can *do* so can P

$$\text{traces}(Q) \subseteq \text{traces}(P),$$

and everything that Q can *refuse* so can P

$$\text{refusals}(Q) \subseteq \text{refusals}(P).$$

In other words P differs from Q only in that it is less deterministic, and that Q can result from P by resolution of some of P's inherent nondeterminism. Thus if $\forall i.\ P_i \xrightarrow{<>} P_{i+1}$, this can mean that there is a potential infinity of nondeterministic decisions to be taken; but perhaps *none* of them will actually reach the limit $(\bigsqcup_i P_i)$. Thus $(\bigsqcup_i P_i)$ can be regarded as an "ideal" element, of which the P_i are an ever improving sequence of approximations, getting as close as we may wish to the ideal, but never actually reaching it. However, in implementing $(\bigsqcup_i P_i)$ we wish

to allow an implementor (if he wishes) to take *all* the non-deterministic choices in advance of delivering his product.

2.4 Continuity.

Let F be a distributive function from processes to processes, and let $\{P_i | i \geqslant 0\}$ be a chain. Then F is *monotonic* in the sense that

$$P \xrightarrow{<>} Q \Rightarrow F(P) \xrightarrow{<>} F(Q)$$

for all P and Q. Suppose now that an implementor is faced with the task of implementing $F(\bigsqcup_i P_i)$. The straightforward method would be to obtain the limit $(\bigsqcup_i P_i)$ and then "plug" it into the assembly F( ). But suppose that the limit $(\bigsqcup_i P_i)$ is in some sense unattainable. Then we can apply F to each of the approximations P_i, obtaining the chain $\{F(P_i) | i \geqslant 0\}$, and then take the limit of that. We would like to be sure that both implementations are the same:

$$\bigsqcup_i F(P_i) = F(\bigsqcup_i P_i).$$

Then, even if the limit $\bigsqcup_i F(P_i)$ is unattainable, we can be sure of getting as close to it as we need by the sequence of approximations $F(P_i)$. If this condition holds for all chains, then F is said to be *continuous*. Another good reason for preferring continuous functions is that they simplify proofs of the properties of processes. A third reason will be explained in the next section.

As an example, the construction $(a \rightarrow P)$ is continuous in P, since

$$(a \rightarrow (\bigsqcup_i P_i)) = \bigsqcup_i (a \rightarrow P_i).$$

A function of two or more arguments is continuous if it is continuous in each argument separately. Thus nondeterministic composition is continuous, because

$$(\sqcup_i P_i) \sqcap Q = \sqcup_i (P_i \sqcap Q)$$

and $Q \sqcap (\sqcup_i P_i) = \sqcup_i (Q \sqcap P_i)$

provided that $\{P_i \mid i \geqslant 0\}$ is a chain.

Furthermore, the construction $(x{:}B \rightarrow F(x))$ is continuous in $F(x)$ for all x in B:

$$(x{:}B \rightarrow (\sqcup_i F_i(x))) = \sqcup_i (x{:}B \rightarrow F_i(x)).$$

Finally, the limit construction is itself continuous:

$$\sqcup_i (\sqcup_j P_{ij}) = \sqcup_j (\sqcup_i P_{ij})$$

provided that for all i, $\{P_{ij} \mid j \geqslant 0\}$ is a chain, and for all j, $\{P_{ij} \mid i \geqslant 0\}$ is a chain.

Thus all operators introduced so far are continuous, and we shall make this a requirement for all operators introduced hereafter. This will ensure that any expression composed from named components by applying continuous operators will also be continuous in each of its named components.

2.5 Recursion.

Let F be a continuous function from processes to processes. We define the n-fold composition of F by induction on n:

$$F^0(p) = p$$

$$F^{n+1}(p) = F(F^n(p)).$$

Since F is continuous, it is also monotonic, so the set

$$\{F^n(\mathrm{CHAOS}) \mid n \geqslant 0\}$$

constitutes a chain; and its limit is defined

$$\mu p.\ F(p) = \sqcup_n F^n(\mathrm{CHAOS}).$$

Note that in this construction, "p" plays the role of a bound variable, so that:

$$\mu p.\ F(p) = \mu q.\ F(q).$$

Let p be a variable standing for an "unknown" process, which is known only to satisfy the equation:

$$p = F(p).$$

Provided that F is continuous, it is clear that $\mu p.F(p)$ is a solution for p in this equation. Furthermore, it is the most general solution, in the sense that it can progress autonomously to every other solution:

$$Q = F(Q) \Rightarrow \mu p.F(p) \xrightarrow{<>} Q.$$

Thus the equation

$$p = F(p)$$

can be regarded as a *recursive* definition of the process $\mu p.F(p)$; for example, we could have defined

$$\mathrm{RUN} = (\mu p.(x{:}A \rightarrow p))$$
$$\mathrm{RUN}_B = (\mu p.(x{:}B \rightarrow p)) \quad \text{for any } B \subseteq A.$$

A similar construction can be used to find the solution of mutually recursive equations such as

$$p = F(p,q)$$
$$q = G(p,q)$$

even when the number of equations is infinite.

The desire to define processes freely by recursion is one of the major motives for requiring all operators to be continuous.

3 Operators on processes.

In this section we define the most important primitive operators on processes, and state their chief properties. The section is sadly devoid of examples: these will be found in the next section.

3.1 Parallel composition by intersection.

The combination $(P \| Q)$ is intended to behave like both P and Q, progressing in parallel. Thus an event can occur only when both P and Q are able to participate in it simultaneously. The same is therefore true of sequences of events:

$$P \xrightarrow{s} P' \ \& \ Q \xrightarrow{s} Q' \Rightarrow (P \| Q) \xrightarrow{s} (P' \| Q')$$

The smallest process which satisfies this law is defined:

$$(P \| Q) = \{(s, X \cup Y) \mid (s,X) \in P \ \& \ (s,Y) \in Q\}.$$

Thus $(P \| Q)$ can refuse a set if P can refuse some of it and Q can refuse the rest.

The operator $\|$ is distributive, continuous, associative and commutative. It has STOP as its zero and RUN as its unit, i.e.,

$$(P \| \mathrm{STOP}) = \mathrm{STOP} \quad \text{and} \quad (P \| \mathrm{RUN}) = P.$$

Furthermore:

$$(x{:}B \rightarrow F(x)) \| (y{:}C \rightarrow G(y)) =$$
$$(z{:}(B \cap C) \rightarrow (F(z) \| G(z))).$$

3.2 Conditional composition.

The process $(P \square Q)$ behaves either like P or like Q; but it differs from $(P \sqcap Q)$ in that the choice between them can be influenced by the environment on the very first step. If the environment offers an event "a" which is possible for P but not for Q, then P is selected; and conversely for Q; but if "a" is possible for both P and Q, the selection between them is nondeterminate, and the environment does not get a second chance to influence it. Thus

$$P \xrightarrow{\langle a\rangle s} R \vee Q \xrightarrow{\langle a\rangle s} R \Rightarrow (P \square Q) \xrightarrow{\langle a\rangle s} R$$

Before occurrence of the first event, P and Q may progress independently:

$$P \xrightarrow{\langle\rangle} P' \;\&\; Q \xrightarrow{\langle\rangle} Q' \Rightarrow (P \square Q) \xrightarrow{\langle\rangle} (P' \square Q')$$

The least process which satisfies these laws is defined:

$$\begin{aligned}(P \square Q) = \{&(\langle\rangle,X) \mid (\langle\rangle,X) \in P \;\&\; (\langle\rangle,X) \in Q\}\\ \cup\ \{&(s,X) \mid s \neq \langle\rangle \;\&\\ &((s,X) \in P \vee (s,X) \in Q)\}.\end{aligned}$$

$(P \square Q)$ refuses a set if and only if it is refused by both P and Q.

The operator $\square$ is distributive, continuous, associative, commutative, and idempotent. It has unit STOP. Furthermore it admits distribution thus:

$$P \sqcap (Q \square R) = (P \sqcap Q) \square (P \sqcap R)$$

$$\begin{aligned}&(z{:}(B \cup C) \rightarrow H(z)) =\\ &\quad (x{:}B \rightarrow F(x)) \square (y{:}C \rightarrow G(y))\\ &\text{where } H(z) = \textbf{if } z \in (B-C) \textbf{ then } F(z)\\ &\qquad\qquad\quad \textbf{else if } z \in (C-B) \textbf{ then } G(z)\\ &\qquad\qquad\qquad\quad \textbf{else } F(z) \sqcap G(z).\end{aligned}$$

When F = G, this last theorem is much more simply expressed:

$$\begin{aligned}&(x{:}B \cup C \rightarrow F(x)) =\\ &\quad (x{:}B \rightarrow F(x)) \square (x{:}C \rightarrow F(x)).\end{aligned}$$

3.3 Parallel composition by interleaving.

The process $(P \,|||\, Q)$ behaves like P and Q operating in parallel; but it differs radically from $(P \,||\, Q)$ in that each event requires participation of only one of the processes rather than both. Thus each trace of $(P \,|||\, Q)$ is an interleaving of a trace of P and a trace of Q, as stated in the law

$$P \xrightarrow{s} P' \;\&\; Q \xrightarrow{t} Q' \Rightarrow (P \,|||\, Q) \xrightarrow{r} (P' \,|||\, Q')$$

where r is an interleaving of s and t.

The smallest process which satisfies this law is

$$\begin{aligned}(P \,|||\, Q) = \{(r,X) \mid \exists s,t.\ &(s,X) \in P \;\&\; (t,X) \in Q \;\&\\ &r \text{ is an interleaving of } s \text{ and } t\}\end{aligned}$$

$(P \,|||\, Q)$ can refuse a set only if both P and Q refuse it.

The operator $|||$ is distributive, continuous, associative, and commutative. It has unit STOP and zero RUN.

Furthermore, if $P = (x{:}B \rightarrow F(x))$ and $Q = (y{:}C \rightarrow G(y))$ then

$$\begin{aligned}(P \,|||\, Q) = \;&((x{:}B \rightarrow (F(x) \,|||\, Q))\\ &\square\ (y{:}C \rightarrow (P \,|||\, G(y)))).\end{aligned}$$

Thus if an event can be performed by both processes, it is nondeterministic which of them actually performs it.

3.4 Sequential Composition.

Let "√" denote an event which we interpret as successful termination of a process. Then SKIP is defined as a process which does nothing but terminate successfully:

$$\mathrm{SKIP} = (\surd \to \mathrm{STOP}).$$

The process (P;Q) behaves like P until P terminates successfully, after which it behaves like Q. However, the occurrence of the "√" at the end of P does not appear in any trace of (P;Q); "√" occurs automatically without the knowledge or participation of the environment. Thus, if s does *not* contain "tick", we formulate the laws:

$$P \xrightarrow{s} P' \;\&\; Q \xrightarrow{\langle\rangle} Q' \Rightarrow (P;Q) \xrightarrow{s} (P';Q')$$

$$P \xrightarrow{s\langle\surd\rangle} P' \;\&\; Q \xrightarrow{t} R \Rightarrow (P;Q) \xrightarrow{st} R.$$

The definition which satisfies these laws is

$$\begin{aligned}(P;Q) = \{(s,X) \mid\ & s \text{ does not contain } \surd \;\& \\ & (s,\ X \cup \{\surd\}) \in P\} \\ \cup\ \{(st,X) \mid\ & s \text{ does not contain } \surd \;\& \\ & (s\langle\surd\rangle,\ \{\}) \in P \;\&\; (t,X) \in Q\}.\end{aligned}$$

This definition shows that while P is still running, (P;Q) cannot refuse X unless P can *also* refuse to terminate successfully.

In general, it is a useful convention that "√" should be used *only* in the process SKIP. In particular, in the construction (x:B → F(x)), the set B should never contain "√"; and in future we shall assume that this convention is observed.

Sequential composition is distributive, continuous, and associative. Furthermore:

$$(\mathrm{SKIP};P) = P$$

$$(\mathrm{STOP};P) = \mathrm{STOP}$$

$$(x{:}B \to F(x));P = (x{:}B \to (F(x);P))$$
$$(\text{since } \surd \notin B)$$

$$(\mathrm{SKIP} \,\square\, Q);P = P \sqcap (P \,\square\, (Q;P)).$$

3.5 Iteration.

The process *P behaves like an infinite sequential composition of the process P:

$$P;P;P;\ \ldots$$

It can be simply defined by recursion:

$$*P = \mu q.\ P;q$$

Iteration has the following properties:

$$(P;\ *P) = *P$$

$$(*(x{:}B \to F(x)));P = *(x{:}B \to F(x))$$
$$(\text{since } \surd \notin B)$$

$$*STOP = STOP$$

$$*SKIP = CHAOS.$$

This last result is the most surprising; it would seem more intuitive that *SKIP should equal STOP. Indeed, it is permitted to implement it as STOP. But in general it is very important to distinguish *SKIP from STOP. For example, an implementation of STOP uses no electricity, whereas an implementation of *SKIP may use an unlimited amount. Since it never interacts with its environment, there is no way of switching it off! Such a process must never be switched on, in any environment. CHAOS is another process that must never be used in any environment. It is not unreasonable to equate such equally useless processes.

It can be argued that the process CHAOS might actually *do* something, whereas intuitively *SKIP cannot. But consider the analogy of an electronic circuit with a race condition. Such a circuit must never be used; but if it is used it may break; and a broken device may behave in any way whatsoever. We allow the same possibility for *SKIP.

A terminating form of iteration can be defined

$$P \textbf{ until } Q = \mu p.(Q \mathbin{\square} (P;\ p)).$$

This repeats P any number of times, possibly ending with a single execution of Q. It has properties:

$$*P = P \textbf{ until } (*P) = P \textbf{ until } STOP$$

$$((x{:}B \to F(x)) \textbf{ until } (y{:}C \to G(y)));P = \\ (x{:}B \to F(x)) \textbf{ until } (y{:}C \to (G(y);P))$$

$$SKIP \textbf{ until } Q = (CHAOS \mathbin{\square} Q).$$

The third result is again surprising; it could be argued that in the implementation of (SKIP **until** Q), the opportunity to behave like Q occurs infinitely often; and it is "unfair" to neglect such an opportunity forever. But it seems impossible to define a notion of "fairness" such that a "fair" process can be distinguished from an "unfair" one by any finite observation. That is why our theory makes no stipulation of fairness, and treats every race condition as a possible cause of breakage.

Some of these problems can be avoided if we insist that * and **until** are used only on processes whose first event cannot be "√"

$$*(x{:}B \to F(x)) = \mu p.(x{:}B \to F(x);\ p)$$

$$(x{:}B \to F(x)) \textbf{ until } (y{:}C \to G(y)) = \\ \mu p.((y{:}C \to G(y)) \mathbin{\square} (x{:}B \to (F(x);p))),$$

The same technique can be used to define a *parallel* iteration, in which each activation of the body of the loop progresses in parallel with all previous activations:

$$**(x{:}B \to F(x)) = \mu p.(x{:}B \to (F(x) \mathbin{|||} p)).$$

Unfortunately, this technique cannot be applied when the same problem arises in the next section.

3.6 Concealment.

Let "b" denote an event (*other* than "√") which is to be regarded as an internal operation of the process P; for example, it may be an interaction between some component processes from which P has been constructed. We wish such events to occur automatically whenever they can, without the participation or even the knowledge of the environment of P. We therefore define (P\b) as the process which behaves like P, except that every occurrence of "b" is removed from its traces; it therefore satisfies the law:

$$P \xrightarrow{s} R \Rightarrow (P\backslash b) \xrightarrow{s\backslash b} (R\backslash b)$$

where s\b is formed from s by removing all occurrences of "b".

For reasons explained in the previous section, if P can engage in an infinite sequence of occurrences of "b", without ever interacting with its environment, then (P\b) equals CHAOS.

$$(\forall n.\ P_n \xrightarrow{\langle b\rangle} P_{n+1}) \Rightarrow P_0\backslash b \xrightarrow{\langle\rangle} \text{CHAOS}.$$

The required definition is:

$$\begin{aligned} P\backslash b = \{&(s\backslash b,\ X) \mid (s,\ X\cup\{b\}) \in P\} \\ \cup\ \{&((s\backslash b)t,\ X) \mid \forall n.\ (sb^n,\ \{\}) \in P\ \& \\ &\qquad (t,\ X) \in \text{CHAOS}\} \end{aligned}$$

where sb^n is s followed by n occurrences of b.

This operation is distributive and continuous, and

$$(P\backslash b)\backslash c = (P\backslash c)\backslash b$$
$$(P\backslash b)\backslash b = P\backslash b.$$

Therefore if B is any finite set of symbols, $\{b_1, b_2, \ldots, b_n\}$, we can define

$$P\backslash B = (\ldots((P\backslash b_1)\backslash b_2)\backslash \ldots\backslash b_n).$$

Other theorems are

$$\text{STOP}\backslash b = \text{STOP}$$

$$\text{RUN}\backslash b = \text{CHAOS}$$

$$\text{CHAOS}\backslash b = \text{CHAOS}$$

$$(b \rightarrow P)\backslash b = P\backslash b.$$

$$(x{:}B \rightarrow F(x))\backslash b = (x{:}B \rightarrow (F(x)\backslash b)) \text{ if } b \notin B.$$

$$\begin{aligned} &((b \rightarrow P)\ \square\ (x{:}B \rightarrow F(x)))\backslash b = \\ &\qquad (P\backslash b) \sqcap ((P\backslash b)\ \square\ (x{:}B \rightarrow (F(x)\backslash b))) \text{ if } b \notin B \end{aligned}$$

3.7 Inverse images.

Let f be any function from events to events. Then $f^{-1}(P)$ is a process which can do "a" whenever P could have done f(a):

$$P \xrightarrow{f(s)} Q \Rightarrow f^{-1}(P) \xrightarrow{s} f^{-1}(Q)$$

where f(s) is formed by applying f to each symbol of s.

The required definition is:

$$f^{-1}(P) = \{(s,\ X) \mid (f(s),\ f(X)) \in P\ \&\ X \text{ is finite}\}$$

where $f(X) = \{f(x) \mid x \in X\ \&\ x$ is the domain of $f\}$

f^{-1} is distributive and continuous; furthermore

$$f^{-1}(g^{-1}(P)) = (g \circ f)^{-1}(P)$$

$$f^{-1}(STOP) = STOP$$

$$f^{-1}(RUN) = RUN_{f^{-1}(A)}$$

$$f^{-1}(x{:}B \to F(x)) = (y{:}f^{-1}(B) \to f^{-1}(F(f(y))))$$

f^{-1} distributes through $\Box$, $\|$, $|||$, and ; (provided $f^{-1}(\surd) = \{\surd\}$) and

$$f^{-1}(P \backslash B) = f^{-1}(P) \backslash f^{-1}(B)$$
where $f^{-1}(X) = \{y \mid f(y) \in X\}$

provided each element of X is in the range of f.

4 Applications.

In this section we give a number of examples of the use of the operators defined above in the description of simple processes. In each case, we use laws about transitions to specify the required behaviour of a process before constructing it.

4.1 A COUNT register.

A COUNT is a process which behaves like an unbounded non-negative integer register, with initial value zero. It engages in three kinds of event:

"up"	denotes incrementation of the register and can occur at any time.
"down"	denotes decrementation of the register and cannot occur when its value is zero.
"iszero"	can occur only when the value is zero.

Thus the behaviour of COUNT is specified by the law

$$\begin{aligned} COUNT \xrightarrow{s} Q \Rightarrow\ & EQ \mathbin{\&} initials\ (Q) = \{up, iszero\} \vee \\ & LESS \mathbin{\&} initials\ (Q) = \{up, down\} \end{aligned}$$

where EQ means the number of "up"s and "down"s in s are equal and LESS means there are less "down"s than "up"s in s.

A simple definition of a process $COUNT_0$, which satisfies these laws, can be given by infinite mutual recursion. The process $COUNT_n$ defines the behaviour of a count register holding the value n.

$$COUNT_0 = (\mathrm{iszero} \rightarrow COUNT_0) \, \square \, (\mathrm{up} \rightarrow COUNT_1)$$

$$COUNT_{n+1} = (\mathrm{down} \rightarrow COUNT_n) \, \square \, (\mathrm{up} \rightarrow COUNT_{n+2}).$$

Another process which satisfies these laws is ZERO, where

$$ZERO = (\mathrm{iszero} \rightarrow ZERO) \, \square \, (\mathrm{up} \rightarrow (POS;ZERO))$$

$$\text{and } POS = (\mathrm{down} \rightarrow SKIP) \, \square \, (\mathrm{up} \rightarrow (POS;POS)).$$

Note that POS terminates successfully when it first performs one more "down" than "up". In order to compensate for an initial "up", it needs to perform *two* more "down"s than "up"s. This is achieved by first performing one more, and then one more again. A third definition of the same process is C_0, where

$$C_0 = (\mathrm{iszero} \rightarrow C_0) \, \square \, (\mathrm{up} \rightarrow C_1)$$

$$C_{n+1} = POS;C_n.$$

4.2 Channel naming.

In this and later sections, we shall assume that the only events are communications between processes. Thus each event consists of two parts "m.t", where "m" is the name of a *channel* along which the communication takes place, and "t" is the content of the message which passes. We define:

$$\mathrm{chan}(m.t) = m \, , \quad \mathrm{cont}_m(m.t) = t.$$

If P is a process which engages in events without a channel named, then (m.P) is the process which engages in m.t whenever P would have engaged in t:

$$(m.P) = \mathrm{cont}_m^{-1}(P).$$

For example,

$$m.COUNT_3 = (m.\mathrm{down} \rightarrow (m.COUNT_2)) \, \square \, (m.\mathrm{up} \rightarrow (m.COUNT_4)).$$

We can now construct two separate COUNTs, communicating along differently named channels:

$$(n.COUNT_0) \, ||| \, (m.COUNT_3).$$

Suppose now that a process MASTER requires to use a count register, communicating with it along channel named "m". To use the register, it engages in the events "m.up", "m.down", and "m.iszero". By using ||, we can ensure that the process (m.COUNT) engages in these events at the same time as the MASTER. But first we need to ensure that (m.COUNT) will *ignore* all communications of the MASTER, except those which are directed along channel "m". This is done by using the ||| parallel operator.

$$\text{Let } M = \{m.\mathrm{up}, \, m.\mathrm{down}, \, m.\mathrm{iszero}\}.$$

$$P \ \mathbf{ignoring} \ X = (P \, ||| \, RUN_X).$$

Then we define:

$$[m{:}COUNT_3 || MASTER] = (((m.COUNT_3) \ \mathbf{ignoring} \ (A - M)) || MASTER) \backslash M.$$

If the MASTER requires to use two differently named counts, we can similarly define:

$$[n:COUNT_0 \| [m:COUNT_3 \| MASTER]].$$

For example, the MASTER may contain the following process code, which terminates successfully when it has added the current value of m to the current value of n, leaving the former unchanged:

$$\begin{aligned}ADD = \mu p((&m.iszero \rightarrow SKIP) \,\square \\ &(m.down \rightarrow \\ &\qquad (n.up \rightarrow (p;(m.up \rightarrow SKIP)))))\end{aligned}$$

ADD has the property that:

$$\begin{aligned}&[n:COUNT_i \| [m:COUNT_j \| ADD;\ RESTOFMASTER]] = \\ &\qquad [n:COUNT_{i+j} \| [m:COUNT_j \| RESTOFMASTER]].\end{aligned}$$

This example shows how simultaneous participation in events by parallel processes can achieve the effect of communication between them.

It is possible (with care) to use the master/slave relation recursively, as shown by yet another definition of the COUNT register.

$$\begin{aligned}COUNT = \mu p.((&iszero \rightarrow p) \,\square \\ &(up \rightarrow ([m:p \| LOOP];p)))\end{aligned}$$

$$\begin{aligned}\text{where } LOOP = \mu q.((&up \rightarrow (m.up \rightarrow q)) \,\square \\ &(down \rightarrow ((m.iszero \rightarrow SKIP) \,\square \\ &\qquad (m.down \rightarrow q)))).\end{aligned}$$

The LOOP passes on to its subordinate process (m.p) all incoming "up"s and "down"s, until a "down" happens when the subordinate process is zero. The LOOP then terminates successfully. Thus [m:p || LOOP] behaves like POS, provided that p behaves like ZERO.

4.3 Buffers and chains.

We define a BUFFER (of type T) as a process which inputs any sequence of values (of type T) from a channel named "in", and outputs the same sequence of values along a channel named "out". Let m be a channel name, and

$$m.T = \{m.t \mid t \in T\}$$

$$(s \upharpoonright m) = cont_m(s \backslash (A - m.T)).$$

Less formally, (s↾m) is the sequence of values whose communication along the channel "m" is recorded in s. Now a BUFFER is a process which for all Q satisfies the laws:

$$\begin{aligned}&BUFFER \xrightarrow{s} Q \Rightarrow \\ &\quad s \in (in.T \cup out.T)^* \ \& \\ &\quad (s \upharpoonright out) \text{ is an initial subsequence of } (s \upharpoonright in) \ \& \\ &\quad (s \upharpoonright out = s \upharpoonright in \Rightarrow initials\ (Q) = in.T) \ \& \\ &\quad (s \upharpoonright out \neq s \upharpoonright in \Rightarrow initials\ (Q) \cap (out.T) \neq \{\}).\end{aligned}$$

The third line states that an empty buffer must input any value of type T, and the fourth line states that a nonempty buffer must always be prepared to output some value of type T. It is left undetermined whether a nonempty buffer may refuse to input.

A simple example that meets this specification is the single-portion buffer B1:

$$B1 = *(x:(in.T) \rightarrow (out.(cont_{in}(x)) \rightarrow SKIP))$$

In future we shall use abbreviations:

$$(?x{:}T \rightarrow F(x)) \text{ for } (y{:}(in.T) \rightarrow F(cont_{in}(y)))$$

$$\text{and } !x \text{ for } (out.x \rightarrow SKIP).$$

Thus the example B1 could be rewritten:

$$B1 = {*}(?x{:}T \rightarrow !x).$$

An unbounded buffer can be defined by an infinite set of mutually recursive equations, indexed on the current content of the buffer, which starts empty:

$$BUFF_{<>} = (?x{:}T \rightarrow BUFF_{<x>})$$

$$BUFF_{<x>s} = (?y{:}T \rightarrow BUFF_{<x>s<y>}) \;\square\; (!x;\ BUFF_{s}).$$

The process (P>>Q) is one in which everthing output by P on channel "out" is simultaneously input by process Q on channel "in"; and all such communications are concealed from their common environment. Thus all external communication on channel "in" is received by P and ignored by Q; and all external communication on channel "out" is sent by Q and ignored by P. Communication between P and Q is established by transforming each event "out.x" of P and each event "in.x" of Q into the *same* event "x". This is achieved by the inverse of the function

$$\begin{aligned} insert_m(x) &= m.x && \text{if } x \in T \\ &= x && \text{otherwise} \end{aligned}$$

$$\begin{aligned} (P{>>}Q) = {} & ((insert_{out}{}^{-1}(P)) \ \mathbf{ignoring}\ out.T) \\ & \parallel ((insert_{in}{}^{-1}(P)) \ \mathbf{ignoring}\ in.T\ ))\backslash T \end{aligned}$$

(here we have assumed that T is finite).

A buffer which stores two portions before refusing further input can be defined:

$$B2 = B1{>>}B1.$$

In general, a buffer with n portions is defined by induction:

$$\begin{aligned} B_1 &= B1 \\ B_{n+1} &= B1{>>}B_n. \end{aligned}$$

An unbounded buffer can be defined:

$$B_\infty = \mu p.\ (?x{:}T \rightarrow (p >> (!x\ ;\ B1))).$$

A buffer which may have any bound or none is

$$B_? = \mu p.\ (B1 \sqcap (?x{:}T \rightarrow (p >> (!x\ ;\ B1)))).$$

Note that it is *not* possible in our model to define a buffer with a nondeterministically chosen finite bound, without also allowing an unbounded buffer as an implementation. This is because there is no *finite* test which could detect that the buffer is unbounded.

Let $f : T^* \rightarrow T^*$ be an arbitrary monotonic function on strings, i.e. f(s) is always an initial subsequence of f(st). A process P is said to be a *pipe* for f if it satisfies the following laws:

$$\begin{aligned} & P \xrightarrow{s} Q \Rightarrow \\ & \quad s \in (in.T \cup out.T)^* \ \& \\ & \quad (s \upharpoonright out) \text{ is an initial subsequence of } f(s \upharpoonright in)\ \& \\ & \quad ((s \upharpoonright out) = f(s \upharpoonright in) \Rightarrow initials(Q) = in.T)\ \& \\ & \quad ((s \upharpoonright out) \neq f(s \upharpoonright in) \Rightarrow initials(Q) \cap out.T \neq \{\}). \end{aligned}$$

Thus a buffer is just a pipe for the identity function. If P is a pipe for f and Q is a pipe for g, then (P>>Q) is a pipe for (g o f). A simple example is a pipe for the sine function:

$$\mathtt{SIN} = *(?x{:}\mathtt{REAL} \rightarrow\ !\ \mathtt{sine}\ (x))$$

and so are (SIN >> B_3), (B_8 >> SIN), etc.

Suppose now a MASTER process requires to use the SIN process to compute sines, using a channel name "sin". It sends the argument x by sin!x (an abbreviation for (sin.in.x → SKIP)), and it inputs the result by (sin?y:REAL → F(y)), which is also an abbreviation for something similar. (Note the coding trick that assimilates output by the master with input by the slave.) The required effect can be achieved by the combination

$$[\mathtt{sin}{:}\mathtt{SIN} \,||\, \mathtt{MASTER}].$$

A pipe for the tangent function can be defined:

$$[\mathtt{sin}{:}\mathtt{SIN} \,||\, [\mathtt{cos}{:}\mathtt{COS} \,||\, \mathtt{TAN}]]$$

$$\begin{aligned} \text{where } \mathtt{TAN} = *((&?x{:}\mathtt{REAL} \rightarrow \mathtt{sin}!\ x;\mathtt{cos}!x); \\ &(\mathtt{sin}?y{:}\mathtt{REAL} \rightarrow (\mathtt{cos}?z{:}\mathtt{REAL} \rightarrow\ !(y/z)))) \end{aligned}$$

A process P is said to be a *variable* (of type T) if it is always prepared to output the value it has most recently input; i.e. for all Q:

$$\begin{aligned} P \xrightarrow{s} Q \Rightarrow\ &(s \upharpoonright \mathtt{in} = \langle\rangle \Rightarrow \mathtt{initials}(Q) = \mathtt{in}.T) \\ &\& \ (s \upharpoonright \mathtt{in} \neq \langle\rangle \Rightarrow \\ &\qquad \mathtt{initials}(Q) = \\ &\qquad\qquad (\mathtt{in}.T \cup \{\mathtt{out.\ last}\ (s \upharpoonright \mathtt{in})\})). \end{aligned}$$

A process definition satisfying these laws is:

$$\mathtt{VAR}_T = (?x{:}T \rightarrow V_x)$$

$$\text{where } V_x = (?y{:}T \rightarrow V_y)\ \Box\ (!x;V_x) \quad \text{for all } x \text{ in } T.$$

V_x is the behaviour of a variable with value x. A fresh local instance of such a variable can be declared thus:

$$[\mathtt{m}{:}\mathtt{VAR}_T \,||\, \mathtt{MASTER}].$$

A *stack* (for type T) is a process P which outputs everything that it has input, on a last-in/first-out principle; and outputs the signal "isempty" when empty. For all Q it obeys the laws:

$$\begin{aligned} P \xrightarrow{s} Q \Rightarrow\ & \\ &(\mathtt{length}(s \upharpoonright \mathtt{in}) = \mathtt{length}(s \upharpoonright \mathtt{out}) \Rightarrow \\ &\qquad \mathtt{initials}(Q) = (\mathtt{in}.T \cup \{\mathtt{out.isempty}\})) \\ &\& \ (\mathtt{length}(s \upharpoonright \mathtt{in}) > \mathtt{length}(s \upharpoonright \mathtt{out}) \Rightarrow \\ &\qquad \mathtt{in}.T \subseteq \mathtt{initials}\ (Q)\ \& \\ &\qquad \mathtt{initials}(Q) \cap \mathtt{out}.T \neq \{\}) \end{aligned}$$

$$\begin{aligned} P \xrightarrow{st\langle out.x\rangle} Q\ \&\ \mathtt{length}(t \upharpoonright \mathtt{in}) = \mathtt{length}(t \upharpoonright \mathtt{out}) \Rightarrow \\ x = \mathtt{last}(s \upharpoonright \mathtt{in}). \end{aligned}$$

Three different implementations of a stack can be modelled on three different implementations of the COUNT. We hope the reader will enjoy constructing them.

5 Prospects.

The original objective of denotational semantics was to provide a clear, consistent, and unambiguous definition of a programming language which is likely to have more than one implementation. Such a definition could serve usefully as a national or international standard; it would give a precise specification which must be met by each implementor; and it would tell each programmer exactly what he can rely on in all implementations. Thus it would achieve the primary objective of standardisation, namely the reliable conjunction of programs and implementations from widely differing sources. The deficiencies of existing language standards can be directly attributed to their failure to take advantage of this known technology - a failure which to future generations will seem amazing. In the area of parallel programming languages, we hope that the development of a suitable semantic model at an early stage will forestall a repetition of the problems that have beset the development and standardisation of sequential programming languages.

Apart from the improved quality of programming language standards, the techniques of mathematical semantics have much to offer in improving the reliability of computer programs. In the first place, they offer the possibility that an implementor can prove with mathematical rigour that his implementation meets the standard specification of the language. Clearly, no program can be more reliable than the implementation of the language in which it is expressed for input to a computer.

A second advantage of a mathematical description of a programming language is that it offers the individual programmer the opportunity to prove the correctness of his program with respect to some description of its intended behaviour. For this, he would need to identify the mathematical object denoted by his program, and then prove that this object exhibits the required mathematical properties. Unfortunately, this method of program proving is impractically laborious; it is like trying to

solve differential equations using only the original definitions of derivatives in the epsilon-delta terminology of analysis. What is required for practical program development and proof is a formal calculus, similar to the assertional calculus for sequential programs, which will permit a reasonably direct expression of the purpose of each command, and a method of proving that it meets its purpose. Such a calculus must be firmly based on a proof of its conformity with the mathematical model, just as the differential calculus is ultimately based on the Dedekind model of real numbers. But these are topics for future research.

References

[1] C. A. R. Hoare Communicating Sequential Processes
Commun. A. C. M **21** 8, Aug 1978

[2] Robin Milner Calculus of Communicating Systems
Springer Lecture Notes in Computer Science **92**
Springer Verlag 1980.

[3] E. W. Dijkstra Cooperating Sequential Processes
in Programming Languages, *ed.* F. Genuys
Academic Press.

[4] D. S. Scott Data Types as Lattices
SIAM Journal on Computing **5** 1976, pp. 522-587