NOTES ON COMMUNICATING SEQUENTIAL PROCESSES

C.A.R. HOARE

Technical Monograph PRG-33

August 1983

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford. OX1 3QD.

ABSTRACT

These notes present a coherent and comprehensive introduction to the theory and applications of Communicating Sequential Processes. Most of the illustrative examples have appeared earlier in PRG-22. The theory described in PRG-16 has been taken as the basis of a number of algebraic laws, which can be used for proofs of equivalence and can justify correctness-preserving transformations. A complete method for specifying processes and proving their correctness has been taken over from PRG-20 and PRG-23. Many of the concepts have been implemented in LISPKIT, as described in PRG-32.

CONTENTS

CHAPTER ONE

PROCESSES

1.1 Introduction

Forget for a while about computers and computer programming, and think instead about objects in the world around us, which act and interact with us and with each other in accordance with some characteristic pattern of behaviour. Think of clocks and counters and telephones and board games and vending machines. To describe their patterns of behaviour, first decide what kinds of event or action will be of interest; and choose a different name for each kind. In the case of a simple vending machine, the actions may be

coin:	the insertion of a coin in the slot of a vending machine.
choc:	the extraction of a chocolate from the dispenser of the machine.

In the case of a more complex vending machine, there may be a greater variety of events:

in1p:	the insertion of one penny
in2p:	the insertion of a two penny coin
small:	the extraction of a small biscuit or cookie
large:	the extraction of a large biscuit or cookie
out1p:	the extraction of one penny in change

The set of names of events which are considered relevant for a particular description of an object are called its <u>alphabet</u>.

The choice of an alphabet usually involves a deliberate simplification, a decision to ignore many other properties and actions which are considered to be of lesser interest. For example, the colour, weight, and shape of a vending machine are not described, and certain very necessary events in its life, such as replenishing the stack of chocolates or emptying the coin box, are deliberately ignored — perhaps on the grounds that they are not, or should not be, of any concern to the customers of the machine.

The actual occurrence of each event in the life of an object should be regarded as instantaneous - an atomic action without duration. Extended or time-consuming actions should be represented by a pair of events, the first denoting its start and the second denoting its finish. The duration of the action is represented by the interval between these two events, which may be saparated by occurrence of other events.

In choosing an alphabet, there is no need to make a distinction between events which are initiated by the object (perhaps "choc") and those which are initiated by some agent outside the object (for example, "coin"). The avoidance of the concept of causality leads to considerable simplification in the theory and its application.

Let us now begin to use the word <u>process</u> to stand for the behaviour pattern of an object, insofar as it can be described in terms of the limited set of events selected as its alphabet. We shall use the following conventions.

1. Words in lower case letters denote events, e.g.,

   coin, choc, in2p, out1p,

   and also the letters c, d, e.

2. Words in upper case letters denote specific defined processes:

   e.g. VMS the simple vending machine

   VMC the complex vending machine

3. The letters x, y, z are variables denoting events.

4. The letters A, B, C stand for sets of events.

5. The letters P, Q, R stand for arbitrary processes.

6. The letters X, Y are variables denoting processes.

7. The alphabet of process P is denoted αP, e.g.,

$$\alpha VMS = \{coin, choc\}$$

$$\alpha VMC = \{in1p, in2p, small, large, out1p\}$$

Example

X1 The process with alphabet A which never actually engages in any of the events of A is called $STOP_A$. This describes the behaviour of a broken object: although it is equipped with the physical capabilities to engage in the events of A, it never exercises those capabilities. Nevertheless, it is useful to distinguish objects with different alphabets, even if they never do anything. So $STOP_{\alpha VMS}$ might have given out a chocolate, whereas $STOP_{\alpha VMC}$ could only have given out biscuits. It is logically impossible for a process to engage in events outside its alphabet; but there may be events within its alphabet which can never actually happen.

In the remainder of this introduction, we shall define some simple notations to aid in the description of objects which actually succeed in doing something.

1.1.1 Prefix

The following notation should be read "x then P":

$$(x \longrightarrow P)$$

It describes an object which first engages in the event x and then behaves exactly as described by P. $(x \longrightarrow P)$ is defined to have the same alphabet as P, so this notation must not be used unless x is in that alphabet:

$$x \in \alpha(x \longrightarrow P) = \alpha P$$

Examples

X1 A simple vending machine which consumes one coin before breaking:

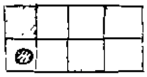
$$(coin \longrightarrow STOP_{\alpha VMS})$$

X2 A simple vending machine that successfully serves two customers before breaking

$$(coin \longrightarrow (choc \longrightarrow (coin \longrightarrow (choc \longrightarrow STOP_{\alpha VMS}))))$$

In future, we shall omit brackets in the case of linear sequences of events, like those in X2.

X3 A counter starts on the bottom left square of a board, and can move only up or right to an adjacent white square



$$\alpha CTR = \{up, right\}$$

$$CTR = (right \longrightarrow up \longrightarrow right \longrightarrow right \longrightarrow STOP_{\alpha CTR})$$

1.1.2 Recursion

The prefix notation can be used to describe the entire behaviour of a process that eventually stops. But it would be extremely tedious to write out the full behaviour of a vending machine for its maximum design life; so we need a method of describing repetitive behaviour patterns by much shorter notations. Preferably these notations should not require a prior decision on the length of the life of an object; this will permit description of objects which will continue to act and interact with their environment for as long as they are needed.

Consider the simplest possible everlasting object, a clock which never does anything but tick (the act of winding it is deliberately ignored)

$$\alpha\text{CLOCK} = \{tick\}$$

Consider next an object that behaves exactly like the clock, except that it first emits a single "tick"

$$(tick \longrightarrow \text{CLOCK})$$

The behaviour of this object is indistinguishable from that of the original clock. The same process therefore describes the behaviour of both objects. This reasoning leads to formulation of the equation:

$$\text{CLOCK} = (tick \longrightarrow \text{CLOCK})$$

This can be regarded as an implicit definition of the behaviour of the clock, in the same way that the square root of two might be defined as the solution for x in the equation

$$x = x^2 + x - 2$$

The equation for the clock has some obvious consequences, which are derived by simply substituting equals for equals:

$$\begin{aligned} \text{CLOCK} &= (tick \longrightarrow \text{CLOCK}) && \text{original equation} \\ &= (tick \longrightarrow (tick \longrightarrow \text{CLOCK})) && \text{by substitution} \end{aligned}$$

$$\therefore \quad \text{CLOCK} = (tick \longrightarrow tick \longrightarrow tick \longrightarrow \text{CLOCK}) \quad \text{similarly}$$

The equation can be "unfolded" as many times as required, and the possibility of further unfolding will still be preserved. The potentially unbounded behaviour of the CLOCK has been effectively defined as:

$$tick \longrightarrow tick \longrightarrow tick \longrightarrow \ldots$$

in the same way as the square root of two can be thought of as the limit of a series of decimals:

$$1.414 \ldots$$

This method of self-referential or recursive definition of processes will work properly only if the right hand side of the equation starts with at least one event prefixed to all recursive occurrences of the process name. For example, the recursive "definition":

$$X = X$$

does not succeed in defining anything, since everything is a solution to this equation. A process description which begins with a prefix is said to be <u>guarded</u>. If $F(X)$ is a guarded expression containing the process name X, and A is the intended alphabet of X, then the equation

$$X = F(X)$$

has an unique solution with alphabet A. It is sometimes convenient to denote this solution by the expression

$$\mu X{:}A.F(X)$$

Here X is a local name (bound variable), and can be changed at will, since

$$\mu X{:}A.F(X) = \mu Y{:}A.F(Y)$$

This equality is justified by the fact that a solution for X of the equation

$$X = F(X)$$

is also a solution for Y of the equation

$$Y = F(Y)$$

Examples

X1 A perpetual clock:

$$CLOCK = \mu X{:}\{tick\}.(tick \longrightarrow X)$$

X2 At last, a simple vending machine which serves as many chocs as required

$$VMS = (coin \longrightarrow (choc \longrightarrow VMS))$$

X3 A machine that gives change for 5p

$$\alpha CH5A = \{in5p,\ out2p,\ out1p\}$$

$$CH5A = (in5p \longrightarrow out2p \longrightarrow out1p \longrightarrow out2p \longrightarrow CH5A)$$

X4 A different change giving machine with the same alphabet

$$CH5B = (in5p \longrightarrow out1p \longrightarrow out1p \longrightarrow out1p \longrightarrow out2p \longrightarrow CH5B)$$

In future, we shall often omit an explicit definition of the alphabet, when this is obvious from the context or content of the process.

1.1.3 Choice

With prefixing and recursion it is possible to describe objects with a single possible stream of behaviour. However, many objects allow their behaviour to be influenced by interaction with the environment within which they are placed. For example, a vending machine may offer a choice of slots for inserting a 2p coin or a 1p coin; and it is the customer that decides between these two events. If x and y are distinct events

$$(x \longrightarrow P \mid y \longrightarrow Q)$$

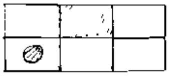
describes an object which initially engages in either of the events x or y. After the first event has occurred, the subsequent behaviour of the object is described by P if the first event was x, or by Q if the first event was y. Since x and y must be different events, the choice between P and Q is determined by the first event that actually occurs. As before we insist on identity of alphabets, i.e.,

$$\{x,y\} \subseteq \alpha(x \longrightarrow P \mid y \longrightarrow Q) = \alpha P = \alpha Q$$

The bar $\mid$ should be pronounced "choice"

Examples

X1 The possible movements of a counter on the board



are defined by the process:

$$(up \longrightarrow STOP \mid right \longrightarrow right \longrightarrow up \longrightarrow STOP)$$

X2 A machine which offers a choice of two combinations of change for 5p

$$\begin{aligned} CH5C = in5p \longrightarrow (&out1p \longrightarrow out1p \longrightarrow out1p \longrightarrow out2p \longrightarrow CH5C \\ \mid &out2p \longrightarrow out2p \longrightarrow out1p \longrightarrow CH5C) \end{aligned}$$

X3 A machine that serves either chocolate or toffee

$$VMCT = \mu X.\ coin \longrightarrow (choc \longrightarrow X \mid toffee \longrightarrow X)$$

X4 A more complicated vending machine, that offers a choice of coins and a choice of goods and change

$$
\begin{array}{l}
VMC = (in2p \longrightarrow (large \longrightarrow VMC \\
\qquad\qquad\qquad\qquad\quad \mid small \longrightarrow out1p \longrightarrow VMC) \\
\qquad\qquad \mid in1p \longrightarrow (small \longrightarrow VMC \\
\qquad\qquad\qquad\qquad\quad \mid in1p \longrightarrow (large \longrightarrow VMC \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid in1p \longrightarrow STOP)))
\end{array}
$$

Like many complicated machines, this has a design flaw. A temporary "fix" for this bug is to write a notice on the machine

"WARNING: do not insert three coins in a row".

The definition of choice can readily be extended to more than two alternatives, e.g.,

$$(x \longrightarrow P \mid y \longrightarrow Q \mid \ldots \mid z \longrightarrow R)$$

In general, if A is any set of events, and $P(x)$ is an expression defining a process for each different x in A, then

$$(x{:}A \longrightarrow P(x))$$

defines a process which first offers a choice of any event y in A, and then behaves like $P(y)$. It should be pronounced "x from A then P of x". In this construction, x is a local variable, so

$$(x{:}A \longrightarrow P(x)) \;=\; (y{:}A \longrightarrow P(y))$$

The set A defines the "initial menu" of the process, since it gives the set of actions between which a choice is to be made at the start.

Example

X5 A process which at all times can engage in any event of its alphabet A

$$\alpha RUN_A = A$$

$$RUN_A = (x{:}A \longrightarrow RUN_A)$$

In the special case that A contains only one event e,

$$(x{:}\{e\} \longrightarrow P(x)) = (e \longrightarrow P(e))$$

since e is the only possible initial event. In the even more special case that A is empty, nothing at all can happen, so

$$(x{:}\{\} \longrightarrow P(x)) \;=\; (y{:}\{\} \longrightarrow Q(y)) \;=\; STOP$$

Thus prefixing and STOP are just special cases of the general choice notation. This will be a great advantage in the formulation of general laws governing processes.

1.1.4 Mutual Recursion

Recursion permits the definition of a single process as the solution of a single equation. The technique is easily generalised to solution of sets of equations in more than one unknown. For this to work properly, all the right hand sides must be guarded, and each unknown process must appear exactly once on the left hand side of one of the equations.

Example

X1 A drinks dispenser has a knob with two settings, labelled GIN and WHISKY. The actions of setting the knob are "setgin" and "setwhisky". The actions of dispensing a drink are "gin" ~~and "whisky".~~ The knob is initially in a neutral position, to which it never returns. Here are the three equations.

$$\alpha DD = \alpha G = \alpha W = \{setgin, setwhisky, coin, gin, whisky\}$$

$$DD = (setgin \longrightarrow G \mid setwhisky \longrightarrow W)$$

$$G = (coin \longrightarrow gin \longrightarrow G \mid setwhisky \longrightarrow W)$$

$$W = (coin \longrightarrow whisky \longrightarrow W \mid setgin \longrightarrow G)$$

Informally, the drinks dispenser may be described as being in a particular one of the two states G and W. In each state it may either serve the appropriate drink or be switched to the other state.

By using indexed variables, it is possible to specify infinite sets of equations.

Example

X2 An object starts on the ground, and may move "up". At any time thereafter it may move "up" or "down", except that when on the ground it cannot move any further down. But when it is on the ground, it may move "around". Let n range over the natural numbers $\{0,1,2, \ldots\}$. For each n, introduce the indexed name CT_n to describe the behaviour of the object when it is n moves off the ground. Its initial behaviour is defined as

$$CT_0 = (up \longrightarrow CT_1 \mid around \longrightarrow CT_0)$$

and the remaining infinite set of equations are

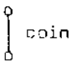
$$CT_{n+1} = (up \longrightarrow CT_{n+2} \mid down \longrightarrow CT_n)$$

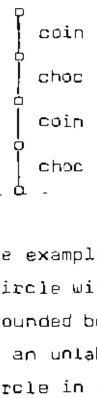
where n ranges over the natural numbers 0,1,2, ...

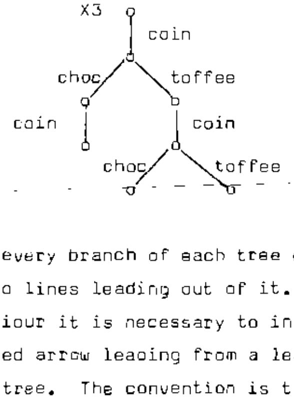
1.2 Pictures

It may be helpful sometimes to make a pictorial representation of a process as a tree structure, consisting of circles connected by lines. The circles represent states of the process, and the lines represent transitions between the states. The single circle at the root of the tree (usually drawn at the top of the page) is the starting state; and the process moves downward along the lines. Each line is labelled by the event which occurs on making that transition.

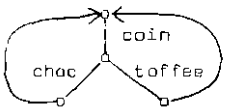
Examples

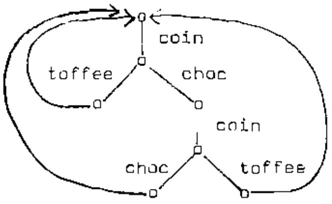
X1


X2


X3


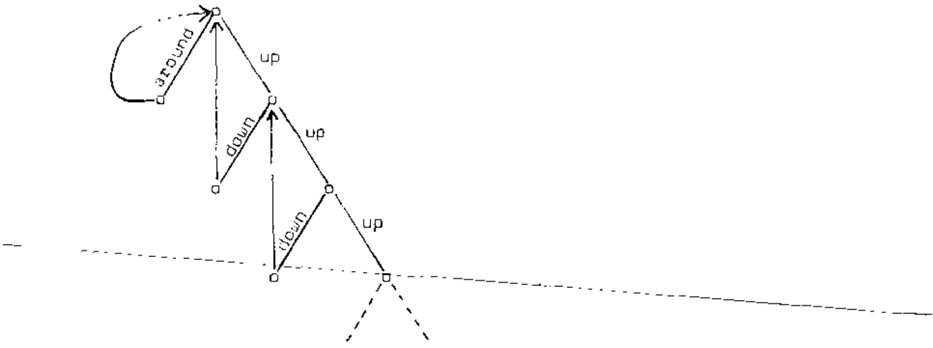
In these three examples, every branch of each tree ends in STOP, represented as a circle with no lines leading out of it. To represent processes with unbounded behaviour it is necessary to introduce another convention, namely an unlabelled arrow leading from a leaf circle back to some earlier circle in the tree. The convention is that when a process reaches the node at the tail of the arrow, it immediately and imperceptibly goes back to the node to which the arrow points.

X4


X5


Clearly, these two different pictures illustrate exactly the same process. It is one of the weaknesses of pictures that proofs of such equality are difficult to conduct pictorially.

Another problem with pictures is that they cannot illustrate processes with a very large or infinite number of states, for example CT_0



A count with only 65 536 different states would take a long time to draw.

1.3 Laws

Even with the very restricted set of notations introduced so far, there are many different ways of describing the same behaviour. For example, it obviously should not matter in which order a choice between events is presented:

$$(x \longrightarrow P \mid y \longrightarrow Q) \;=\; (y \longrightarrow Q \mid x \longrightarrow P).$$

On the other hand, a process that can do something is not the same as one that can't do anything

$$(x \longrightarrow P) \;\neq\; STOP$$

In order to understand a notation properly and to use it effectively, we must learn to recognise which expressions describe the same object and which do not, just as everyone who understands arithmetic knows that $(x + y)$ is the same number as $(y + x)$. Identity of processes with the same alphabet may be proved or disproved by appeal to algebraic laws very like those of arithmetic.

The first law deals with the choice operator. It states that two processes defined by choice are different if they offer different choices on the first step, or if after the same first step they behave differently. However, if the initial choices are the same, and for each initial choice the subsequent behaviours are the same, then obviously the processes are identical:

L1 $\quad (x:A \longrightarrow P(x)) \;=\; (y:B \longrightarrow Q(y))$

$\quad \equiv \; A = B \;\;\&\;\; \forall x \in A.\;\; P(x) = Q(x)$

Here and elsewhere, we assume without stating it that the alphabets of the processes on each side of an equation are the same.

The law L1 has a number of immediate notational consequences:

L1A $\quad STOP \neq (d \longrightarrow P)$

Proof. LHS $= x: \{\} \longrightarrow P$ $\qquad$ by definition

$\neq x: \{d\} \longrightarrow P$ $\qquad$ because $\{\} \neq \{d\}$

$=$ RHS $\qquad$ by definition

L1B $\quad (c \longrightarrow P) \neq (d \longrightarrow Q)$ $\qquad$ if $c \neq d$

Proof. $\{c\} \neq \{d\}$

L1C $\quad (c \longrightarrow P \mid d \longrightarrow Q) \;=\; (d \longrightarrow Q \mid c \longrightarrow P)$

Proof $\quad$ define $R(x) = P \quad$ if $x = c$

$\qquad = Q \quad$ if $x = d$

LHS $= (x: \{c,d\} \longrightarrow R(x))$

$= (x: \{d,c\} \longrightarrow R(x))$

$=$ RHS.

L1D $\quad (c \longrightarrow P) = (c \longrightarrow Q) \equiv P = Q.$

These laws permit proof of simple theorems:

Examples

X1 $(coin \longrightarrow choc \longrightarrow coin \longrightarrow choc \longrightarrow STOP) \neq (coin \longrightarrow STOP)$

Proof by L1C then L1A.

X2 $\mu X.(coin \longrightarrow (choc \longrightarrow X \mid toffee \longrightarrow X))$

$= \mu X.(coin \longrightarrow (toffee \longrightarrow X \mid choc \longrightarrow X))$

To prove more general theorems about recursively defined processes, it is necessary to introduce a law which states that every properly guarded recursive equation has only one solution.

L2 If $F(X)$ is a guarded expression,

$(Y = F(Y)) \equiv (Y = \mu X.F(X))$

Corollary: $\mu X.F(X) = F(\mu X.F(X))$

Examples

X1 Let $VM1 = \mu X.(coin \longrightarrow choc \longrightarrow X)$

$VM2 = \mu X.(choc \longrightarrow coin \longrightarrow X)$

Required to prove:

$VM2 = (choc \longrightarrow VM1)$.

Proof. $VM1 = (coin \longrightarrow (choc \longrightarrow VM1))$ by def VM1

By prefixing "choc $\longrightarrow$ " to both sides:

$(choc \longrightarrow VM1) = (choc \longrightarrow (coin \longrightarrow (choc \longrightarrow VM1)))$

i.e., $(choc \longrightarrow VM1)$ is a solution for X in the equation

$X = (choc \longrightarrow coin \longrightarrow X)$

But VM2 is defined as the only solution to this equation.

$\therefore$ $(choc \longrightarrow VM1) = VM2$.

This theorem is so obviously true that its proof in no way adds to its credibility. The only purpose of the proof is to show by example that the laws are powerful enough to establish facts of this kind. When proving obvious facts from less obvious laws, it is important to justify every line of the proof in full, as a check that the proof is not circular.

The law for recursion can be extended to mutual recursion.

If $F(i,X)$ is a guarded expression for all i in B

$(\forall i \in B . X_i = F(i,X) \ \& \ Y_i = F(i,Y)) \Longrightarrow X = Y.$

1.4 Implementation of Processes

Every process P expressible in the notations introduced so far can be written in the form

$$(x{:}A \longrightarrow P(x))$$

where P is a function from symbols to processes, and where A may be empty (in the case of STOP), or may contain only one member (in the case of prefix) or may contain more than one member (in the case of choice). In the case of a recursively defined process, we have insisted that the recursion should be guarded, so that it may be written

$$\mu X.(x{:}A \longrightarrow P(x,X));$$

and this may be *unfolded* to the required form:

$$(x{:}A \longrightarrow P(x,\mu X.(x{:}A \longrightarrow P(x,X)))).$$

Thus every process may be regarded as a function P with a domain A, defining the set of events in which the process is initially prepared to engage; and for each x in A, $P(x)$ defines the future behaviour of the process if the first event was x.

This insight permits every process to be implemented as a function in some suitable functional programming language such as LISP. Each event in the alphabet of a process is implemented as an atom, for example "COIN, "TOFFEE. A process is a function which can be applied to such a symbol as argument. If the symbol is not a possible first event for the process, the function gives as its result a special symbol "BLEEP, which is used only for this purpose. For example, since STOP never engages in any event, this is the only result it can ever give:

$$\text{STOP} = \lambda x.\ \text{"BLEEP}.$$

But if the actual argument is a possible event for the process, the function gives back as its result another function, representing the subsequent behaviour of the process. Thus $(coin \longrightarrow \text{STOP})$ is implemented as the function:

```
λx. if x = "COIN then STOP
                      else "BLEEP
```

This last example takes advantage of the facility of LISP for returning a function (e.g., STOP) as the result of a function. LISP also allows a function to be passed as an argument to a function, a facility used in implementing a general prefixing function ($c \longrightarrow P$):

```
prefix (c,P)  =  λx. if x = c then P
                                  else "BLEEP
```

A function to implement a general binary choice ($c \longrightarrow P \mid d \longrightarrow Q$) would be:

```
choice2(c,P,d,Q) =  λx.if x = c then P
                       else if x = d then Q
                                     else "BLEEP
```

Recursively defined processes may be implemented with the aid of the LABEL feature of LISP. For example, the simple vending machine ($\mu X.\mathrm{coin} \longrightarrow \mathrm{choc} \longrightarrow X$):

```
LABEL X.prefix("COIN, prefix ("CHOC,X))
```

The LABEL may also be used to implement mutual recursion. For example (1.1.4.X2), CT may be regarded as a function from natural numbers to processes (which are themselves functions - but let not that be a worry). So CT may be defined:

```
CT = LABEL X. λn.
     if n = 0 then choice2("AROUND,X(0),"UP,X(1))
               else choice2("UP,X(n+1),"DOWN,X(n-1))
```

The process that starts on the ground is CT(0).

If P is a function representing a process, and A is a list containing the symbols of its alphabet, the LISP function

```
menu (A,P)
```

gives a list of all those symbols of A which can occur as the first event in the life of P:

```
menu (A,P)  =  if A = NIL then NIL
               else if P(car(A)) = "BLEEP then menu(cdr(A),P)
                                          else cons(car(A), menu(cdr(A),P))
```

If x is in menu(A,P), P(x) is not "BLEEP, and is therefore a function defining the future behaviour of P after engaging in x. Thus if y is in menu(A,P(x)) then P(x)(y) will give its later behaviour, after both x and y have occurred.

This suggests a useful method of exploring the behaviour of a process. Write a program which first outputs the value of menu(A,P) on a screen, and then inputs a symbol from the keyboard. If the symbol is not in the menu, it should be greeted with an audible bleep and then ignored. Otherwise the symbol is accepted, and the process is repeated with P replaced by the result of applying P to the accepted symbol. The process is terminated by typing an "END symbol. Thus if k is the sequence of symbols input from the keyboard, the following function gives the sequence of outputs required:

```
interact(A,P,k) = cons(menu(A,P),
                       if car(k) = "END then NIL
                       else if P(car(k)) = "BLEEP then
                                 cons("BLEEP,interact(A,P,cdr(k)))
                            else interact(A,P(car(k)),cdr(k)))
```

The notations used above for defining LISP functions are very informal, and they will need to be translated to the specific conventional S-expression form of each particular implementation of LISP. For example in LISPkit, the prefix function can be defined:

```
(prefix
   lambda
     (a p)
     (lambda (x) (if (eq x a) p (quote BLEEP))))
```

Fortunately, we shall use only a very small subset of pure functional LISP, so there should be no difficulty in translating and running these processes in a variety of dialects on a variety of machines. For this reason we may freely mix higher level notations with the code of the LISP functions.

1.5 Traces

A <u>trace</u> of the behaviour of a process is a finite sequence of symbols recording the events in which a process has engaged up to some moment in time. Imagine there is an observer with a notebook who watches the process and writes down the name of each event as it occurs. We can validly ignore the possibility that two events occur simultaneously; for if they did, the observer would still have to record one of them first and then the other; and the order in which he records them will not matter.

A trace will be denoted as a sequence of symbols, separated by commas and enclosed in angular brackets:

$\langle x,y \rangle$	consists of two events, x followed by y.
$\langle x \rangle$	is a sequence containing only the event x.
$\langle \rangle$	is the empty sequence containing no events.

Examples

X1 A trace of ~~the simple~~ vending machine VMS (1.1.2.X2) at the moment it has completed service of its first two customers:

$\langle$coin, choc, coin, choc$\rangle$

X2 A trace of the same machine before the second customer has extracted his choc:

$\langle$coin, choc, coin$\rangle$

Neither the process nor its observer understands the concept of a "completed transaction". The hunger of the expectant customer, and the readiness of the machine to satisfy it are not in the alphabet of these processes, and cannot be observed or recorded.

X3 Before a process has engaged in any events, the notebook of the observer is empty. This is represented by the empty trace

$\langle \rangle$

Every process has this as its shortest possible trace.

X4 The complex vending machine VMC (1.1.3.X4) has the following seven traces of length two or less

$\langle \rangle$

$\langle$in2p$\rangle$ $\langle$in1p$\rangle$

$\langle$in2p,large$\rangle$ $\langle$in2p,small$\rangle$ $\langle$in1p,in1p$\rangle$ $\langle$in1p,small$\rangle$

Only one of the four traces of length two can actually occur for a given machine. The choice between them will be determined by the wishes of the first customer to use the machine.

X5 A trace of the same machine if its first customer has ignored the warning:

$$\langle in1p, in1p, in1p \rangle$$

The trace does not actually record the breakage of the machine. Breakage is only indicated by the fact that among all the possible traces of the machine, there is no trace which extends this one, i.e., there is no event x such that

$$\langle in1p, in1p, in1p, x \rangle$$

is a possible trace of VMC. The customer may fret and fume; the observer may watch eagerly with pencil poised; but not another single event can occur, and not another symbol will ever be written in the notebook. The ultimate disposal of customer and machine are not in our chosen alphabet; and we have decided to ignore them.

1.6 Operations on Traces

Traces play a central role in recording, describing, and understanding the behaviour of processes. In this section we explore some of the general properties of traces and of operations on them. We will use the following conventions

s, t, u	stand for traces
S, T, U	stand for sets of traces
f, g, h	stand for functions

1.6.1 Catenation

By far the most important operation on traces is catenation, which constructs a trace from a pair of operands s and t by simply putting them together in this order; the result will be denoted

$$s \,\hat{}\, t,$$

For example:

$$\langle coin, choc \rangle \,\hat{}\, \langle coin, toffee \rangle = \langle coin, choc, coin, toffee \rangle$$

$$\langle in1p \rangle \,\hat{}\, \langle in1p \rangle = \langle in1p, in1p \rangle$$

$$\langle in1p, in1p \rangle \,\hat{}\, \langle \rangle = \langle in1p, in1p \rangle$$

The most important properties of catenation are that it is associative, and has $\langle\rangle$ as its unit.

L1 $s^\frown\langle\rangle = \langle\rangle^\frown s = s$ (unit)

L2 $s^\frown(t^\frown u) = (s^\frown t)^\frown u$ (associative)

The following laws are both obvious and useful.

L3 $s^\frown t = s^\frown u \equiv t = u$

L4 $s^\frown t = u^\frown t \equiv s = u$

L5 $s^\frown t = \langle\rangle \equiv s = \langle\rangle \;\&\; t = \langle\rangle$

Let f stand for a function which maps traces onto traces. The function is said to be strict if it maps the empty trace to the ~~empty~~ trace:

$$f(\langle\rangle) = \langle\rangle \qquad \text{(strict)}$$

It is said to be distributive if it distributes through catenation:

$$f(s^\frown t) = f(s)^\frown f(t) \qquad \text{(distributive)}$$

Many useful operations on traces will have these properties.

If n is a natural number, we define t^n as n copies of t catenated with each other. It is readily defined by induction.

L6 $t^0 = \langle\rangle$

L7 $t^{n+1} = t^\frown t^n$

This definition itself gives two useful laws; here are two more:

L8 $t^{n+1} = t^{n\frown} t$

L9 $(s^\frown t)^{n+1} = s^\frown(t^\frown s)^{n\frown} t$

1.6.2 Restriction

The expression $(t \upharpoonright A)$ denotes the trace t when restricted to symbols in the set A; it is formed from t simply by omitting all symbols outside A. For example:

$$\langle around, up, down, around\rangle \upharpoonright \{up, down\} = \langle up, down\rangle$$

Restriction is strict and distributive:

L1 $\langle\rangle \upharpoonright A = \langle\ \rangle$

L2 $(s^\frown t) \upharpoonright A = (s \upharpoonright A)^\frown (t \upharpoonright A)$

Its effect on unit sequences is obvious.

L3 $\langle x\rangle \upharpoonright A = \langle x\rangle$ if $x \in A$

L4 $\langle y\rangle \upharpoonright A = \langle\ \rangle$ if $y \tilde{\in} A$

A strict and distributive function is uniquely defined by defining its effect on unit sequences, since its effect on all longer sequences can be calculated by distributing the function to each individual element of the sequence and catenating the results. For example if $y \neq x$:

$$\langle x, y, x\rangle \upharpoonright \{x\} = (\langle x\rangle^\frown \langle y\rangle^\frown \langle x\rangle) \upharpoonright \{x\}$$

$$= (\langle x\rangle \upharpoonright \{x\})^\frown (\langle y\rangle \upharpoonright \{x\})^\frown (\langle x\rangle \upharpoonright \{x\}) \quad \text{by L2}$$

$$= \langle x\rangle ^\frown \langle\ \rangle ^\frown \langle x\rangle \quad \text{by L3, L4}$$

$$= \langle x, x\rangle$$

The following laws show the relationship between restriction end set operations. A trace restricted to the empty set of symbols leaves nothing; and a successive restriction by two sets is the same as a single restriction by the intersection of the two sets.

L5 $s \upharpoonright \{\ \} = \langle\ \rangle$

L6 $(s \upharpoonright A) \upharpoonright B = s \upharpoonright (A \cap B)$

1.6.3 Head and Tail

If s is a nonempty sequence, its first symbol is denoted s_0, and the result of removing the first symbol is s'. For example

$$\langle x, y, x\rangle_0 = x$$

$$\langle x, y, x\rangle' = \langle y, x\rangle.$$

Both of these operations are undefined for the empty sequence.

L1 $(\langle x \rangle ^\frown s)_0 = x$

L2 $(\langle x \rangle ^\frown s)' = s$

L3 $s = (\langle s_0 \rangle ^\frown s')$ if $s \neq \langle \rangle$

The following law gives a convenient method of proving whether two traces are equal

L4 $s = t \equiv (s = t = \langle \rangle \vee s_0 = t_0 \ \& \ s' = t')$

1.6.4 Star

The set A^* is the set of all finite traces (including $\langle \rangle$) which are formed from symbols in the set A. When such traces are restricted to A, they remain unchanged. This fact permits a simple definition:

$$A^* = \{ s \mid s \upharpoonright A = s \}.$$

The following set of laws are sufficiently powerful to determine whether a trace is a member of A^* or not.

L1 $\langle \rangle \in A^*$

L2 $\langle x \rangle \in A^* \equiv x \in A$

L3 $(s ^\frown t) \in A^* \equiv s \in A^* \ \& \ t \in A^*$

For example, if $x \in A$, $y \notin A$

$$\langle x, y \rangle \in A^* \equiv (\langle x \rangle ^\frown \langle y \rangle) \in A^*$$

$$\equiv (\langle x \rangle \in A^*) \ \& \ (\langle y \rangle \in A^*) \qquad \text{by L4}$$

$$\equiv \text{true} \ \& \ \text{false} \qquad \text{by L2, L3}$$

1.6.5 Ordering

If s is a copy of an initial subsequence of t, it is possible to find some extension u of s such that $s ^\frown u = t$. We therefore define

$$s \leq t = (\exists u \,.\, s ^\frown u = t)$$

and say that s is a <u>prefix</u> of t.

For example,

$$\langle x, y \rangle \leq \langle x, y, x, w \rangle$$

but not $\langle x, y \rangle \leq \langle z, y, x \rangle$ if $z \neq x$.

The $\leqslant$ relation is a partial ordering, and its least element is $\langle\rangle$, as stated in laws 1 to 4

L1 $\langle\rangle \leqslant s$ least element

L2 $s \leqslant s$ reflexive

L3 $s \leqslant t \;\&\; t \leqslant s \implies s = t$ antisymmetric

L4 $s \leqslant t \;\&\; t \leqslant u \implies s \leqslant u$ transitive

The following law, together with L1 gives a method for computing whether $s \leqslant t$ or not.

L5 $(\langle x\rangle \frown s) \leqslant t \equiv t \neq \langle\rangle \;\&\; x = t_0 \;\&\; s \leqslant t'$

The prefixes of a given sequence are totally ordered.

L6 $s \leqslant u \;\&\; t \leqslant u \implies s \leqslant t \vee t \leqslant u$

If s is a subsequence of t (not necessarily initial), we say s is <u>in</u> t; this may be defined:

$$s \text{ \underline{in} } t = (\exists u, v \,.\, t = u \frown s \frown v).$$

This relation is also a partial ordering, in that it satisfies laws L1 to L4 above. It also satisfies:

L7 $(\langle x\rangle \frown s) \text{ \underline{in} } t \equiv t \neq \langle\rangle \;\&\; ((t_0 = x \;\&\; s \leqslant t') \vee (\langle x\rangle \frown s) \text{ \underline{in} } t')$

A function f from traces to traces is said to be <u>monotonic</u> if it <u>respects the ordering</u> $\leqslant$, i.e.

$f(s) \leqslant f(t)$ whenever $s \leqslant t$.

All distributive functions are monotonic, for example

L8 $s \leqslant t \implies (s \upharpoonright A) \leqslant (t \upharpoonright A)$

A dyadic function may be monotonic in either argument, keeping the other argument constant. For example, catenation is monotonic in its second argument (but not in its first):

L9 $t \leqslant u \implies (s \frown t) \leqslant (s \frown u)$.

1.6.6 Length

The number of elements in the trace t is denoted $\#t$. For example

$$\# \langle x, y, z\rangle = 3$$

The laws which define $\#$ are given

L1 $\#\langle\rangle = 0$

L2 $\#\langle x\rangle = 1$

L3 $\#(s^\frown t) = (\# s) + (\# t)$

The number of occurrences in t of symbols from A is counted by $\#(t \upharpoonright A)$.

L4 $\#(t \upharpoonright (A \cup B)) = \#(t \upharpoonright A) + \#(t \upharpoonright B) - \#(t \upharpoonright (A \cap B))$

L5 $s \leq t \implies \# s \leq \# t$

L6 $\#(t^n) = n \times (\# t)$

1.7 Implementation of Traces

In order to represent traces in a computer and to implement operations on them, we need a high-level list processing language. Fortunately, LISP is very suitable for this purpose. Traces are represented in the obvious way by lists of atoms representing its events:

$$\langle\rangle \;=\; \text{NIL}$$

$$\langle coin\rangle \;=\; \text{cons ("COIN, NIL)}$$

$$\langle coin, choc\rangle \;=\; \text{"(COIN CHOC)}$$

Operations on traces can be readily implemented as functions on lists. For example, the head and tail of a nonempty list are given by the primitive functions "car" and "cdr"

$$t_0 \;=\; \text{car}(t)$$

$$t' \;=\; \text{cdr}(t)$$

$$\langle x\rangle^\frown s \;=\; \text{cons}(x,s)$$

General catenation is implemented as the familiar "append" function, which uses recursion:

$$s^\frown t \;=\; \text{append}(s,t)$$

$$\text{append}(s,t) \;=\; \underline{\text{if}}\ s = \text{NIL}\ \underline{\text{then}}\ t$$

$$\underline{\text{else}}\ \text{cons (car(s), append (cdr(s), t))}$$

The correctness of this definition follows from the laws

$$\langle\rangle^\frown t \;=\; t$$

$$s^\frown t \;=\; \langle s_0\rangle^\frown (s'^\frown t) \qquad \text{when } s \neq \langle\rangle$$

The termination of the append function is guaranteed by the fact that the list supplied as first argument of each recursive call is shorter than it was at the next higher level of recursion. Similar arguments establish the correctness of the implementations of the other operations.

To implement restriction, a set is represented as a function which gives the answer "true" whenever its argument is in the set and "false" otherwise, so that

$$x \in A \equiv A(x) = \text{true}$$

$(s \upharpoonright A)$ can now be implemented as the function:

```
restrict (s,A)  =  if s = NIL then NIL
                   else if A(car(s)) = false then restrict(cdr(s),A)
                        else cons(car(s), restrict(cdr(s),A))
```

A test of $s \leq t$ is implemented as a function which delivers the answer true or false

```
isprefix (s,t) = if s = NIL then true
                 else if t = NIL then false
                      else car(s) = car(t)
                           & isprefix (cdr(s), cdr(t))
```

The implementation of

$$\# s = \text{length}(s)$$

is left as an exercise.

1.6 Traces of a Process

In the previous section, a trace of a process was introduced as a sequential record of the behaviour of a process up to some moment in time. Before the process starts, it is not known which of its possible traces will actually be recorded: that will depend on environmental factors beyond the control of the process. However the complete set of all possible traces of a process can be known in advance, and we define a function "traces(P)" to map each process onto that set.

Examples

X1 The only trace of the behaviour of the process STOP is $< >$. The notebook of the observer of this process remains forever blank.

$$\text{traces}(STOP) = \{< >\}$$

X2 There are only two traces of the machine that ingests a coin before breaking

$$\text{traces}(coin \longrightarrow STOP) = \{< >, <coin>\}$$

X3 A clock does nothing but "tick"

$$\text{traces}(\mu X.tick \longrightarrow X) = \{<>, <tick>, <tick, tick>, \ldots\}$$
$$= \{tick\}^*$$

As with most interesting processes, the set of traces is infinite, although of ~~course each~~ individual trace is finite.

X4 A simple vending machine:

$$\text{traces}(\mu X.coin \longrightarrow choc \longrightarrow X) =$$
$$\{s \mid \exists n. \; s \leq <coin, choc>^n\}$$

1.8.1 Laws

In this section we show how to calculate the set of traces of any process defined using the notations introduced so far. As mentioned above, STOP has only one trace.

L1 $\text{traces}(STOP) = \{t \mid t = < >\}$

A trace of $(c \longrightarrow P)$ may be empty, because $< >$ is a trace of the behaviour of every process up to the moment that it engages in its very first action. Every nonempty trace begins with c, and its tail must be a possible trace of P.

L2 $\text{traces}(c \longrightarrow P) = \{t \mid t = < >$

$$\vee \; t_0 = c \; \& \; t' \in \text{traces}(P)\} .$$
$$= \{<>\} \cup \{<c>^\frown t \mid t \in \text{traces}(P)\}$$

A trace of the behaviour of a process which offers a choice between initial events must be a trace of one of the alternatives:

L3 $\text{traces}(c \longrightarrow P \mid d \longrightarrow Q) =$

$$\{t \mid t = < >$$
$$\vee \; t_0 = c \; \& \; t' \in \text{traces}(P)$$
$$\vee \; t_0 = d \; \& \; t' \in \text{traces}(Q)\} .$$

These three laws are summarised in the single general law governing choice:

L4 $\quad \text{traces}\,(x{:}A \longrightarrow P(x)) \;=\; \{t \mid t = \langle\,\rangle \;\vee\; t_0 \in A \;\&\; t' \in \text{traces}(P(t_0))\}$

To discover the set of traces of a recursively defined process is a bit more difficult. A recursively defined process is the solution of an equation

$$X = F(X)$$

First, we define iteration of the function F by induction.

$$F^0(X) = X$$

$$F^{n+1}(X) = F(F^n(X))$$

$$= F^n(F(X))$$

$$= \underbrace{F(\ldots\,(F(F(X)))}_{n \text{ times}}\;\ldots)$$

Then we can define

L5 $\quad \text{traces}(\mu X{:}A.\; F(X)) \;=\; \bigcap_{n \geq 0} \text{traces}(F^n(RUN_A))$

Finally, it is necessary to define the traces of RUN_A by the obvious law

L6 $\quad \text{traces}(RUN_A) \;=\; A^*$

Examples

X1 $\quad$ Let $F(X) = tick \longrightarrow X$

$\quad\;$ Let $A = \{tick\}$

Then $\text{traces}\,(F^0(RUN_A)) = \text{traces}\,(RUN_A)$ $\qquad$ def F^0

$= \{tick\}^*$ .... $\qquad$ (1) L6

$\text{traces}\,(F^{n+1}(RUN_A)) = \text{traces}\,(F(F^n(RUN_A)))$ $\qquad$ def F^{n+1}

$= \text{traces}\,(tick \longrightarrow F^n(RUN_A))$ $\qquad$ def F

$= \{t \mid t = \langle\,\rangle$ $\qquad$ L2

$\quad \vee (t_0 = tick \wedge t' \in \text{traces}(F^n(RUN_A)))\}$ ... $\quad$ (2)

We now propose the hypothesis that

$$\text{traces}\,(F^n(RUN_A)) = \{tick\}^* \qquad \text{for } \underline{\text{all}}\ n.$$

Proof (1) For $n = 0$, the proof is given above (1).

(2) Assume the hypothesis; it follows from (2) above that

$$\text{traces}\ (F^{n+1}(RUN_A)) = \{t \mid t = \langle\rangle \vee (t_0 = \text{tick} \wedge t' \in \{\text{tick}\}^*)\}$$
$$= \{\text{tick}\}^* \qquad 1.6.4,\ L1\text{-}L3.$$

By L5 we deduce

$$\text{traces}\ (\mu X{:}A.\text{tick} \longrightarrow X) = \bigcap_{n \geqslant 0} \{\text{tick}\}^*$$
$$= \{\text{tick}\}^*$$

X2 Let $F(X) = (\text{coin} \longrightarrow \text{choc} \longrightarrow X)$

Let $A = \{\text{coin, choc}\}$.

Let $VMS = \mu X{:}A.\ F(X)$

We want to prove

$$\text{traces}\ (VMS) = \{tr \mid \forall n.\ \ tr \leqslant \langle \text{coin,choc}\rangle^n \vee (tr \geqslant \langle \text{coin,choc}\rangle^n \wedge tr \in A^*)\}$$

Proof. Let $T_n = \text{traces}\ (F^n(RUN_A))$

By L5, it is sufficient to prove for all n

$$T_n = \{tr \mid tr \leqslant \langle \text{coin,choc}\rangle^n \vee (tr \geqslant \langle \text{coin,choc}\rangle^n \wedge tr \in A^*)\}$$

This is proved by induction on n.

(1) $T_0 = \text{traces}\ (RUN_A)$

$$= \{\text{coin,choc}\}^*$$
$$= \{tr \mid tr \leqslant \langle\ \rangle \vee (tr \geqslant \langle\rangle \wedge tr \in A^*)\}$$

The conclusion follows, since $\langle\rangle = \langle \text{coin,choc}\rangle^0$

(2) $$T_{n+1} = \{tr \mid tr = \langle\ \rangle \vee (tr_0 = \text{coin} \wedge (tr' = \langle\rangle \vee (tr'_0 = \text{choc} \wedge tr'' \in T_n)))\}$$

$$= \{tr \mid tr \leqslant \langle \text{coin,choc}\rangle \vee (tr \geqslant \langle \text{coin,choc}\rangle \wedge tr'' \in T_n)\}$$

$$= \{ tr \mid tr \leq \langle coin, choc \rangle^1 \vee (tr \geq \langle coin, choc \rangle^1 \wedge (tr'' \leq \langle coin, choc \rangle^n \vee (tr'' \geq \langle coin, choc \rangle^n \wedge tr'' \in A^*))) \}$$ by induction hypothesis

$$= \{ tr \mid tr \leq \langle coin, choc \rangle^{n+1} \vee tr \geq \langle coin, choc \rangle^{n+1} \wedge tr \in A^* \}.$$

since $tr \leq s \vee (tr \geq s \wedge tr'' \leq s^n) \equiv tr \leq s^{n+1}$

and $tr \geq s \wedge tr'' \geq s^n \equiv tr \geq s^{n+1}$, wherever $\# s = 2$

As mentioned in 1.5, a trace is a sequence of symbols recording the events in which a process P has engaged up to some moment in time. From this it follows that $\langle\rangle$ is a trace of every process up to the moment in which it engages in its very first event. Furthermore, if $s \frown t$ is a trace of a process up to some moment then s must have been a trace of that process up to some earlier moment. Finally, every event that occurs must be in the alphabet of the process. These three facts are formalised in the laws:

L7 $\langle\rangle \in traces\ (P)$

L8 $s \frown t \in traces\ (P) \Rightarrow s \in traces\ (P)$

L9 $traces\ (P) \subseteq \alpha P^*$

1.8.2 Implementation

Suppose a process has been implemented as a LISP function P, and let s be a trace. Then it is possible to test whether s is a possible trace of P by the function

```
istrace (s,P)  =  if s = NIL then true
                  else if P(s_0) = "BLEEP then false
                       else istrace (s',P(s_0))
```

Since s is finite, the recursion involved here will terminate, having explored only a finite initial segment of the behaviour of the process P. It is only because we avoid infinite exploration that we can safely define a process as an "infinite" object, i.e., a function whose result is function whose result is function whose [illegible]

1.8.3 After

If $s \in$ traces (P) then

P/s (P <u>after</u> s)

is a process which behaves the same as P behaves from the time after it has engaged in all the actions of s. If s is not a trace of P, P/s is not defined.

Examples

X1 $(VMS/\langle coin \rangle) = (choc \longrightarrow VMS)$

X2 $VMS/\langle coin, choc \rangle = VMS$

X3 $VMC/\langle in1p \rangle^3 = STOP$

The following laws describe the meaning of the operator /. After doing nothing, a process remains unchanged.

L1 $P/\langle\rangle = P$

After engaging in $s^\frown t$, the behaviour of P is the same as that of (P/s) would be after engaging in t

L2 $P/(s^\frown t) = (P/s)/t$

After engaging in the single event c, the behaviour of a process is as defined by this initial choice.

L3 $(x{:}A \longrightarrow P(x))/\langle c \rangle = P(\langle c \rangle)$ provided that $c \in A$

A corollary shows that $(/\langle c \rangle)$ is the inverse of the prefixing operator $(c \longrightarrow)$.

L3A $(c \longrightarrow P)/\langle c \rangle = P$

The traces of (P/s) are defined

$$\text{traces}(P/s) = \{t \mid s^\frown t \in \text{traces}(P)\}$$

provided that $s \in$ traces (P).

In order to prove that a process P never stops it is sufficient to prove that

$P/s \neq STOP$ for all $s \in$ traces (P).

Another desirable property of a process is <u>liveness</u>; a process P is defined as live if in all circumstances it is possible for it to return to its initial state, i.e.,

$$\forall s \in \text{traces}(P).\ \exists t.\ (P/(s^\frown t) = P)$$

STOP is trivially live; but if any other process is live then it also has the desirable property of never stopping.

Examples

X1 The following processes are live

FCH_A, VMS, (choc $\longrightarrow$ VMS), VMCT, CT_7

X2 The following are not live, because it is not possible to return them to their initial state:

(coin $\longrightarrow$ VMS), (choc $\longrightarrow$ VMCT), (around $\longrightarrow$ CT_7)

In the initial state of (choc $\longrightarrow$ VMCT), only a chocolate is obtainable, but subsequently whenever choc is obtainable a choice of toffee is also possible; consequently none of these subsequent states is equal to the initial state.

Warning. The use of / in a recursively defined process has the unfortunate consequence of invalidating its guards, thereby introducing the danger of multiple solutions to the recursion equations. For example

$$X = (a \longrightarrow (X/\langle a\rangle))$$

is not guarded, and has as its solution <u>any</u> process of the form

$$a \longrightarrow P$$

for any P.

Proof. $(a \longrightarrow ((a \longrightarrow P)/\langle a\rangle)) = (a \longrightarrow P)$ by L34.

1.9 More operations on traces

This section describes some further operations on traces; it may be skipped at this stage, since backward references will be given in later chapters where the operations are used.

1.9.1 Change of symbol

Let f be a function mapping symbols from a set A to symbols in a set B. From f we can derive a new function f^* which maps a sequence of symbols in A^* to a sequence in B^* by applying f to each element of the sequence. For example, if double is a function which doubles its integer argument,

$$\text{double}^*(\langle 1,5,3,1\rangle) = \langle 2,10,6,2\rangle$$

A starred function is obviously strict and distributive:

L1 $f^*(\langle\rangle) = \langle\rangle$

L2 $f^*(\langle x\rangle) = \langle f(x)\rangle$

L3 $f^*(s^\frown t) = f^*(s)^\frown f^*(t)$

Other laws are obvious consequences

L4 $f^*(s)_0 = \langle f(s_0)\rangle$ if $s \neq \langle\rangle$

L5 $\#f^*(s) = \#s$

But here is an "obvious" law which is unfortunately not generally true

$$f^*(s\restriction A) = f^*(s)\restriction f(A)$$

where $f(A) = \{f(x) \mid x \in A\}$

The simplest counterexample is given by ~~the~~ function f such that

$$f(b) = f(c) = c$$

$$\therefore\quad f^*(\langle b\rangle\restriction\{c\}) = f^*(\langle\rangle)$$
$$= \langle\rangle$$
$$\neq \langle c\rangle$$
$$= \langle c\rangle\restriction\{c\}$$
$$= f^*(\langle c\rangle)\restriction f(\{c\})$$

However, the law is true if f is a one-one function.

L6 $f^*(s\restriction A) = f^*(s)\restriction f(A)$

provided that f is an injection.

1.9.2 Catenation and zip

Let s be a sequence, each of whose elements is itself a sequence. Then $^\frown/s$ is obtained by catenating all the elements together in the original order.

For example

$$^\frown/\langle\langle 1,3\rangle,\langle\rangle,\langle 7\rangle\rangle = \langle 1,3\rangle^\frown\langle\rangle^\frown\langle 7\rangle$$
$$= \langle 1,3,7\rangle$$

This operator is strict and distributive:

L1 $\hat{}/\langle\rangle = \langle\rangle$

L2 $\hat{}/\langle s\rangle = s$

L3 $\hat{}/(s\hat{}t) = (\hat{}/s)\hat{}(\hat{}/t)$

The function zip gives a sequence formed by taking alternately the elements of each of its two operands. Thus

$\mathrm{zip}(\langle 1,3,5\rangle,\langle 6,1,4\rangle) = \langle 1,6,3,1,5,4\rangle$

The function is totally defined by:

L4 $\mathrm{zip}(\langle\rangle, t) = \langle\rangle$

L5 $\mathrm{zip}(\langle x\rangle\hat{}s, t) = \langle x\rangle\hat{}\,\mathrm{zip}(t,s)$

1.9.3 Interleaving

A sequence s is an interleaving of two sequences t and u if it can be split into a series of subsequences, with alternate subsequences extracted from t and u. For example

$s = \langle 1,6,3,1,5,4,2,7\rangle$

is an interleaving of

$t = \langle 1,6,5,2,7\rangle$ and $u = \langle 3,1,4\rangle$

because $s = \hat{}/\langle\langle 1,6\rangle,\langle 3,1\rangle,\langle 5\rangle,\langle 4\rangle,\langle 2,7\rangle\rangle$

and $t = \hat{}/\langle\langle 1,6\rangle,\langle 5\rangle,\langle 2,7\rangle\rangle$

and $u = \hat{}/\langle\langle 3,1\rangle,\langle 4\rangle,\langle\rangle\rangle$

This example suggests how a definition of interleaving can be formulated in terms of catenation and zip.

$$s \text{ interleaves } (t,u) = \exists T,U.\ s = \hat{}/\mathrm{zip}(T,U) \wedge t = \hat{}/T \wedge u = \hat{}/U.$$

A more constructive definition of interleaving can be given by means of the following laws.

L1 $\langle\rangle$ interleaves $(t,u) \equiv t = \langle\rangle \wedge u = \langle\rangle$

L2 s interleaves $(t,u) \equiv$ s interleaves (u,t)

L3 $(\langle x\rangle\hat{}s)$ interleaves $(t,u) \equiv$

$(t \neq \langle\rangle \wedge t_0 = x \wedge s$ interleaves $(t',u))$

$\vee\ (u \neq \langle\rangle \wedge u_0 = x \wedge s$ interleaves $(t,u'))$

1.9.4 Subscription

If $0 \leq i < \#s$, we use the conventional notation $s[i]$ to denote the i^{th} element of the sequence s.

L1 If $s \neq \langle\,\rangle$

$s[0] = s_0 \wedge s[i+1] = s'[i]$

L2 $f^*(s)[i] = f(s[i])$ for $i < \#s$

1.9.5 Reversal

If s is a sequence $\overline{s}$ is formed from its elements in reverse order. For example

$\overline{\langle 3,5,37\rangle} = \langle 37,5,3\rangle$

Reversal is defined fully by the following laws

L1 $\overline{\langle\,\rangle} = \langle\,\rangle$

L2 $\overline{\langle x\rangle} = \langle x\rangle$

L3 $\overline{s\frown t} = \overline{t}\frown\overline{s}$

Reversal enjoys a number of simple algebraic properties, including

L4 $\overline{\overline{s}} = s$

Exploration of other properties is left to the reader. One of the useful facts about reversal is that $\overline{s}_0$ is the <u>last</u> element of the sequence, and in general

L5 $\overline{s}[i] = s[\#s - i - 1]$

1.10 Specifications

A specification of a product is a description of the way it is intended to behave. This description is a predicate containing free variables, each of which stands for some observable aspect of the behaviour of the product. For example, the specification of an electronic amplifier, with an input range of one volt and with an approximate gain of 10 could be given by the mathematical predicate:

$AMPL = (0 \leq v \leq 1 \Longrightarrow |v' - 10 * v| \leq 1)$

In this specification, it is understood that v stands for the input voltage and v' stands for the output voltage. Such an understanding of the meaning of variables is essential to the use of mathematics in science and engineering.

In the case of a process, the most obviously relevant observation of its behaviour is of the trace of events that occur up to a given moment in time. We will use the special variable "tr" to stand for an arbitrary trace of the process being specified.

Examples

X1 The owner of a vending machine does not wish to make a loss by installing it. He therefore specifies that the number of chocolates dispensed must never exceed the number of coins inserted:

$$NOLOSS = (\#(tr \restriction \{choc\}) \leq \#(tr \restriction \{coin\}))$$

In future, we will use the abbreviation

$$tr.c = \#(tr \restriction \{c\})$$

to stand for the number of occurrences of the symbol c in tr.

X2 The customer of a vending machine wants to ensure that it will not absorb further coins until it has dispensed the chocolate already paid for:

$$FAIR1 = tr.coin \leq tr.choc + 1$$

X3 The manufacturer of a simple vending machine must meet the requirements both of its owner and of its customer.

$$VMSPEC = NOLOSS \wedge FAIR1$$

$$= (0 \leq tr.coin - tr.choc \leq 1)$$

X4 The specification of a correction to the complex vending machine forbids it to accept three coins in a row

$$VMCFIX = (\neg \langle in1p \rangle^3 \ \underline{in} \ tr)$$

X5 The specification of a mended machine

$$MENDVMC = (tr \in traces\ (VMC) \wedge VMCFIX)$$

1.10.1 Satisfaction

If P is a product which meets a specification S, we say that

P sat S (P satisfies S)

This means that every possible observation of the behaviour of P is described by S; or in other words, S is true whenever its variables take values observed from the product P. For example, the following table gives some observations of the properties of an amplifier

v	v'
0	0
$\frac{1}{2}$	5
$\frac{1}{2}$	4
2	1
$\frac{1}{10}$	3

All observations except the last are described by AMP10. The second and third lines illustrate the fact that the output of the amplifier is not completely determined by its input. The fourth line shows that if the input voltage is outside its specified range, the output voltage can be anything at all, without violating the specification. (In this simple example we have ignored the possibility that excessive input may break the product.)

The following laws give the most general properties of the "satisfies" relation. The specification "true" which places no constraints whatever on observations of a product will be satisfied by all products; even a broken product satisfies such a weak and undemanding specification.

L1 P <u>sat</u> true

If a product satisfies two different specifications, it also satisfies their conjunction

L2A If P <u>sat</u> S
and P <u>sat</u> T
then P <u>sat</u> $(S \wedge T)$

The law L2A generalises to infinite conjunctions, i.e., to universal quantification. Let S(n) be a predicate containing the variable n.

L2 If $\forall n.$ (P <u>sat</u> S(n))
then P <u>sat</u> ($\forall n.S(n)$)

provided that P <u>does not</u> contain n.

If a specification S logically implies another specification T, then every observation described by S is also described by T. Consequently every product which satisfies S must also satisfy the weaker specification T

L3 If P sat S

and $S \Longrightarrow T$

then P sat T

1.10.2 Proofs

In the design of a product, the designer has a responsibility to ensure that it will satisfy its specification; for this purpose he will wherever possible use the reasoning methods of the relevant branches of mathematics, for example, geometry or the differential and integral calculus. In this section we shall give a calculus which permits the use of mathematical reasoning to ensure that a process P meets its specification S.

Any observation of the process STOP will always be of an empty trace, since this process never does anything.

L4A STOP sat $tr = \langle \rangle$

trace of the process $(c \longrightarrow P)$ is initially empty. Every subsequent trace begins with c, and its tail is a trace of P. Consequently its tail must be described by any specification of P.

L4B If P sat S(tr)

then $(c \longrightarrow P)$ sat $(tr = \langle \rangle$

$$\vee (tr_0 = c \wedge S(tr')))$$

A corollary of this law deals with double prefixing.

L4C If P sat S(tr)

then $(c \longrightarrow d \longrightarrow P)$ sat $(tr \leq \langle c,d \rangle$

$$\vee (tr \geq \langle c,d \rangle \wedge S(tr'')))$$

Binary choice is similar to prefixing, except that the trace may begin with either of the two alternative events, and its tail must be described by the specification of the chosen alternative.

L4D If P sat $S(tr)$

and Q sat $T(tr)$

then $(c \longrightarrow P \mid d \longrightarrow Q)$ sat $(tr = \langle\rangle$

$\vee (tr_0 = c \wedge S(tr'))$

$\vee (tr_0 = d \wedge T(tr')))$

All the laws given above are special cases of the law for general choice.

L4 If $\forall x \in A.\ (P(x)$ sat $S(tr,x))$

then $(x{:}A \longrightarrow P(x))$ sat $(tr = \langle\rangle$

$\vee (tr_0 \in A \wedge S(tr', tr_0)))$

The law governing the after operator is surprisingly simple. If tr is a trace of (P/s), $s^\frown tr$ must be a trace of P, and therefore must be described by any specification which P satisfies.

L5 If P sat $S(tr)$

and $s \in$ traces (P)

then (P/s) sat $S(s^\frown tr)$

Finally, we need a law to establish the correctness of a recursively defined process. Let $S(n)$ be a predicate containing the variable n, which ranges over the natural numbers 0,1,2, ...

L6 If $S(0)$

and $(X$ sat $S(n)) \Longrightarrow (F(X)$ sat $S(n+1))$

then $(\mu X.\ F(X))$ sat $\forall n.\ S(n))$

The justification of this law is as follows. Suppose we have proved the two antecedants of L6.

Then $F^0(RUN_A)$ sat $S(0)$ by L1

and $(F^n(RUN_A)$ sat $S(n)) \Longrightarrow (F^{n+1}(RUN_A)$ sat $S(n+1))$

By induction we can conclude

$F^n(RUN_A)$ sat $S(n)$ for all n.

Consider now an arbitrary trace of $\mu X.\ F(X)$. This trace must be also a trace of $F^n(RUN_A)$ for <u>all</u> n. By the conclusion of the above induction, it is described by $S(n)$, for all n. It is therefore described by $\forall n.\ S(n)$. This argument applies to all traces of $\mu X.\ F(X)$, and justifies the conclusion of L6.

Example

X1 We shall prove the obvious fact that

$$VMS\ \underline{sat}\ VMSPEC$$

Since VMS is defined by recursion, we shall need a suitable induction hypothesis $S(n)$, mentioning the induction variable n. In the case of a guarded recursion, a simple but effective technique is to add a clause to the specification:

$$S(n) \;=\; \#tr \geqslant n \vee VMSPEC$$

Since $\#t \geqslant 0$ is always true, so is $S(0)$; this gives the basis of the induction.

Now assume $X\ \underline{sat}\ S(n)$

$$\begin{aligned}
\therefore\ & (coin \longrightarrow choc \longrightarrow X)\ \underline{sat} \\
& \qquad (tr \leqslant \langle coin, choc\rangle \\
& \qquad \vee (tr \geqslant \langle coin, choc\rangle \\
& \qquad\qquad \wedge (\#tr'' \geqslant n \vee 0 \leqslant tr''.coin - tr''.choc \leqslant 1))) \\
& \Longrightarrow (\#tr \geqslant n+1 \vee 0 \leqslant tr.coin - tr.choc \leqslant 1) \\
& \equiv S(n+1)
\end{aligned}$$

This establishes the induction step of the proof:

$$(X\ \underline{sat}\ S(n)) \Longrightarrow ((coin \longrightarrow choc \longrightarrow X)\ \underline{sat}\ S(n+1))$$

$$\begin{aligned}
\therefore\ \mu X.(coin \longrightarrow choc \longrightarrow X)\ \underline{sat}\ & (\forall n.\ \#tr \geqslant n \ \vee VMSPEC) \\
& \equiv (\forall n.\#tr \geqslant n) \vee VMSPEC \\
& \equiv VMSPEC
\end{aligned}$$

since the length of a trace must be finite.

The fact that a process P satisfies its specification does not necessarily mean that it is going to be satisfactory in use. For example, since

$$tr = \langle\rangle \Longrightarrow 0 \leqslant tr.coin - tr.choc \leqslant 1$$

one can prove by L3 and L4A that

$$\text{STOP } \underline{\text{sat}}\ 0 \leqslant tr.coin - tr.choc \leqslant 1.$$

Yet STOP will not serve as an adequate vending machine, either for its owner or for the customer. It certainly avoids doing anything wrong; but only by the lazy expedient of doing nothing at all. For this reason, STOP satisfies every specification which is satisfiable by any process.

Fortunately, it is obvious that VMS will never stop. In fact any process defined solely by prefixing, choice, and guarded recursions will never stop. The only way to write a process that can stop is to include explicitly ~~the process~~ STOP, or equivalently the process $(x:A \longrightarrow P(x))$ where A is the empty set. By avoiding such ~~elementary~~ mistakes one can guarantee to write processes that never stop.

CHAPTER TWO

CONCURRENT PROCESSES

2.1 Introduction

A process is defined by describing the whole range of its potential behaviour. Frequently, there will be a choice between several different actions, for example, the insertion of a large coin or a small one into a vending machine VMC (1.1.3.X4). On each such occasion, the choice of which event will actually occur can be controlled by the environment within which process evolves. For example, it is the customer of the vending machine who may select what coin to insert. Fortunately, the environment of a process itself may be described as a process, with its behaviour defined by familiar notations. This permits investigation of the behaviour of a complete system composed from the process and its environment, acting and interacting with each other as they evolve concurrently. The complete system should also be regarded as a process, whose range of behaviour is definable in terms of the behaviour of its component processes; and the system may in turn be placed within a yet wider environment. In fact, it is best to forget the distinction between processes, environments, and systems; they are all of them just processes whose behaviour may be prescribed, described, recorded and analysed in a simple and homogeneous fashion.

2.2 Intersection

When two processes are brought together to evolve concurrently, the usual intention is that they will interact with each other. These interactions may be regarded as events that require simultaneous participation of both the processes involved. For the time being, let us confine attention to such events, and ignore all others. Thus we will assume that the alphabets of the two processes are the same. Consequently, each event that actually occurs must be a possible event in the independent behaviour of each process separately. For example, a chocolate can be extracted from a vending machine only when its customer wants it and only when the vending machine is prepared to give it. If P and Q are processes with the same alphabet, we introduce the notation

$$P \,\|\, Q$$

to denote the process which behaves like the system composed of processes P and Q interacting in the manner described above.

Examples

X1 The greedy customer.

A certain customer of a vending machine is perfectly happy to obtain a toffee or even a chocolate without paying. However, if thwarted in these desires, he is reluctantly prepared to pay a coin, but then he insists on taking a chocolate.

$$\begin{array}{lll} \mathrm{GRCUST} & = & (toffee \longrightarrow \mathrm{GRCUST} \\ & & \mid choc \longrightarrow \mathrm{GRCUST} \\ & & \mid coin \longrightarrow choc \longrightarrow \mathrm{GRCUST}) \end{array}$$

When this customer is brought together with the machine VMCT (1.1.3.X3) all his greed is frustrated, since the vending machine does not allow goods to be extracted before payment.

$$(\mathrm{GRCUST} \,\|\, \mathrm{VMCT}) = \mu X.(coin \longrightarrow choc \longrightarrow X)$$

This example shows how a process which has been defined as a composition of two subprocesses may also be described as a simple single process, without using the concurrency operator $\|$.

X2 The foolish customer.

A foolish customer wants a large biscuit, so he puts his coin in the vending machine VMC. He does not notice whether he has inserted a large coin or a small one; nevertheless, he is determined on a large biscuit.

$$\begin{array}{lll} \mathrm{FOOLCUST} & = & (in2p \longrightarrow large \longrightarrow \mathrm{FOOLCUST} \\ & & \mid in1p \longrightarrow large \longrightarrow \mathrm{FOOLCUST}) \end{array}$$

Unfortunately, the vending machine is not prepared to yield a large biscuit for only a small coin.

$$\begin{array}{lll} (\mathrm{FOOLCUST} \,\|\, \mathrm{VMC}) & = & \mu X.(in2p \longrightarrow large \longrightarrow X \\ & & \quad \mid in1p \longrightarrow \mathrm{STOP}). \end{array}$$

The STOP that supervenes after the first in is known as <u>deadlock</u>. Although each component process is prepared to engage in some further action, these actions are different; since the processes cannot agree on what the next action shall be, nothing further can happen.

These examples show a sad betrayal of proper standards of scientific abstraction and objectivity. It is important to remember that events are intended to be neutral transitions which could be observed and recorded by some dispassionate visitor from another planet, who knows nothing of the pleasures of eating biscuits, or of the hunger suffered by the foolish customer as he vainly tries to obtain sustenance. We have deliberately chosen the alphabet of relevant events to exclude such internal emotional states; if and when desired, further events can be introduced to model "internal" state changes, as shown in section 2.3.

2.2.1 Laws

The laws governing the behaviour of $(P \| Q)$ are exceptionally simple and regular. The first law expresses the logical symmetry between a process and its environment.

L1 $\quad P \| Q = Q \| P$ (symmetry)

The next law shows that when three processes are assembled, it does ~~not matter in which order they are put together.~~

L2 $\quad P \| (Q \| R) = (P \| Q) \| R$ (associativity)

Thirdly, a deadlocked process infects the whole system with deadlock.

L3 $\quad P \| STOP_{\alpha P} = STOP_{\alpha P}$ (zero)

The next laws show how a pair of processes either engage simultaneously on the same action, or deadlock if they disagree on what the first action should be:

L4A $\quad (c \rightarrow P) \| (c \rightarrow Q) = (c \rightarrow (P \| Q))$

L4B $\quad (c \rightarrow P) \| (d \rightarrow Q) = STOP$ if $c \neq d$

This law readily generalises to cases when one or both processes offer a choice of initial event; only events which they both offer will remain possible when the processes are combined:

L4 $\quad (x:A \rightarrow P(x)) \| (y:B \rightarrow Q(y))$

$\quad\quad = (z:(A \cap B) \rightarrow (P(z) \| Q(z)))$

It is this law which permits a system defined in terms of concurrency to be given an alternative description without concurrency.

X1 Let $P = (a \longrightarrow b \longrightarrow P \mid c \longrightarrow P)$

$Q = (a \longrightarrow (b \longrightarrow Q \mid c \longrightarrow Q))$

$(P \| Q) = a \longrightarrow ((c \longrightarrow P) \| (b \longrightarrow Q \mid c \longrightarrow Q))$ by L4

$= a \longrightarrow (c \longrightarrow (P \| Q))$ by L4

$= \mu X\ (a \longrightarrow c \longrightarrow X)$ since the recursion is guarded.

2.2.2 Implementation

The implementation of the $\|$ combinator is clearly based on L4

intersect (P,Q) = λz.

<u>if</u> P(z) = "BLEEP $\vee$ Q(z) = "BLEEP <u>then</u> "BLEEP

<u>else</u> intersect (P(z), Q(z))

2.2.3 Traces

Since each action of $(P \| Q)$ requires simultaneous participation of both P and Q, each sequence of such actions must be possible for both these operands:

L1 traces $(P \| Q)$ = traces (P) $\cap$ traces (Q).

L2 $(P \| Q)/s = (P/s) \| (Q/s)$

2.3 Concurrency

The operator described in the previous section can be generalised to the case when its operands P and Q have different alphabets:

$\alpha P \neq \alpha Q$

When such processes are assembled to run concurrently, events that are in both their alphabets (as explained in the previous section) require simultaneous participation of both P and Q. However events in the alphabet of P but not in the alphabet of Q are of no concern to Q, which is physically incapable of controlling or even of noticing them. Such events may occur independently of Q whenever P engages in them. Similarly, Q may engage alone in events which are in the alphabet of Q but not of P. Thus the set of all events that are logically possible for the system is simply the union of the alphabets of the component processes:

$$\alpha(P \| Q) = \alpha P \cup \alpha Q$$

This is a rare example of an operator which takes operands with different alphabets, and yields a result with yet a third alphabet. However in the case when the two operands have the same alphabet, so does the resulting combination, and $(P \| Q)$ has exactly the meaning described in the previous section.

Examples

X1 Let $\alpha NOISYVM = \{coin, choc, clink, clunk, toffee\}$

where "clink" is the sound of a coin dropping into the moneybox of a noisy vending machine

and "clunk" is the sound made by the vending machine on completion of a transaction.

The noisy vending machine has run out of toffee:

$$NOISYVM = (coin \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow NOISYVM)$$

Let $\alpha CUST = \{coin, choc, curse, toffee\}$

The customer of this machine definitely prefers toffee; and "curse" is what he does when he fails to get it; he then has to take a chocolate instead.

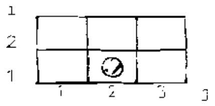
$$\begin{aligned} CUST = (coin \rightarrow (&toffee \rightarrow CUST \\ &| curse \rightarrow choc \rightarrow CUST) \end{aligned}$$

The result of the concurrent activity of these two processes is:

$$\begin{aligned} (NOISYVM \| CUST) = \\ \mu X.(coin \rightarrow (&clink \rightarrow curse \rightarrow choc \rightarrow clunk \rightarrow X \\ &| curse \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow X)) \end{aligned}$$

Note that the relative ordering of the "clink" and the "curse" is not determined. They may even occur simultaneously, and it will not matter in which order they are recorded.

X2



A counter starts at the middle bottom square of the board, and may move within the board either "up", "down", "left" or "right".

Let $\alpha P = \{up, down\}$

$P = (up \longrightarrow down \longrightarrow P)$

$\alpha Q = \{left, right\}$

$Q = (right \longrightarrow left \longrightarrow Q$
$\mid left \longrightarrow right \longrightarrow Q)$

The behaviour of the counter may be defined

$$P \| Q$$

In this example, the alphabets αP and αQ have no event in common. Consequently, the movements of the counter are an arbitrary interleaving of actions from the process P with actions from the process Q. Such interleavings are very laborious to describe without concurrency. Mutual recursion is usually needed: just introduce a process for each state of the system. For example, let R_{ij} stand for the behaviour of a counter (X2) when ~~situated~~ in row i and column j of the board, for $i \in \{1,2\}$, $j \in \{1,2,3\}$.

Then $(P \| Q) = R_{12}$, where

$R_{21} = (down \longrightarrow R_{11} \mid right \longrightarrow R_{22})$

$R_{11} = (up \longrightarrow R_{21} \mid right \longrightarrow R_{12})$

$R_{22} = (down \longrightarrow R_{12} \mid left \longrightarrow R_{21} \mid right \longrightarrow R_{23})$

$R_{12} = (up \longrightarrow R_{22} \mid left \longrightarrow R_{11} \mid right \longrightarrow R_{13})$

$R_{23} = (down \longrightarrow R_{13} \mid left \longrightarrow R_{22})$

$R_{13} = (up \longrightarrow R_{23} \mid left \longrightarrow R_{12})$

2.3.1 Laws

The laws for the extended form of concurrency are similar to those for intersection.

L1,2 $\|$ is symmetric and associative.

L3 $P \| STOP_{\alpha P} = STOP_{\alpha P}$

Let $a \in (\alpha P - \alpha Q)$, $b \in (\alpha Q - \alpha P)$ and $\{c,d\} \subseteq (\alpha P \cap \alpha Q)$

The following laws show the way in which P engages alone in a, Q engages alone in b, but c and d require simultaneous participation of both P and Q.

L4 $(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q)$

$(c \rightarrow P) \parallel (d \rightarrow Q) = STOP$ if $c \neq d$.

L5 $(a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q))$

$(c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q)$

L6 $(a \rightarrow P) \parallel (b \rightarrow Q) = (a \rightarrow (P \parallel (b \rightarrow Q))$
$\quad | \, b \rightarrow ((a \rightarrow P) \parallel Q))$

These laws can be generalised at the expense of some complexity, to deal with the general choice operator:

L7 Let $P = (x{:}A \rightarrow P(x))$

$Q = (y{:}B \rightarrow Q(y))$

Then $(P \parallel Q) = (z{:}C \rightarrow P' \parallel Q')$

where $C = A \cap B \cup (A - \alpha Q) \cup (B - \alpha P)$

$P' = P(z)$ if $z \in A$
$\quad P$ otherwise

and $Q' = Q(z)$ if $z \in B$
$\quad Q$ otherwise.

These laws permit a process defined by concurrency to be re-defined without that operator, as shown in the following example.

X1 Let $\alpha P = \{a,c\}$, $\alpha Q = \{b,c\}$

$P = (a \rightarrow c \rightarrow P)$

$Q = (c \rightarrow b \rightarrow Q)$

$P \parallel Q = (a \rightarrow c \rightarrow P) \parallel (c \rightarrow b \rightarrow Q)$ by definition

$= a \rightarrow ((c \rightarrow P) \parallel (c \rightarrow b \rightarrow Q))$ by L5

$= a \rightarrow c \rightarrow (P \parallel (b \rightarrow Q))$ by L4 ... (1)

$P \parallel (b \rightarrow Q) = (a \rightarrow (c \rightarrow P) \parallel (b \rightarrow Q)$
$\quad | \, b \rightarrow (P \parallel Q))$ by L6

$= (a \rightarrow b \rightarrow ((c \rightarrow P) \parallel Q)$
$\quad | \, b \rightarrow (P \parallel Q))$ by L5

$= (a \rightarrow b \rightarrow c \rightarrow (P \parallel (b \rightarrow Q))$
$\quad | \, b \rightarrow a \rightarrow c \rightarrow (P \parallel (b \rightarrow Q)))$ by L4 and (1)

$$= \mu X.(a \rightarrow b \rightarrow c \rightarrow X \;|\; b \rightarrow a \rightarrow c \rightarrow X) \quad \text{since this is guarded.}$$

$$\therefore \quad (P \,||\, Q) = (a \rightarrow c \rightarrow \mu X.(a \rightarrow b \rightarrow c \rightarrow X \;|\; b \rightarrow a \rightarrow c \rightarrow X)) \quad \text{by (1)}$$

2.3.2 Implementation

The implementation of the operator $||$ is derived directly from the law L7. The alphabets of the operands are represented as finite lists of symbols, A and B. Test of membership uses the function

```
ismember (x, A)  =  if null (A) then false
                    else if  x  =  car(A) then true
                                   else ismember (x, cdr(A)).
```

$(P \,||\, Q)$ is implemented by calling a function

```
        concurrent (P, αP, αQ, Q)
```

which is defined as follows:

```
        concurrent (P, A, B, Q)  =  aux (P,Q)
    where  aux (P,Q)  =  λx.
        if P = "BLEEP  or Q = "BLEEP then "BLEEP
    else if ismember (x,A)  and ismember (x,B)
                     then  aux (P(x), Q(x))
    else if ismember (x,A) then aux (P(x), Q)
    else if ismember (x,B) then aux (P, Q(x))
                              else "BLEEP
```

2.3.3 Traces

Let t be a trace of $(P \,||\, Q)$. Then every event in t which belongs to the alphabet of P has been an event in the life of P; and every event in t which does not belong to αP has occurred without the participation of P. Thus $(t \upharpoonright \alpha P)$ is a trace of all those events in which P has participated, and must therefore be a trace of P. By a similar argument $(t \upharpoonright \alpha Q)$ must be a trace of Q. Furthermore, every event in t must be in either αP or αQ. This reasoning suggests the definition

L1 $\text{traces}(P \parallel Q) = \{t \mid (t\restriction\alpha P) \in \text{traces}(P)$

$\& (t\restriction\alpha Q) \in \text{traces}(Q)$

$\& t \in (\alpha P \cup \alpha Q)^* \}$

L2 $(P \parallel Q)/s = (P/(s\restriction\alpha P)) \parallel (Q/(s\restriction\alpha Q))$

Example

X1 See 2.3 X1.

Let $t1 = \langle coin, clink, curse\rangle$

Then $t1\restriction\alpha NOISYVM = \langle coin, clink\rangle \in \text{traces}(NOISYVM)$

$t1\restriction\alpha CUST = \langle coin, curse\rangle \in \text{traces}(CUST)$

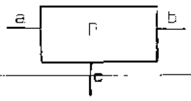
$\therefore$ $t1 \in \text{traces}(NOISYVM \parallel CUST)$

The same reasoning shows that

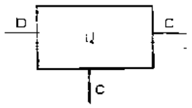
$\langle coin, curse, clink\rangle \in \text{traces}(NOISYVM \parallel CUST)$.

2.4 Pictures

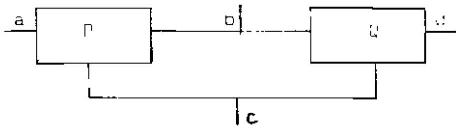
A process P with alphabet $\{a,b,c\}$ is pictured as a box labelled P, from which emerge a number of lines, each labelled with a different event from its alphabet:



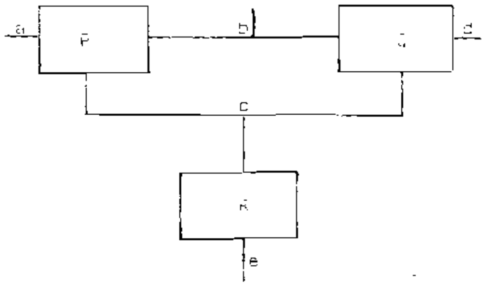
Similarly, Q with alphabet $\{b,c,d\}$ may be pictured:



When these two processes are put together to evolve concurrently, the resulting system may be pictured as a network in which similarly labelled lines are connected, but lines labelled by events in the alphabet of only one process are left free

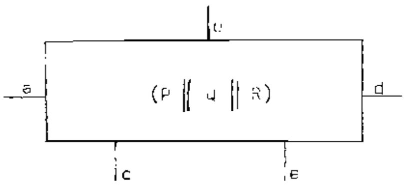


A third process R with $\alpha R = \{c, e\}$ may be added:



This diagram shows that the event c requires participation of all three processes, b requires participation of P and Q, whereas each remaining event is the sole concern of a single process.

But these pictures could be quite misleading. A system constructed from three processes is still only a single process, and should therefore be pictured as a single box



The number 60 can be constructed as the product of three other numbers (3 × 4 × 5); but after it has been so constructed it is still only a single number, and the manner of its construction is no longer relevant or even observable.

2.5 Example: the Dining Philosophers

In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. The names of the philosophers were $Phil_0$, $Phil_1$, $Phil_2$, $Phil_3$, $Phil_4$ and they were disposed in this order anticlockwise round the table. To the left of each philosopher there was laid a golden fork, and in the centre a large bowl of spaghetti, which was constantly replenished.

A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went to the dining room, sat down in his own chair, picked up his own fork on his left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second fork is required to carry it to the mouth. The philosopher therefore had also to pick up the fork on his right. When he was finished he would put down both his forks, get up from his chair, and continue thinking. Of course, a fork can be used by only one philosopher at a time. If the other philosopher wants it, he just has to wait until the fork is available again.

2.5.1 Alphabets

We shall now construct a mathematical model of this system. First we must select the relevant sets of events. For Phil_i, the set is defined:

$$\begin{aligned}\alpha\mathrm{Phil}_i = \{&\text{i sits down, i gets up,}\\ &\text{i picks up fork i, i picks up fork } (i \oplus 1),\\ &\text{i puts down fork i, i puts down fork } (i \oplus 1)\}\end{aligned}$$

where $\oplus$ is addition modulo 5.

Note that the alphabets of the philosophers are mutually disjoint. There is no event in which they can agree to participate jointly, so there is no way whatsoever in which they can interact or communicate with each other – a realistic reflection of the behaviour of philosophers of those days.

The other actors in our little drama are the five forks, each of which bears the same number as the philosopher who owns it. A fork is picked up and put down either by this philosopher, or by his neighbour on the other side. Its alphabet is defined

$$\begin{aligned}\alpha\mathrm{Fork}_i \triangleq \{&\text{i picks up fork i, } (i \ominus 1) \text{ picks up fork i,}\\ &\text{i puts down fork i, } (i \ominus 1) \text{ puts down fork i}\}\end{aligned}$$

where $\ominus$ denotes subtraction modulo 5.

Thus each event except sitting down and getting up requires participation of exactly two adjacent actors, a philosopher and a fork, as shown in the connection diagram of figure 1.

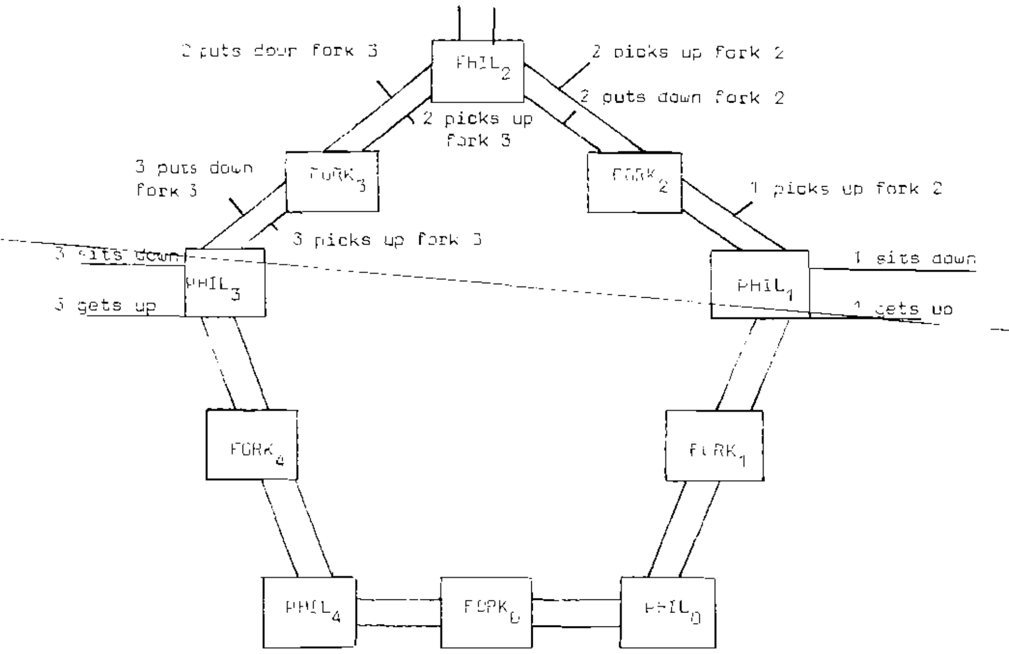


Figure 1

2.5.2 Behaviour

Apart from thinking and eating which we have chosen to ignore, the life of each philosopher is described:

$PHIL_i$ = (i sitsdown $\rightarrow$ i picksup fork i $\rightarrow$ i picks up fork (i $\oplus$ 1) $\rightarrow$ i puts down fork i $\rightarrow$ i puts down fork (i $\oplus$ 1) $\rightarrow$ i gets up $\rightarrow$ $PHIL_i$)

The rôle of a fork is a simple one; it is repeatedly picked up and put down by one of its adjacent philosophers:

$$\begin{aligned} FORK_i = (&i \text{ picks up fork } i \rightarrow i \text{ puts down fork } i \rightarrow FORK_i \\ &| (i \ominus 1) \text{ picks up fork } i \rightarrow (i \ominus 1) \text{ puts down fork } i \rightarrow FORK_i) \end{aligned}$$

The behaviour of the whole College is the concurrent combination of the behaviour of each of these components

$$\begin{aligned} PHILOS &= (PHIL_0 \| PHIL_1 \| PHIL_2 \| PHIL_3 \| PHIL_4) \\ FORKS &= (FORK_0 \| FORK_1 \| FORK_2 \| FORK_3 \| FORK_4) \\ COLLEGE &= (PHILOS \| FORKS) \end{aligned}$$

2.5.3 Deadlock!

When this mathematical model had been constructed, it revealed a serious danger. Suppose all the philosophers get hungry at about the same time; they all sit down; they all pick up their own forks; and they all reach out for the other fork - which isn't there. In this undignified situation, they will all assuredly starve. Although each actor is capable of further action, there is no action which any pair of them can agree to do next.

However, our story does not end so sadly. Once the danger was detected, there were suggested many ways to avert it. For example, one of the philosophers could always pick up the wrong fork first - if only they could have agreed which one it should be! The purchase of a single additional fork was ruled out for similar reasons, whereas the purchase of five more forks was much too expensive.

The solution finally adopted was the appointment of a footman, whose duty it was to assist each philosopher into and out of his chair. His alphabet was defined:

$$\alpha Footman = \bigcup_{i=0}^{4} \{i \text{ sits down}, i \text{ gets up}\}$$

This footman was given secret instructions never to allow more than four philosophers to be simultaneously seated. His behaviour is most simply defined by mutual recursion.

Let $U = \bigcup_{i=0}^{4} \{i \text{ gets up}\}$, $D = \bigcup_{i=0}^{4} \{i \text{ sits down}\}$

$FOOT_j$ defines the behaviour of the footman with j philosophers seated.

$$FOOT_0 = (x{:}D \longrightarrow FOOT_1)$$

$$FOOT_j = (x{:}D \longrightarrow FOOT_{j+1} \mid y{:}\,U \longrightarrow FOOT_{j-1}) \quad \text{for } j \in \{1, 2, 3\}$$

$$FOOT_4 = (y{:}\,U \longrightarrow FOOT_3).$$

$$NEWCOLLEGE = COLLEGE \parallel FOOT_0$$

The edifying tale of the dining philosophers is due to Edsger W. Dijkstra. The footman is due to Carel Scholten.

2.6 Change of symbol

It is frequently useful to describe a number of different processes, which behave in a very similar fashion, except that the names of the events in which they engage are different. Let f be a function which maps symbols of a process P onto symbols of the process Q; and suppose that whenever P is ready to perform some action c, Q is ready to perform the action f(c), and vice versa. In this case P is the inverse image under f of the process Q, and we write

$$P = f^{-1}(Q).$$

Clearly, this can only be true if

$$\alpha P = f^{-1}(\alpha Q)$$

where $f^{-1}(A) = \{x \mid f(x) \in A\}$

Examples

X1 After a few years, the price of everything goes up. To represent the effect of inflation on a vending machine, define:

$f(in2p) = in10$ $\qquad$ $f(small) = large$

$f(in1p) = in10$ $\qquad$ $f(very\ small) = small$

$f(out1p) = out\ p$

The new vending machine may be simply described

$$NEWVMC = f^{-1}(VMC)$$

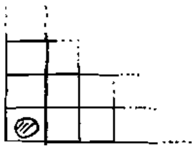
X2 - counter behaves like CT_0, except that it moves "right" and "left" instead of "up" and "down".

$f(\text{right}) = \text{up}$, $f(\text{left}) = \text{down}$, $f(\text{around}) = \text{around}$

$LR = f^{-1}(CT_0)$

The main reason for changing event names of processes in this fashion is to enable them to be composed usefully in concurrent combination.

X3 A counter moves left, right, up or down on an infinite board with margins at the left and at the bottom:



It starts at the bottom left corner. On this square alone, it can turn "around". As in 2.3 X2, vertical and horizontal movements can be modelled as independent actions of separate processes; but "around" requires simultaneous participation of both.

$LRUD = LR \| CT_0$

Change of symbol is particularly useful in constructing groups of similar processes which operate concurrently in providing a service to their common environment, but which ~~do not interact with each other in~~ any way at all. This means that they must all have different and mutually disjoint alphabets. To achieve this, each process is labelled by a different name; and each event of a labelled process is also labelled by its name. A labelled event is a pair $\langle l, x\rangle$, where l is a label, and x is the symbol standing for the event.

A process P labelled by l is denoted by

$$l:P$$

It engages in the event $\langle l,x\rangle$ whenever P would have engaged in x.

The function required to define $l:P$ is

$strip_l(\langle l,x\rangle) = x$

$strip_l(y)$ is undefined if y is not labelled with l.

$l:P = strip_l^{-1}(P)$.

X4 A pair of vending machines standing side by side

$$(\mathrm{left}:VMS) \| (\mathrm{right}:VMS)$$

The alphabets of the two processes are disjoint, and every event that occurs is labelled by the name of the machine on which it occurred. If the machines were not named before being placed in parallel, every event would require participation of both of them; and the pair would be indistinguishable from a single machine; this is a consequence of the fact that

$$(VMS \| VMS) = VMS$$

2.6.1 Laws

Change of symbol distributes in the obvious way through all ~~other~~ operators introduced so far.

L1 $f^{-1}(STOP_A) = STOP_{f^{-1}(A)}$

L2 $f^{-1}(x:A \longrightarrow P(x)) = (x:f^{-1}(A) \longrightarrow f^{-1}(P(f(x))))$

L3 $f^{-1}(P \| Q) = f^{-1}(P) \| f^{-1}(Q)$

L4 $f^{-1}(\mu X:A.\ F(X)) = \mu X:f^{-1}(A).\ f^{-1}(F(X))$

Compounded change of symbol behaves in the expected way.

L5 $g^{-1}(f^{-1}(P)) = (f \circ g)^{-1}(P)$

2.6.2 Implementation

In order to implement symbol change, we must be able to test whether a symbol is in the domain of the function. We therefore assume that the function will answer "BLEEP when presented with a symbol not in its domain.

```
inverse image (f,P) =
      λx. if f(x) = "BLEEP then "BLEEP
        else if P(f(x)) = "BLEEP then "BLEEP
                                  else inverse image (f,P(f(x)))
```

To implement the naming operator, we represent the labelled event $\langle l, x \rangle$ as the pair cons(l,x).

```
name(l,f) =
     λx. if null(x) ∨ atom(x) then "BLEEP
         else if car(x) ≠ l then "BLEEP
         else if f(cdr(x)) = "BLEEP then "BLEEP
                                else name(l, f(cdr(x)))
```

2.6.3 Traces

If x is an event in the life of $f^{-1}(P)$ then f(x) must be the corresponding event in the life of P. Conversely, if f(x) occurs in the life of P, x must be a possible corresponding event in the life of $f^{-1}(P)$. The same argument applies to sequences of consecutive events. If t is a trace of $f^{-1}(P)$ then $f^*(t)$ is the result of applying f to each individual event in t; the result must be a possible trace of P.

L1 $\quad \text{traces}\,(f^{-1}(P)) \;=\; \{t \mid f^*(t) \in \text{traces}(P)\}$

where f^* is defined in 1.9.1.

L2 $\quad f^{-1}(P)/s \;=\; f^{-1}(P/f^*(s)) \qquad$ provided $f^*(s) \in \text{traces}(P)$.

2.7 Specifications

Let P and Q be processes intended to run concurrently, and suppose we have proved

P sat S(tr)

and Q sat T(tr).

Let tr be a trace of $(P \| Q)$. It follows by 2.3.3. L1 that $(tr \upharpoonright \alpha P)$ is a trace of P, and consequently it satisfies S:

$S(tr \upharpoonright \alpha P)$

Similarly, $(tr \upharpoonright \alpha Q)$ is a trace of Q, so

$T(tr \upharpoonright \alpha Q)$

This holds for every trace of $(P \| Q)$.

Consequently we may deduce

$$(P \| Q) \ \underline{\text{sat}}\ (S(tr \upharpoonright \alpha P) \wedge T(tr \upharpoonright \alpha Q)).$$

This informal reasoning is summarised in the law

L1 If $P \ \underline{\text{sat}}\ S(tr)$

and $Q \ \underline{\text{sat}}\ T(tr)$

then $(P \| Q) \ \underline{\text{sat}}\ (S(tr \upharpoonright \alpha P) \wedge T(tr \upharpoonright \alpha Q))$

Example

X1 (as 2.3.1 X1)

Let $\alpha P = \{a,c\}$, $\alpha Q = \{b,c\}$

$P = (a \longrightarrow c \longrightarrow P)$

$Q = (c \longrightarrow b \longrightarrow P)$

The proof of 1.10.2 X1 can obviously be adapted to show that

$P \ \underline{\text{sat}}\ (0 \leqslant tr.a - tr.c \leqslant 1)$

and $Q \ \underline{\text{sat}}\ (0 \leqslant tr.c - tr.b \leqslant 1)$

By L1 it follows that

$$(P \| Q) \ \underline{\text{sat}}\ (0 \leqslant (tr \upharpoonright \alpha P).a - (tr \upharpoonright \alpha P).c \leqslant 1$$
$$\wedge\ 0 \leqslant (tr \upharpoonright \alpha Q).c - (tr \upharpoonright \alpha Q).b \leqslant 1$$
$$\Longrightarrow 0 \leqslant tr.a - tr.b \leqslant 2$$

since $(tr \upharpoonright A).a = tr.a$ wherever $a \in A$

Of course, this proof does not exclude the possibility that $(P \| Q)$ will stop as a result of deadlock, as illustrated in 2.2 X2 and 2.2.1 L4B. One way to eliminate the risk of stoppage is to prove that a process defined by the parallel combinator is equivalent to a non-stopping process defined without this combinator, as was done in 2.3.1 X1. However such proofs involve long and tedious algebraic transformations. Wherever possible, one should appeal to some general law, such as

L2 If P and Q never stop and if $(\alpha P \cap \alpha Q)$ contains at most one event then $(P \| Q)$ never stops.

X2 The process $(P \| Q)$ defined in X1 will never stop, because $\alpha P \cap \alpha Q = \{c\}$.

The proof rule for change of symbol is fairly obvious.

L2 If P sat S(tr)

then $f^{-1}(P)$ sat $S(f^{*}(tr))$

CHAPTER THREE

NON-DETERMINISM

3.1 Introduction

The choice operator $(x:B \longrightarrow P(x))$ is used to define a process which exhibits a range of possible behaviours; and the concurrency operator $\|$ permits some other process to make a selection between the alternatives offered. For example, the change-giving machine CH5C (1.1.3,X7) offers its customer the choice of taking his change as three small coins and one large or two large coins and one small. Sometimes a process has a range of possible behaviours, but the environment of the process does not have any ability to influence the selection between the alternatives. For example, a different change-giving machine may give change in either of the combinations described above; but the choice between them cannot be controlled or even predicted by its user. The choice is made, as it were "internally", by the machine itself, in a non-deterministic fashion. There is nothing mysterious about this kind of non-determinism: it arises from a deliberate decision to ignore other factors which influence the selection. For example, the combination of change given by the machine may depend on the way in which the machine has been loaded with large and small coins; but we have excluded these events from the alphabet.

3.2 Nondeterministic or

If P and Q are processes, then we introduce the notation

$$P \sqcap Q \qquad\qquad (P \text{ or } Q)$$

to denote a process which behaves either like P or like Q, where the selection between them is made arbitrarily, without the knowledge or control of the external environment. The alphabets of the operands are assumed to be the same:

$$\alpha(P \sqcap Q) \;=\; \alpha P = \alpha Q$$

Examples

X1 A change-giving machine which always gives the right change in one of two combinations on each occasion

$$CH5D = (in5p \rightarrow ((out1p \rightarrow out1p \rightarrow out1p \rightarrow out2p \rightarrow CH5D)$$
$$\sqcap (out2p \rightarrow out1p \rightarrow out2p \rightarrow CH5D))$$

X2 CH5D may give a different combination of change on each occasion of use. Here is a machine that always gives the same combination, but we do not know initially which it will be (see 1.1.2.X3,X4)

$$CH5E = CH5A \sqcap CH5B$$

Of course, after this machine gives its first coin in change, its subsequent behaviour is entirely predictable. For this reason,

$$CH5D \neq CH5E$$

3.2.1 Laws

The algebraic laws governing non-deterministic choice are exceptionally simple and obvious. A choice between P and P is vacuous;

L1 $P \sqcap P = P$ idempotence

It does not matter in which order the choice is presented:

L2 $P \sqcap Q = Q \sqcap P$ symmetry

A choice between three alternatives can be split into two binary choices. It does not matter in which way this is done:

L3 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ associativity

The occasion on which a non-deterministic choice is made is not significant. A process which first does x and then makes a choice is indistinguishable from one which first makes the choice and then does x.

L4 $x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q)$ distribution

The law L4 states that the prefixing operator distributes through non-determinism. Such operators are said to be distributive. A dyadic operator is said to be distributive if it distributes through $\sqcap$ in both its argument positions independently. All the operators defined so far for processes are distributive in this sense:

L5 $(x{:}A \rightarrow (P(x) \sqcap Q(x)) = (x{:}A \rightarrow P(x)) \sqcap (x{:}A \rightarrow Q(x))$

L6 $P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R)$

L7 $(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$

L8 $f^{-1}(P \sqcap Q) = f^{-1}(P) \sqcap f^{-1}(Q)$

However, the recursion operator is <u>not</u> distributive, except in the trivial case when the operands of $\sqcap$ are identical. This point is well illustrated by the difference between the two processes

$$P = \mu X.((a \longrightarrow X) \sqcap (b \longrightarrow X))$$

$$Q = (\mu X.(a \longrightarrow X)) \sqcap (\mu X.(b \longrightarrow X)))$$

P can make an independent choice between "a" and "b" on each iteration, so ~~its traces~~ include

$$\langle a,b,b,a,b \rangle$$

Q must make a choice between always doing "a" and always doing "b", and so this trace cannot be a trace of Q.

In view of laws L1 to L3 it is useful to introduce a multiple choice operator. Let A be a finite nonempty set:

$$A = \{a, b, \ldots, z\}$$

Then we define

$$\prod_{x:A} P(x) = P(a) \sqcap P(b) \sqcap \ldots \sqcap P(z)$$

3.2.2 Implementation

There are several different permitted implementations of $(P \sqcap Q)$. In fact one of the main reasons for introducing non-determinism is to permit a range of possible implementations, from which a cheap or efficient one can be selected. A very efficient implementation is to make an arbitrary choice between the operands. For example, one might choose

$$or1(P,Q) = P$$

or one might choose

$$or2(P,Q) = Q$$

If the event that happens first is possible for both P and Q, the decision may be postponed to some later occasion. The "kindest" (but least efficient) implementation is one that continues to entertain both alternatives until the environment chooses between them:

```
or3(P,Q)  =  λx.if P(x) = "BLEEP then Q(x)
                 else if Q(x) = "BLEEP then P(x)
                                        else or3(P(x), Q(x))
```

The implementation "or3" is the only one that obeys the law of symmetry L2. Both the other implementations are asymmetric in P and Q. This does not matter. The laws should be taken to assert the identities of the processes, not of any particular implementation of them.

If desired, the laws may be regarded as asserting the identity of the set of all permitted implementations of their left and right hand sides. For example, if $\{or1, or2, or3\}$ are all permitted implementations of $\sqcap$, the law of symmetry states:

$$\{or1(P,Q),\ or2(P,Q),\ or3(P,Q)\}$$
$$= \{or1(Q,P),\ or2(Q,P),\ or3(Q,P)\}$$

One of the advantages of introducing non-determinism is to avoid loss of symmetry that would result from selecting one of the two efficient implementations, without incurring the inefficiency of the symmetric implementation.

3.2.3 Traces

If t is a trace of P, then t is also a possible trace of $(P \sqcap Q)$, i.e., in the case that P is selected. Similerly if t is a trace of Q, it is also a trace of $(P \sqcap Q)$. Conversely, each trace of $(P \sqcap Q)$ must be a trace of one or both alternatives.

L1 $\quad$ traces $(P \sqcap Q)$ = traces $(P) \cup$ traces (Q)

L2 $\quad (P \sqcap Q)/s = Q/s \qquad$ if $s \tilde{\in}$ traces (P)

$\qquad\qquad\quad = P/s \qquad$ if $s \tilde{\in}$ traces (Q)

$\qquad\qquad\quad = (P/s) \sqcap (Q/s) \qquad$ otherwise

3.3 General Choice

The environment of $(P \sqcap Q)$ has no control or even knowledge of the choice that is made between P and Q, or even the time at which the choice is made. So $(P \sqcap Q)$ is not a helpful way of combining processes, because the environment must be prepared to deal with either P or Q; and either one of them separately would have been easier to deal with. We therefore introduce another operation $(P \, [\,] \, Q)$, for which the environment can control which of P and Q will be selected, provided that this control is exercised on the very first action. If this action is not a possible first action of P, then Q will be selected; but if Q cannot engage in the action, P will be selected. If however the first action is possible for both P and Q, then the choice between them is non-deterministic. (Of course, if the event is impossible for both P and Q, then it just can't happen.) As usual

$$\alpha(P \, [\,] \, Q) = \alpha P = \alpha Q$$

In the case that no initial event of P is also possible for Q, the general choice operator is ~~the same as~~ the $|$ operator, which has been used hitherto to represent choice between different events:

$$(c \longrightarrow P \, [\,] \, d \longrightarrow Q) = (c \longrightarrow P \mid d \longrightarrow Q) \qquad \text{if } c \neq d.$$

However, if the initial events are the same, $(P \, [\,] \, Q)$ degenerates to non-deterministic choice:

$$(c \longrightarrow P) \, [\,] \, (c \longrightarrow Q) = (c \longrightarrow P) \sqcap (c \longrightarrow Q)$$

3.3.1 Laws

The algebraic laws for $[\,]$ are similar to those for $\sqcap$, and for the same reasons.

L1–L3 $[\,]$ is idempotent, symmetric, and associative.

L4 $P \, [\,] \, STOP = P$ unit

The following law encapsulates the informal definition of the operation.

L5 $(x:A \longrightarrow F(x)) \, [\,] \, (y:B \longrightarrow G(y)) =$

$(z:(A \cup B) \longrightarrow$ (if $z \in (A - B)$ then $F(z)$

else if $z \in (B - A)$ then $G(z)$

else if $z \in (A \cap B)$ then $(F(z) \sqcap G(z))))$

Like all other operators introduced so far, $[\,]$ distributes through $\sqcap$

L6 $P \, [\,] \, (Q \sqcap R) = (P \, [\,] \, Q) \sqcap (P \, [\,] \, R).$

What may seem more surprising is that $\sqcap$ distributes through $\square$

L7 $\quad P \sqcap (Q \square R) \;=\; (P \sqcap Q) \square (P \sqcap R)$

This law states that the choices involved in $\sqcap$ and $\square$ are independent. The left hand side describes a non-deterministic choice, followed (in one case) by an external choice between Q and R. The right hand side describes an external choice followed by a non-deterministic choice between the selected alternative and P. The law states that the set of possible results of these two choice strategies are the same.

3.3.2 Implementation

The implementation of the choice operator follows closely the law L5. Thanks to the symmetry of "or", it is also symmetrical.

```
choice (P,Q) = λx. if P(x) = "BLEEP then Q(x)
                   else if Q(x) = "BLEEP then P(x)
                            else  or (P(x), Q(x))
```

3.3.3 Traces

Every trace of $(P \square Q)$ must be a trace of P or a trace of Q, and conversely.

$$\text{traces } (P \square Q) \;=\; \text{traces } (P) \cup \text{traces } (Q)$$

3.4 Refusals

The distinction between $(P \sqcap Q)$ and $(P \square Q)$ is quite subtle. They cannot be distinguished by their traces, because each trace of one of them is also a possible trace of the other. However it is possible to put them in an environment in which $(P \sqcap Q)$ can deadlock, on its first step, but $(P \square Q)$ cannot. For example let $x \neq y$ and

$$P = x \longrightarrow P \quad , \quad Q = y \longrightarrow Q$$

(1) $\quad (P \square Q) \parallel P \;=\; (x \longrightarrow P) \;=\; P$

(2) $\quad (P \sqcap Q) \parallel P \;=\; (P \parallel P) \sqcap (Q \parallel P)$

$\qquad\qquad\qquad\qquad = \; P \sqcap STOP$

This shows that in environment P, $(P \sqcap Q)$ may reach deadlock, but that $(P \square Q)$ cannot. If deadlock occurs, then at least we know that it wasn't $(P \mathbin{[]} Q)$. Of course, even with $(P \sqcap Q)$ we can't be sure that deadlock will occur; and if it doesn't occur, we will never know that it might have. But the mere possibility of an occurrence of deadlock is enough to distinguish the two processes.

In general, let X be a set of events which are offered initially by the environment of a process P. If it is possible for P to deadlock on its first step when placed in this environment, we say that X is a *refusal* of P. The set of all such refusals of P is denoted

refusals (P).

3.4.1 Laws

The following laws define the refusals of various simple processes. ~~The process STOP is already deadlocked, and refuses everything.~~

~~L1 refusals $(STOP_A)$ = all subsets of A~~

A process $(c \longrightarrow P)$ refuses every set that does not contain the event c:

L2 $\quad$ refusals $(c \longrightarrow P) \;=\; \{X \mid c \notin X\}$

These two laws generalise to

L3 $\quad$ refusals $(x{:}A \longrightarrow P(x)) \;=\; \{X \mid X \cap A = \{\}\}$

If P can refuse X, so will $(P \sqcap Q)$, in the case that P is selected. Similarly every refusal of Q is also a possible refusal of $(P \sqcap Q)$. These are its only refusals, so

L4 $\quad$ refusals $(P \sqcap Q) \;=\;$ refusals $(P) \cup$ refusals (Q).

A converse argument applies to $(P \mathbin{[]} Q)$. If X is *not* a refusal of P, then P *can't* refuse X, and neither can $(P \mathbin{[]} Q)$. Similarly if X is not a refusal of Q, then it is not a refusal of $(P \mathbin{[]} Q)$. However if *both* P and Q can refuse X, so can $(P \mathbin{[]} Q)$.

L5 $\quad$ refusals $(P \mathbin{[]} Q) \;=\;$ refusals $(P) \cap$ refusals (Q)

Comparison of L5 with L4 shows most clearly the distinction between $[]$ and $\sqcap$.

If P can refuse X and Q can refuse Y, then their combination $(P \parallel Q)$ can refuse all events refused by P as well as all events refused by Q, i.e., it can refuse the union of the two sets X and Y.

L6 refusals $(P \parallel Q) = \{X \cup Y \mid X \in$ refusals (P) & $Y \in$ refusals $(Q)\}$

For symbol change, the relevant law is clear

L7 refusals $(f^{-1}(P)) = \{X \mid f(X) \in$ refusals $(P)\}$

where $f(X) = \{f(x) \mid x \in X\}$.

There are a number of general laws about refusals. A process can refuse only events in its own alphabet. A process deadlocks when the environment offers no events; and if a process refuses a nonempty set, it can also refuse any subset of that set. Finally any event x which can't occur initially may be added to any set X already refused.

L8 $X \in$ refusals $(P) \implies X \subseteq \alpha P$

L9 $\{\} \in$ refusals (P)

L10 $(X \cup Y) \in$ refusals $(P) \implies X \in$ refusals (P).

L11 $X \in$ refusals $(P) \implies (X \cup \{x\}) \in$ refusals (P)
$\vee \langle x \rangle \in$ traces (P)

3.5 Concealment

In general, the alphabet of a process contains just those events which are considered to be relevant, and whose occurrence requires simultaneous participation of its environment. We therefore often wish to regard certain events as internal transitions of the process, which occur automatically as soon as they can, without being observed or controlled by the environment of the process. If C is a finite set of events to be concealed in this way, then

$$P \setminus C$$

is a process which behaves like P, except that the occurrence of all events in C is concealed. Clearly it is our intention that:

$$\alpha(P \setminus C) = (\alpha P) - C$$

Examples

X1 A noisy vending machine (2.3.X1) can be placed in a soundproof box:

$$NOISYVM \setminus \{clink, clunk\}$$

Its unexercised capability of dispensing toffee can also be removed from its alphabet, without affecting its actual behaviour. The resulting process is equal to the simple vending machine

$$VMS = NOISYVM \setminus \{clink, clunk, toffee\}$$

When two processes have been combined to run concurrently, their mutual interactions are usually regarded as internal workings of the resulting system; they are intended to occur autonomously and as quickly as possible without the knowledge or intervention of the system's outer environment. ~~Thus it is~~ the symbols in the intersection of the alphabets of the two components that need to be ~~concealed.~~

X2 (See 2.3.1.X1)

Let $\alpha P = \{a, c\}$, $\alpha Q = \{b, c\}$

$$P = (a \longrightarrow c \longrightarrow P), \quad Q = (c \longrightarrow b \longrightarrow Q)$$

$$(P \,||\, Q) \setminus \{c\} = a \longrightarrow \mu X.(a \longrightarrow b \longrightarrow X \\ \quad [] \; b \longrightarrow a \longrightarrow X)$$

3.5.1 Laws

The first laws state that concealing no symbols has no effect, and that it makes no difference in what order the symbols of a set are concealed. The remaining laws in this group show how concealment distributes through other operators.

L1 $P \setminus \{\,\} = P$

L2 $(P \setminus B) \setminus C = P \setminus (B \cup C)$

L3 $\setminus C$ distributes through $\sqcap$

L4 $STOP_A \setminus C = STOP_{A-C}$

L5 $(x \rightarrow P) \setminus C = x \rightarrow (P \setminus C)$ if $x \tilde{\in} C$
$= P \setminus C$ if $x \in C$

L6 If $\alpha P \cap \alpha Q \cap C = \{\,\}$ then
$(P \,\|\, Q) \setminus C = (P \setminus C) \,\|\, (Q \setminus C)$

L7 $f^{-1}(P \setminus C) = f^{-1}(P) \setminus f^{-1}(C)$

Note that $\setminus C$ does <u>not</u> distribute through $\Box$.

If none of the possible initial events of a choice is concealed, then the initial choice remains the same:

L8 If $A \cap C = \{\,\}$
then $(x{:}A \rightarrow P(x)) \setminus C = (x{:}A \rightarrow (P(x) \setminus C))$

Like the choice operator $\Box$, the concealment of events can introduce non-determinism. When several different concealed events can happen, it is not determined which of them will occur; but whichever does occur is concealed.

L9 If $A \subseteq C$, and A is not empty, then

$$(x{:}A \longrightarrow P(x)) \setminus C \;=\; \prod_{x:A} (P(x) \setminus C)$$

In the intermediate case, when some of the initial events are concealed and some are not, the situation is rather more complicated. Consider the process

$$(c \longrightarrow P \mid d \longrightarrow Q) \setminus C \qquad \text{where } c \in C,\ d \notin C$$

The concealed event c may happen immediately. In this case the total behaviour will be defined by $(P \setminus C)$, and the possibility of occurrence of the event d will be withdrawn. But we cannot reliably assume that d won't happen. If the environment is ready for it, d may very well happen before the hidden event, after which the hidden event c can no longer occur. But even if d occurs, it might have been performed by $(P \setminus C)$ after occurrence of c. In this case, the total behaviour is as defined by

$$(P \setminus C) \;\square\; (d \longrightarrow (Q \setminus C))$$

The choice between this and $(P \setminus C)$ is non-deterministic.

This is a rather convoluted justification for the rather complex law:

$$(c \longrightarrow P \mid d \longrightarrow Q) \setminus C \;=\; (P \setminus C) \sqcap ((P \setminus C) \;\square\; (d \longrightarrow (Q \setminus C)))$$

Similar reasoning justifies the more general law

L10 If $C \cap A \neq \{\}$ then

$$(x{:}A \longrightarrow P(x)) \setminus C \;=\; Q \sqcap (Q \;\square\; (x{:}(A-C) \longrightarrow P(x)))$$

$$\text{where} \qquad Q \;=\; \prod_{x:A \cap C} P(x) \setminus C$$

3.5.2 Implementation

For simplicity, we shall implement an operation which hides a single symbol at a time

$$\text{hide}(P,c) \;=\; P \setminus \{c\}$$

To hide a set of two or more symbols, they may be hidden one after the other, since

$$P \setminus \{c1, c2, \ldots, cn\} \;=\; (\ldots((P \setminus \{c1\}) \setminus \{c2\}) \setminus \ldots) \setminus \{cn\}$$

The most efficient implementation is one that always makes the hidden event occur as soon as it can.

```
hide (P,c)  =  if P(c) = "BLEEP then
                        λx. if P(x) = "BLEEP then "BLEEP
                                  else  hide (P(x),c)
                  else  hide (P(c),c).
```

This implementation fails to work when its parameter P can begin with an arbitrarily long sequence of the events "c". In this case, the hide operation will degenerate to an infinite recursion, since it will always choose the second main branch ($P(c) \neq$ "BLEEP), and try to compute:

hide(P,c), hide(P(c),c), hide((P(c))(c),c), ...

This phenomenon is known as <u>divergence</u>; and we will regard this as an error in the process $P \setminus \{c\}$ rather than an error in the implementation of concealment.

3.5.3 Traces

If t is a trace of P, the corresponding trace of $P \setminus C$ is obtained from t simply by removing all occurrences of any of the symbols in C. Conversely each trace of $P \setminus C$ must have been obtained from some such trace of P. We therefore state

L1 $\text{traces}(P \setminus C) = \{t \mid t \upharpoonright \bar{C} \in \text{traces}(P)\}$

provided that $P \setminus C$ does not diverge.

The possible divergence of $P \setminus C$ can be defined in terms of the traces of P:

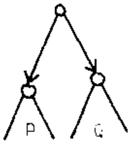
diverges (P,C) =

$$\exists t.\ \{s \mid s \in \text{traces}(P)\ \&\ s \upharpoonright \bar{C} = t\} \text{ is infinite.}$$

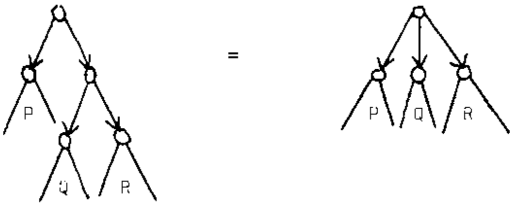
3.5.4 Pictures

Non-deterministic choice can be represented in a picture by a node from which emerge two or more unlabelled arrows; on reaching the node, a process passes imperceptibly along one of the emergent arrows:

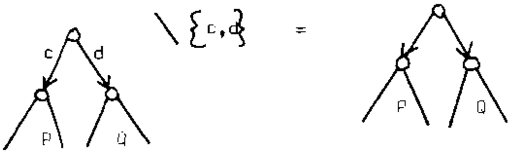
$P \sqcap Q$ is pictured as



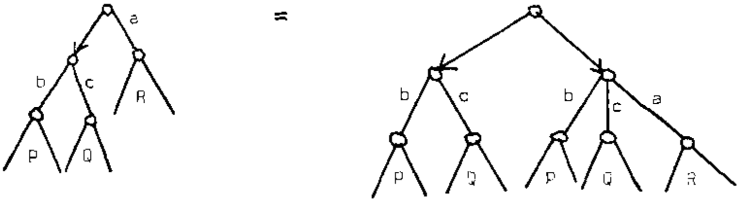
The algebraic laws governing non-determinism assert identities between such pictures, e.g. by associativity



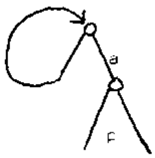
Concealment of symbols may be regarded as ~~an operation~~ which simply removes concealed symbols from all arcs which they label, so that these arcs turn into unlabelled arrows. The resulting non-determinism emerges naturally, as shown below.



But what is the meaning of a node if some of its arcs are labelled and some are not? The answer is given by the law 3.5.1 L10. Such a node can be eliminated by redrawing as shown below



Such eliminations are always possible for finite trees. They are also possible for infinite graphs, provided that the graph contains no infinite path of consecutive unlabelled arrows, as for example:



Such a picture can arise only as a result of divergence, which we have already decided to regard as an error.

3.5.5 Warning

Unfortunately the introduction of hiding permits the construction of recursion equations which do not have an unique solution.

For example let

$$\alpha X = \{a,b\}$$
$$X = a \longrightarrow ((X \setminus \{a\}) \| STOP_{\{a\}})$$

This equation looks as though it is guarded, but it has many solutions. If P is any process whatsoever, then a possible solution is

$$a \longrightarrow ((P \setminus \{a\}) \| STOP_{\{a\}})$$

Proof. Substitute this formula for X in the RHS of the recursive equation.

$$a \longrightarrow (((a \longrightarrow (P \setminus \{a\} \| STOP_{\{a\}})) \setminus \{a\}) \| STOP_{\{a\}})$$
$$= a \longrightarrow (((P \setminus \{a\} \| STOP_{\{a\}}) \setminus \{a\}) \| STOP_{\{a\}})$$
$$= a \longrightarrow (P \setminus \{a\} \setminus \{a\}) \| (STOP_{\{a\}} \setminus \{a\})) \| STOP_{\{a\}}$$
$$= a \longrightarrow ((P \setminus \{a\}) \| STOP_{\{\}}) \| STOP_{\{a\}}$$
$$= a \longrightarrow ((P \setminus \{a\}) \| STOP_{\{a\}})$$

$P = STOP$ and $P = (b \rightarrow P)$ give two different solutions.

In order to preserve the validity of the law of unique solutions (1.3.L2) any recursion which involves hiding must be regarded as <u>not guarded</u>, and possibly even meaningless. however, there is an easy method to determine whether this is so. Introduce a new event Υ into the alphabet of the process, and prefix it to the right hand side of each recursive equation. The equations are now guarded, and therefore have an unique solution. In this solution, each occurrence of the event Υ represents a recursive call of the process. Such events are clearly internal to the operation of the process, and should be concealed from its external environment. If this concealment is well-defined, the result is exactly what we want - an unique solution to the original recursive equations. If the concealment is ill defined, then there is a possibility that the process will recurse infinitely, without ever engaging in any more externally visible actions. Such recursions are also with justice declared to be ill-defined.

X4 The obviously disastrous recursion

$$\mu X:A.X \;=\; (\mu X:A.\Upsilon \longrightarrow X)\setminus\{\Upsilon\}$$

But traces $(\mu X:A.\ \Upsilon \longrightarrow X) \;=\; \{\Upsilon\}^*$,

and $\{\Upsilon\}^*\setminus\{\Upsilon\}$ is certainly ill definec.

3.6 Interleaving

The $\|$ operator was defined in chapter 2 in such a way that actions in the alphabet of both operends require simulteneous participation of them both, whereas the remaining actions of the system occur in an arbitrary interleaving. Using this operator, it is possible to combine interacting processes into systems with a possibility of concurrent activity, but without introducing non-determinism.

However, it is sometimes useful to join processes to operate concurrently without directly interacting or synchronising with each other. In this cese, each action of the system is an action of exactly one of the processes. If one of the processee cannot engage in the action, then it must have been the other one; but if both processes can engage in the action, the choice between them is non-deterministic. This form of combination is denoted

$$P \;|||\; Q \qquad\qquad (P \text{ interleave } Q)$$

and its alphabet is defined by the usual stipulation

$$\alpha(P\;|||\;Q) \;=\; \alpha P \;=\; \alpha Q.$$

Examples

X1 A vending machine that will accept up to two coins before dispensing up to two chocs:

$$VMS\;|||\;VMS$$

X2 A footmar mede from four lackeys, each serving only one philosopher at a time (see 2.5.3)

$$L \;|||\; L \;|||\; L \;|||\; L$$

where $L = (x{:}D \longrightarrow (y{:}\,U \longrightarrow L))$

3.6.1 Laws

L1-3 $|||$ is associetive, symmetric, ano oistributive.

L4 $P \;|||\; STOP = P$

L5 $P \;|||\; RUN = RUN$

L6 $(x \longrightarrow P) \;|||\; (y \longrightarrow Q) = (x \longrightarrow (P \;|||\; (y \longrightarrow Q))$
$\qquad\qquad\qquad\qquad\qquad\qquad \square\; y \longrightarrow ((x \longrightarrow P) \;|||\; Q))$

~~L7 If P = (x:A → P(x))~~

and $Q = (y{:}B \longrightarrow P(y))$

then $P \;|||\; Q = (x{:}A \longrightarrow (P(x) \;|||\; Q)$
$\qquad\qquad \square\; y{:}B \longrightarrow (P \;|||\; Q(x)))$

Note: $|||$ does not distribute through $\square$

Example

X1 Let $R = (a \longrightarrow b \longrightarrow R)$

$(R \;|||\; R) = (e \longrightarrow ((b \longrightarrow R) \;|||\; R)$ L6
$\qquad \square\; a \longrightarrow (R \;|||\; (b \longrightarrow R)))$
$\qquad = a \longrightarrow ((b \longrightarrow R) \;|||\; R) \sqcap (R \;|||\; (b \longrightarrow R))$
$\qquad = a \longrightarrow ((b \longrightarrow R) \;|||\; R)$ L2

$(b \longrightarrow R) \;|||\; R = (a \longrightarrow ((b \longrightarrow R) \;|||\; (b \longrightarrow R))$ L6
$\qquad \square\; b \longrightarrow (R \;|||\; R))$
$\qquad = (a \longrightarrow (b \longrightarrow ((b \longrightarrow R) \;|||\; R))$
$\qquad \square\; b \longrightarrow (a \longrightarrow ((b \longrightarrow R) \;|||\; R)))$ } as shown above
$\qquad = \mu X.(a \longrightarrow b \longrightarrow X$
$\qquad\qquad \square\; b \longrightarrow a \longrightarrow X)$

since the recursion is guarded.

Thus $(R \mid\mid\mid R)$ is identical to the example 3.5 X2.

3.6.2 Traces

A trace of $(P \mid\mid\mid Q)$ is an arbitrary interleaving of a trace from P with a trace from Q. For a definition of interleaving, see 1.9.3.

$$\text{traces}(P \mid\mid\mid Q) = \{s \mid \exists t \in \text{traces}(P).\exists u \in \text{traces}(Q).s \text{ interleaves } (t,u)\}$$

$(P \mid\mid\mid Q)$ can engage in any initial action possible for either P or Q; and it can therefore refuse only those sets which are refused by both P and Q.

$$\text{refusals } (P \mid\mid\mid Q) = \text{refusals } (P \mathbin{[]} Q).$$

The behaviour of $(P \mid\mid\mid Q)$ after engaging in the events of the trace s is defined by the rather elaborate formula

$$(P \mid\mid\mid Q)/s = \bigsqcap_{(t,u)\in T} (P/t)\mid\mid\mid(Q/u)$$

$$\text{where } T = \{(t,u) \mid t \in \text{traces}(P) \wedge u \in \text{traces}(Q) \wedge s \text{ interleaves } (t,u)\}$$

This law reflects the fact that there is no way of knowing in which way a trace s of $(P \mid\mid\mid Q)$ has been constructed as an interleaving of a trace from P and a trace from Q; thus after s, the future behaviour of $(P \mid\mid\mid Q)$ may reflect any one of the possible interleavings. The choice between them is ~~not known~~ and not determined.

3.7 Specifications

In 3.4 we have seen the need to introduce refusal sets as one of the important properties ~~of a process~~. In specifying a process, we therefore need to describe the desired properties of its refusal sets as well as its traces. Let us use the variable "ref" to denote the refusal set of a process.

X1 When a vending machine has ingested more coins than it has dispensed chocolates, the customer specifies that it must not refuse to dispense a chocolate

$$\text{FAIR} = (\text{tr.choc} < \text{tr.coin} \Rightarrow \text{choc} \tilde{\in} \text{ref})$$

X2 When a vending machine has given out as many chocolates as have been paid for, the owner specifies that it must not refuse a further coin

$$PROFIT1 = (tr.choc = tr.coin \Longrightarrow coin \ \tilde{\in}\ ref)$$

X3 A simple vending machine should satisfy the combined specification

$$NEWVMSPEC = FAIR \wedge PROFIT1 \wedge tr.choc \leq tr.coin$$

This specification is satisfied by VMS. It is also satisfied by a vending machine which will accept several coins in a row, and then give out several chocolates.

X4 If desired, one may place a limit on the balance of coins which may be accepted in a row.

$$ATMOST2 = (tr.coin - tr.choc \leq 2)$$

X5 If desired, one can insist that the machine accept at least two coins in a row:

$$ATLEAST2 = tr.coin - tr.choc < 2 \Longrightarrow coin \ \tilde{\in}\ ref$$

X6 The process STOP refuses every event in its alphabet. The following predicate specifies that a process with alphabet A will never stop:

$$NONSTOP = (ref \neq A)$$

Since

$$NEWVMSPEC \Longrightarrow ref \neq \{coin, choc\}$$

it follows that any process which satisfies NEWVMSPEC will never stop.

These examples show how the introduction of "ref" into the specification of a process permits the expression of a number of subtle but important properties; perhaps the most important of all is the property that the process must not stop (X6). These advantages are obtained at the cost of slightly increased complexity in proof rules and in proofs.

3.7.1 Proofs

By the definition of nondeterminism, $(P \sqcap Q)$ behaves either like P or like Q. Therefore every observation of its behaviour must be an observation possible for P or for Q or for both. This observation must therefore be described by the specification of P or by the specification of Q or by both. Consequently, the proof rule for non-determinism has an exceptionally simple form.

L1 If P sat S

and Q sat T

then $(P \sqcap Q)$ sat $(S \vee T)$

The proof rule for STOP is the same as given in 1.10.2 L1A.

L2A STOP sat $tr = \langle\rangle$

There is no need for explicit mention of a refusal set, since STOP may refuse any set whatsoever. Strictly, the rule should mention the alphabet A

$$STOP_A \text{ sat } (tr = \langle\rangle \wedge ref \subseteq A)$$

but in future we shall not be so strict.

The previous law for prefixing (1.10.2 L1B) is also still valid, but it is not quite strong enough to prove properties of ref. The rule must be strengthened by mention of the fact that in the initial state the initial action cannot be refused:

L2B If P sat $S(tr)$

then $(c \rightarrow P)$ sat $(tr = \langle\rangle \wedge c \notin ref$

$\vee\ tr_0 = c \wedge S(tr'))$

The law for general choice (1.10.2 L4) needs to be similarly strengthened.

L2 If $\forall x \in A.\ P(x)$ sat $S(tr,x)$

then $(x{:}A \longrightarrow P(x))$ sat $(tr = \langle\rangle \wedge (A \cap ref) = \{\}$

$\vee\ tr_0 \in A \wedge S(tr',tr_0))$

The law for parallel composition given in 2.7 L1 is still valid, provided that the specifications make no mention of refusal sets. In order to deal correctly with refusals, a slightly more complicated law is required

L3 If P sat $S(tr, ref)$

and Q sat $T(tr, ref)$

then $(P \parallel Q)$ sat $(\exists X, Y.\ ref = X \cup Y$

$\wedge S(tr \upharpoonright \alpha P, X) \wedge T(tr \upharpoonright \alpha Q, Y))$

The law for change of symbol needs a similar adaptation:

L4 If P sat $S(tr, ref)$

then $f^{-1}(P)$ sat $S(f^*(tr), f(ref))$

The law for $[]$ is surprisingly simple

L5 If P sat S

and Q sat T

then $(P [] Q)$ sat (if $tr = \langle\rangle$ then $(S \wedge T)$ else $(S \vee T))$

Initially, when $tr = \langle\rangle$, a set is refused by $(P [] Q)$ only if it is refused by both P and Q. This set must therefore be described by both their specifications. Subsequently, when $tr \neq \langle\rangle$, each observation of $(P [] Q)$ must be an observation either of P or of Q, and must therefore be described by one of their specifications (or both).

The law for interleaving does not need to mention refusal sets.

L6 If P sat $S(tr)$

and Q sat $T(tr)$

then $(P \mid\mid\mid Q)$ sat $(\exists s, t.\ tr$ interleaves (s, t)

$\wedge S(s) \wedge T(t))$

The law for concealment is complicated by the need to guard against divergence

L7 If P sat $S(tr, ref)$

then $(P \setminus C)$ sat (NODIV $\Longrightarrow$

$\exists s.\ tr = s \upharpoonright \overline{C}$

$\wedge S(s, ref \cup C))$

where NODIV states that divergence could not have taken place at any time during the past history of the process:

$$\text{NODIV} = (\forall s.\ s \leqslant tr \Longrightarrow \{t \mid S(t,\{\}) \wedge t \restriction \overline{C} = s\} \text{ is finite})$$

CHAPTER FOUR

COMMUNICATING PROCESSES

4.1 Introduction

In previous chapters we have introduced and illustrated a general concept of an event as an action without duration, whose occurrence may require simultaneous participation by more than one independently described process. In this chapter we shall concentrate on a special class of event known as a communication. A communication is an event that is described by a pair

$$<c,v>$$

where c is the name of the channel on which the communication takes place and v is the value of the message which passes.

The set of all communications that can take place on channel c of process P is defined:

$$\alpha c(P) = \{<c,v> \mid <c,v> \in \alpha P\}$$

The set of messages which can pass along channel c of process P is

$$\alpha c'(P) = \{v \mid <c,v> \in \alpha P\}$$

All the operations introduced in this chapter can be defined in terms of the more primitive concepts introduced in earlier chapters, and most of the laws are just special cases of familiar laws. The reason for introducing special notations is that they are suggestive of useful applications and implementation methods; and because in some cases imposition of notational restrictions permits the use of more powerful reasoning methods.

4.2. Input and Output

Let v be a member of $\alpha c'(P)$. A process which first outputs v on the channel c and then behaves like P is defined:

$$(c!v \longrightarrow P) = (<c,v> \longrightarrow P).$$

The only event in which this is initially prepared to engage is the communication event $<c,v>$.

A process which is initially prepared to input any value x which can be communicated on the channel c, and then behave like P(x) is defined

$$(c?x \longrightarrow P(x)) = (y: \alpha c(P) \longrightarrow P(y'))$$

where y' is the message part of the communication y

i.e. $y = \langle c, y' \rangle$ for all y in $\alpha c(P)$.

We shall observe the convention that channels are used for communication in only one direction and between only two processes. A channel which is used only for output by a process will be called an output channel of that process; and one used only for input will be called an input channel. In both cases, we shall say loosely that the channel name is a member of the alphabet of the process, i.e.,

$$c \in \alpha P \quad \text{means} \quad \alpha c(P) \subseteq \alpha P$$

One useful operation which deserves special mention is that which changes the name of a channel. The process $(d \xrightarrow[c]{} P)$ is defined as one which behaves exactly like P, except that the channel c has been renamed to d, where d is not in the alphabet of P.

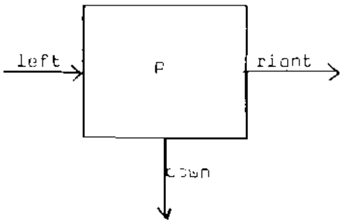
$$d \xrightarrow[c]{} P = f^{-1}(P)$$

where $f(\langle d,v \rangle) = f(\langle c,v \rangle)$ for $\langle c,v \rangle \in \alpha c(P)$

$f(\langle b,v \rangle) = \langle b,v \rangle$ for $b \neq c,\ b \in \alpha P$.

$$\alpha(d \xrightarrow[c]{} P) = (\alpha P - \{c\}) \cup \{d\}$$

When drawing a picture of a process, the channels are drawn as arrows in the appropriate direction, and labelled with the name of the channel:



Let P and Q be processes, and let c be an output channel of P and an input channel of Q. When P and Q are composed concurrently in the system $(P \| Q)$, communication will occur on channel c on each occasion that P outputs a message and Q simultaneously inputs that message. An outputting process specifies an unique value for the message, whereas the inputting process is prepared to accept any communicable value. Thus the event that will actually occur is the communication $\langle c,v \rangle$, where v is the value specified by the outputting process. This requires the obvious constraint that the channel c must have the same alphabet at both ends, i.e.

$$\alpha c'(P) = \alpha c'(Q)$$

In general, the value to be output by a process is specified by means of expression containing variables to which a value has been assigned by ~~some previous~~ input, as illustrated in the following examples.

X1 A process which immediately copies every message it has input from the left by outputting it to the right

$$COPY = \mu X(left?x \longrightarrow right!x \longrightarrow X)$$

X2 A process like COPY, except that every number input is doubled before it is output:

$$DOUBLE = \mu X(left?x \longrightarrow right!(x+x) \longrightarrow X)$$

X3 The value of a punched card is a sequence of eighty characters, which may be read as a single value along the left channel. A process which reads cards and outputs their characters one at a time, interposing an extra blank character " " after each card:

$$UNPACK = P_{\langle\rangle}$$

where $P_{\langle\rangle} = left?s \longrightarrow P_s$

and $P_{\langle x\rangle} = right!x \longrightarrow right!\text{" "} \longrightarrow P_{\langle\rangle}$

$P_{\langle x\rangle s} = right!x \longrightarrow P_s$ if $s \neq \langle\rangle$

X4 A process which inputs characters one at a time from the left, and assembles them into lines of 125 characters' length. Each completed line is output on the right as a single array-valued message.

$$PACK = P_{<>}$$

where $P_l = right!l \longrightarrow P_{<>}$ if $\#l = 125$

$= left?x \longrightarrow P_{l<x>}$ if $\#l < 125$

Here, P_l describes the behaviour of the process when it has input and packed the characters in the sequence l; they are waiting to be output when the line is long enough.

X5 A process which copies from left to right, except that consecutive pairs of asterisks are replaced by a single " ↑ "

```
SQUASH = μX. left?x ⟶
           if x ≠ "*" then (right!x ⟶ X)
           else left?y ⟶(if y = "*" then (right!"↑" ⟶ X)
                               else (right!"*" ⟶ right!y ⟶ X))
```

A process may be prepared initially to communicate on any one of a set of channels, leaving the choice between them to the other processes with which it is connected. For this purpose we adapt the choice notation introduced in chapter One. If c and d are distinct channel names

$$(c?x \longrightarrow P(x) \mid d?y \longrightarrow Q(y))$$

denotes a process which initially inputs x on c and then behaves like P(x), or initially inputs y on channel d and then behaves like Q(y). The choice is determined by whichever of the corresponding outputs is ready first.

X6 A process which accepts input on either of the two channels left1 or left2, and immediately outputs the message to the right:

$$MERGE = (left1?x \longrightarrow right!x \longrightarrow MERGE$$
$$\mid left2?x \longrightarrow right!x \longrightarrow MERGE)$$

The output of this process is an interleaving of the messages input from left1 and left2.

X7 A process that is always prepared to input a value on the left, or to output to the right the value which it has most recently input.

$$VAR = left?x \longrightarrow VAR_x$$

where $VAR_x = (left?y \longrightarrow VAR_y$

$\quad\quad\quad\quad | right!x \longrightarrow VAR_x)$

here VAR_x behaves like a program variable with current value x. New values are assigned to it by communication on the left channel, and its current value is obtained by communication on the right channel.

X8 A process which inputs from "up" and "left", and outputs to "down" a function of what it has input, before repeating:

$$MULT(v) = \mu X.(up?sum \longrightarrow left?prod \longrightarrow down!(sum + v*prod) \longrightarrow X)$$

X9 A process which is at all times ready to input a message on the left, and to output on its right the first message which it has input but not yet output.

$$BUFFER = P_{\langle\rangle}$$

where $P_{\langle\rangle} = left?x \longrightarrow P_{\langle x\rangle}$

$P_{\langle x\rangle s} = (left?y \longrightarrow P_{\langle x\rangle s\langle y\rangle}$

$\quad\quad\quad\quad | right!x \longrightarrow P_s)$

This process is like a queue; messages join the queue on the left and leave it on the right, in the same order but possibly after some delay.

X10 A process which behaves like a stack of messages. When empty, it responds to the signal "empty". At all times it is ready to input a new message from the left and put it on top of the stack; and whenever nonempty, it is prepared to output and remove the top element of the stack

$$STACK = P_{\langle\rangle}$$

where $P_{\langle\rangle} = (empty \longrightarrow P_{\langle\rangle} \mid left?x \longrightarrow P_{\langle x\rangle})$

$P_{\langle x\rangle s} = (right!x \longrightarrow P_s$

$\quad\quad\quad\quad | left?y \longrightarrow P_{\langle y\rangle\langle x\rangle s})$

4.2.1 Implementation

In a LISP implementation of communicating processes, the event $<c,v>$ is naturally represented by the list "(c,v), which is constructed by

cons("c,cons(v,NIL))

An input command inspects a proffered communication to check the channel name. If this is not right the result is "BLEEP. But if the name matches, the content of the message is extracted and used. Thus the input command

$(c?x \longrightarrow P(x))$

is implemented as a LISP function call

input("c, $\lambda x.P(x)$)

where the input function is defined:

```
input(c,F)  =   λy. if y = NIL ∨ atom(y) then "BLEEP
                    else if car(y) ≠ c then "BLEEP
                         else F(car(cdr(y)))
```

This implementation of input follows closely its definition as a general communicating process. A similar definition cannot be given for output, since in order to test whether a process is ready for output on a given channel c it would be necessary to construct and test the event $<c,v>$ for every possible message value v in αc. If αc is large this would be very inefficient; if αc is infinite it would be impossible. For this reason, we implement an outputting process as a function which is applied first to the channel name alone. If the process is not ready to output on that channel, it gives the familiar "BLEEP. But if it <u>is</u> ready, then its result is a <u>pair</u>

(v,P)

where v is the value it is ready to output
and P is its subsequent behaviour.

Thus the outputting process

$(c!v \longrightarrow P)$

is implemented as the LISP function call

output("c,v,P)

where the output function is defined:

```
     output(c,v,P)  =    λx. if x ≠ c then "BLEEP
                                  else cons(v,P).
Examples
X1   COPY  =  LABEL X.
              input("left,   λx.output("right,x,X))

X2   PACK  =  P(NIL)

where  P   =  LABEL X.
               λl. if length(l) = 125 then output ("right,l,X(NIL))
                      else input ("left,
                             λx. X(append(l,cons(x,NIL))))
```

4.2.2 Specifications

In specifying the behaviour of a communicating process, it is convenient to describe separately the sequences of messages that pass along each of the channels. If c is a channel name, we define

$$tr.c \;=\; strip_c^*(tr \upharpoonright \alpha c).$$

where $\quad strip_c(\langle c,v \rangle) \;=\; v.$

For example,

if $\quad tr \;=\; \langle\langle left,3\rangle, \langle right,3\rangle, \langle left,37\rangle\rangle$

then $tr.left = \langle 3,37\rangle$

and $\;tr.right = \langle 3\rangle \qquad$ and $\quad tr.mid = \langle\,\rangle$

It is convenient also just to omit the "tr.", and write "right $\leq$ left" instead of "tr.right $\leq$ tr.left".

Another useful definition places a lower bound on the length of a prefix:

$$s \overset{n}{\leq} t \;=\; s \leq t \wedge \#t \leq \#s + n.$$

From this it follows that:

$$s \overset{0}{\leq} t \;\equiv\; (s = t)$$

$$s \overset{n}{\leq} t \wedge t \overset{m}{\leq} u \;\Longrightarrow\; s \overset{n+m}{\leq} u$$

$$s \leq t \;\equiv\; \exists n.\; s \overset{n}{\leq} t$$

X1. COPY <u>sat</u> right $\overset{1}{\leq}$ left

X2. DOUBLE <u>sat</u> right $\overset{1}{\leq}$ double* (left)

X3. UNPACK <u>sat</u> right $\leq$ $^\wedge/$ extsp* (left)

where $\quad {}^\wedge/\langle s_0, s_1, \ldots, s_{n-1}\rangle \;=\; s_0{}^\wedge\, s_1{}^\wedge \ldots {}^\wedge s_{n-1} \qquad$ (see 1.9.3)

and $\;$ extsp (s) $\;=\;$ $s^\wedge\langle space\rangle$

X4. PACK <u>sat</u> $(({}^\wedge/right \overset{125}{\leq} left) \wedge (\#^*\, right) \in \{125\}^*)$

This specification states that each element output on the right is itself a sequence of length 125, and the catenation of all these sequences is an initial subsequence of what has been input on the left.

If $\oplus$ is a binary operator, it is convenient to apply it distributively to the corresponding elements of two sequences. The length of the resulting sequence is equal to that of the shorter operand.

$$s \oplus t = \langle\rangle \quad \text{if} \quad s = \langle\rangle \quad \text{or } t = \langle\rangle$$
$$= \langle s_0 \oplus t_0 \rangle ^\wedge (s' \oplus t') \quad \text{otherwise}$$

Clearly $(s \oplus t)[i] = s[i] \oplus t[i]$ for $i < \min(\#s, \#t)$.
and $s \overset{n}{\leq} t \Longrightarrow (s \oplus u \overset{n}{\leq} t \oplus u) \wedge (u \oplus s \overset{n}{\leq} u \oplus t)$

X5. The Fibonacci sequence

$\langle 1,1,2,3,5,8, \ldots \rangle$

is defined by the recurrence relation

$$fib[0] = fib[1] = 1$$
$$fib[i+2] = fib[i+1] + fib[i].$$

The second line can be rewritten using the ' operator to "shift" the sequence to the left.

$$fib'' = fib' + fib$$

A process which outputs the Fibonacci sequence to the right is specified

$$FIB \ \underline{sat} \ (right \leq \langle 1,1 \rangle \vee (\langle 1,1 \rangle \leq right \wedge right'' \overset{1}{\leq} right' + right))$$

X6. A variable with value x outputs on the right only the value most recently input on the left, or x, if there is no such input.

$$VAR_x \ \underline{sat} \ (\overline{tr}_0 \in \alpha right \Longrightarrow \overline{right}_0 = \overline{\langle x \rangle ^\wedge \langle left \rangle}_0)$$

where $\tilde{s}_0$ is the last element of s (1.9.5).

This is an example of a process that cannot be adequately specified solely in terms of the sequences of messages on its separate channels. It is also necessary to know the order in which the communications on separate channels are interleaved, for example that the latest communication is on the right.

X7. The MERGE process produces an interleaving of the two sequences input on left1 and left2.

MERGE sat $\exists r.\ \text{right} \leq r \wedge \text{interleaves}\ (r, \text{left1}, \text{left2})$

X8. BUFFER sat right $\leq$ left

A process which satisfies the specification (right $\leq$ left) may be used as a "transparent" communications protocol, which is guaranteed to deliver on the right only those messages which have been submitted on the left, and in the same order. This must be achieved in spite of the possibility that the place where the messages are submitted is widely separated from the place where they are received; and that the communications medium which connects the two places is somewhat unreliable.

4.3 Communication

Let P and Q be processes, and let c be a channel used for output by P and for input by Q. Thus the set αc, containing all communication events of the form $\langle c,v \rangle$, is within the intersection of the alphabet of P with the alphabet of Q. When these processes are composed concurrently in the system $(P \| Q)$, a communication $\langle c,v \rangle$ can occur only when both processes engage simultaneously in that event, i.e., whenever P outputs a value on the channel c, and Q simultaneously inputs the same value. An inputting process is prepared to accept any communicable value, so it is the outputting process ~~that determines~~ which actual message value ~~is transmitted~~ on each occasion.

Thus output may be regarded as a specialised case of the prefix operator, and input a special case of choice; and this leads to the law:

L1. $(c!v \longrightarrow P) \| (c?x \longrightarrow Q(x)) = c!v \longrightarrow (P \| Q(v))$

Note that c!v remains on the right hand side of this equation as an observable action in the behaviour of the system. This represents the physical possibility of tapping the wires connecting the components of a system, and of thereby keeping a log of their internal communications. It is also a help in reasoning about the system.

Let c be the name of a channel along which P and Q communicate. In the specification of P, c stands for the sequence of messages communicated by P on c. Similarly, in the specification of Q, c stands for the sequence of messages communicated by Q. Fortunately, by the very nature of communication, when P and Q communicate on c the sequences of messages sent and received must at all times be identical. Consequently this sequence must satisfy both the specification of P and the specification of Q. The same is true for all channels in the intersection of their alphabets.

Consider now a channel d in the alphabet of P but <u>not</u> of Q. This channel cannot be mentioned in the specification of Q, so the values communicated on it are constrained only by the specification of P. Similarly, it is Q that determines the properties of the communications on its own channels. Consequently a specification of the behaviour of $(P \parallel Q)$ can be simply formed as the logical conjunction of the specification of P with the specification of Q. However, this simplification is valid only when the specifications of P and Q are expressed wholly in terms of the channel names, which is not always possible.

X1. Let $P = (\text{left?x} \longrightarrow \text{mid!}(x \times x) \longrightarrow P)$

$Q = (\text{mid?y} \longrightarrow \text{right!}(173 \times y) \longrightarrow Q)$

Clearly P <u>sat</u> $\text{mid} \overset{1}{\leq} \text{square}^*(\text{left})$

and Q <u>sat</u> $\text{right} \overset{1}{\leq} 173 \times \text{mid}$

It follows that

$(P \parallel Q)$ <u>sat</u> $(\text{right} \overset{1}{\leq} 173 \times \text{mid}) \wedge (\text{mid} \overset{1}{\leq} \text{square}^*(\text{left}))$

The specification here implies

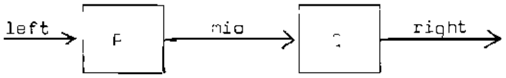
$$\text{right} \leq 173 \times \text{square}^*(\text{left})$$

which was presumably the original intention.

When communicating processes are connected by the concurrency operator $\parallel$, the resulting formulae are highly suggestive of a physical implementation method in which electronic components are connected by wires along which they communicate. The purpose of such an implementation is to increase the speed with which useful results can be produced.

The technique is particularly effective when the same calculation must be performed on each member of a stream of input data, and the results must be output at the same rate as the input, but possibly after an initial delay. Such systems are called data flow networks.

A picture of a system of concurrent processes closely represents their physical realisation. An output channel of one process is connected to an input channel of the other process which has the same name, but channels in the alphabet of only one process are left free. Thus the example X1 can be drawn:



X2. Two streams of numbers are to be input from left1 and left2. For each x read from left1 and each y from left2, the number $(a \times x + b \times y)$ is to be output on the right. The speed requirement dictates that the multiplications must proceed concurrently. We therefore define two processes, and compose them:

$$X21 = (left1?x \longrightarrow mid!(a \times x) \longrightarrow X21)$$

$$X22 = (left2?y \longrightarrow mid?z \longrightarrow right!(z + b \times y) \longrightarrow X22)$$

$X2 = (X21 \| X22)$

~~Clearly,~~ X2 sat $(mid \overset{1}{\leqslant} a \times left1 \wedge right \overset{1}{\leqslant} mid + b \times left2)$

$\therefore$ X2 sat $(right \leqslant a \times left1 + b \times left2)$

X3. A stream of numbers is to be input on the left, and on the right is output a weighted sum of consecutive pairs of input numbers, with weights a and b. More precisely, we require that

$$right \leqslant a \times left + b \times left'$$

The solution can be constructed by adding a new process X23 to the solution of X2,

$$X3 = (X2 \| X23)$$

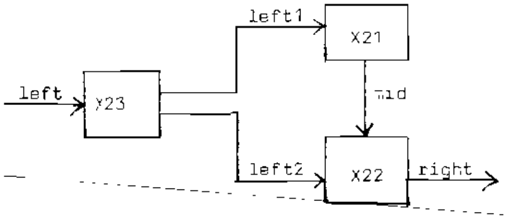
where X23 sat $(left1 \overset{1}{\leqslant} left \wedge left2 \overset{1}{\leqslant} left')$

X23 can be defined:

$$X23 = (left?x \longrightarrow left1!\,x \longrightarrow$$
$$(\mu X.\ left?x \longrightarrow left2!x \longrightarrow left1!x \longrightarrow X))$$

It copies from left to both left1 and left2, but omits the first element in the case of left2.

A picture of the network of X3 is



The cycle in this diagram reveals a danger of deadlock. For example, if the two outputs in the loop of X23 were reversed, deadlock would occur rapidly. In proving absence of deadlock it is often possible to ignore the content of the messages, and regard each communication on channel c as a single event named "c". Communications on unconnected channels can be ignored. Thus X3 can be written in terms of these events:

$$X3 = (\mu X.left1 \longrightarrow mid \longrightarrow X)$$
$$\| (\mu Y.\, left2 \longrightarrow mid \longrightarrow Y)$$
$$\| (left1 \longrightarrow \mu Z.left2 \longrightarrow left1 \longrightarrow Z)$$
$$= left1 \longrightarrow left2 \longrightarrow mid \longrightarrow X3.$$

This proves that X3 cannot deadlock.

These examples show how data flow networks can be set up to compute one or more streams of results from one or more streams of input data. The shape of the network corresponds closely to the structure of operands and operators appearing in the expressions to be computed. When these patterns are large but regular, it is convenient to use subscripted names for channels, and to introduce an iterated notation for concurrent combination:

$$\mathop{\|}_{i<n} P(i) = (P(0) \| P(1) \| \ldots \| P(n-1))$$

A regular network of this kind is known as an iterative array.

X4. The channels $\{\text{left}_j \mid j < n\}$ are used to input successive coordinates in n-dimensional space. Each coordinate set is to be scalar multiplied by a fixed vector V of length n, and the resulting scalar product is to be output to the right; or more formally

$$\text{right} \leqslant \sum_{j=0}^{n-1} V_j \times \text{left}_j$$

The interval between successive outputs permits only one intervening multiplication, so at least n processes are required.

Let us define the $\sum$ in the specification by the usual induction.

$$\text{mid}_0 = 0$$

$$\text{mid}_{j+1} = V_j \times \text{left}_j + \text{mid}_j \qquad \text{for } j < n$$

$$\text{right} = \text{mid}_n.$$

Thus we have split the specification into a conjunction of n+1 component equations, each containing at most one multiplication. All that is required is to write a process for each equation.

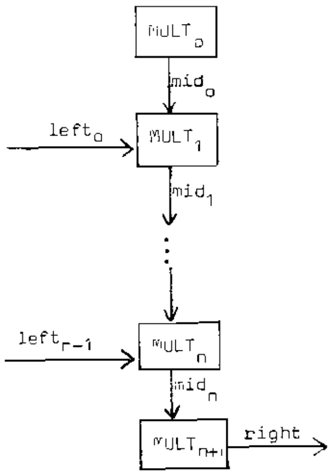
$$\text{MULT}_0 = (\mu X.\ \text{mid}_0!0 \longrightarrow X)$$

$$\text{MULT}_{j+1} = (\mu X.\ \text{left}_j?x \longrightarrow \text{mid}_j?y \longrightarrow \text{mid}_{j+1}!(V_j \times x + y) \longrightarrow X) \qquad \text{for } j < n$$

$$\text{MULT}_{n+1} = (\mu X.\ \text{mid}_n?x \longrightarrow \text{right}!x \longrightarrow X)$$

$$\text{NETWORK} = \mathop{\big\|}_{j < n+2} \text{MULT}_j$$

A picture of the connection diagram is:

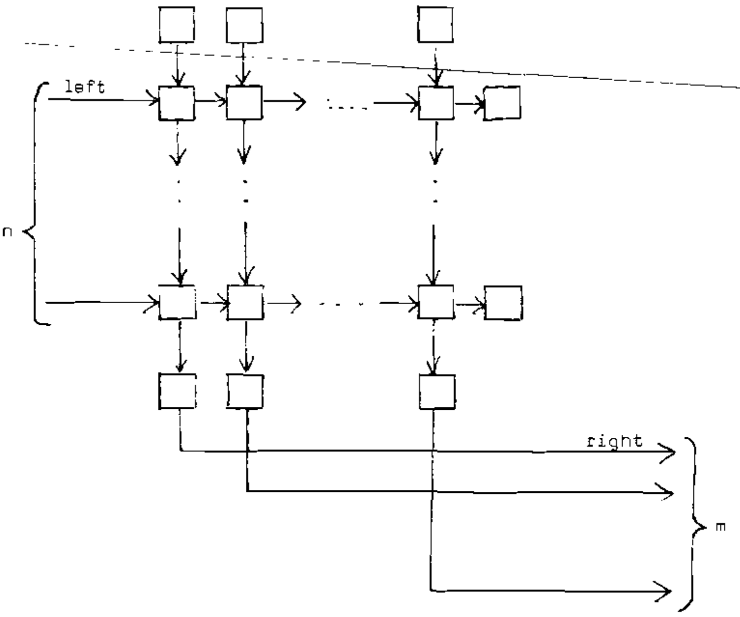


X5. This is similar to X4, except that m scalar products are required almost simultaneously. Effectively, the channel left_j (for $j < n$) is to be used to input the jth column of an infinite array; this is to be multiplied by the $n \times m$ matrix M, and the ith column of the result is to be output on right_i for $i < m$. In formulae:

$$\mathrm{right}_i = \sum_{j<n} M_{ij} \times \mathrm{left}_j$$

The coordinates of the result are required as rapidly as before, so at least $m \times n$ processes are required.

The solution is based on the diagram



Each column of this array (except the last) is modelled on the solution to X4; but it copies each value input on its horizontal input channel to its neighbour on its horizontal output channel. The processes on the right margin merely absorb the values they input.

The details of the solution are left as an exercise.

X6. The input on channel c is to be interpreted as the successive digits of a natural number C in b-adic notation:

$$C = \sum_{i \geq 0} c[i] \times b^i$$

where $c[i] < b$ for all i

Given a fixed multiplier M, the output on channel d is to be the successive digits of the product $M * C$. The digits are to be output after minimal delay.

Let us specify the problem more precisely. The desired output d is:

$$d = \sum_{i \geq 0} M \times c[i] \times b^i$$

The j th element of d must be the j th digit of this:

$$d[j] = ((\sum_{i \geq 0} M \times c[i] \times b^i) \div b^j) \underline{\text{mod}}\ b$$

$$= (M \times c[j] + z_j) \underline{\text{mod}}\ b$$

where $z_j = (\sum_{i<j} M \times c[i] \times b^i) \div b^j$.

z_j is the "carry" term, and can readily be proved to satisfy the inductive dafinition:

$$z_0 = 0$$

$$z_{j+1} = ((M \times c[j] + z_j) \div b)$$

We therefore define a process MULT1(z), which keeps the carry z as a parameter:

$$\text{MULT1}(z) = c?x \longrightarrow d!(M \times x + z)\ \underline{\text{mod}}\ b$$

$$\longrightarrow \text{MULT1}\ ((M \times x + z) \div b)$$

The initial value of z is zero, so the required solution is:

MULT1(0)

X7. The problem is the same as X6, except M is a multi-digit number:

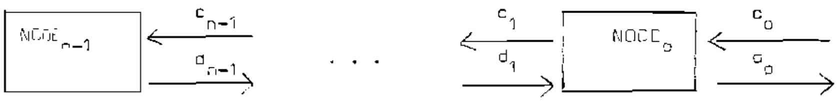
$$M = \sum_{i<m} M_i \times b^i$$

A single processor can multiply only single-digit numbers. However, output is to be produced at a rate which allows only one multiplication per digit. Consequently, at least n processors are required. We will get each $NODE_i$ to look after one digit M_i of the multiplier.

The basis of a solution is the traditional manual algorithm for multi-digit multiplication, except that the partial sums are added immediately to the next row of the table:

..... 153091	C	the incoming number
253	M	the multiplier
.... 306182	$M_2 \times C$	computed by $NODE_2$
....765455	$M_1 \times C$	computed by $NODE_1$
.....827275	$25 \times C$	
....459273	$M_0 \times C$	computed by $NODE_0$
....732023	$M \times C$	

The nodes are connected as shown:



The original input comes in on c_0 and is propagated leftward on the c channels. The partial answers are propagated rightward on the d channels, and the desired answer is output on d_0. Fortunately each node can give one digit of its result before communicating with its left neighbour. Furthermore, the leftmost node can be defined to behave like the answer to X6.

$$NODE_{n-1}(z) = c_{n-1} ?\ x \longrightarrow c_{n-1}\ !\ (M_{n-1} \times x + z) \ \underline{\text{mod}}\ b$$

$$\longrightarrow NODE_{n-1}((M_{n-1} \times x + z) \div b)$$

The remaining nodes are similar, except that each of them adds the result from its left neighbour to the carry. For $k < n-1$

$$NODE_k(z) = c_k?x \longrightarrow d_k!(M_k \times x + z) \underline{\text{mod}}\ b$$
$$\longrightarrow c_{k+1}!x \longrightarrow d_{k+1}?y$$
$$\longrightarrow NODE_k(y + (M_k \times x + z) \div b)$$

The whole network is defined

$$\big\|_{i<n} NODE_i(U)$$

X7 is a simple example from a class of ingenious network algorithms, in which there is an essential cycle in the directed graph of communication channels. But the statement of the problem has been much simplified by assumption that the multiplier is known in advance and fixed for all time. In a practical application, it is much more likely that such parameters would have to be input along the same channel as the subsequent data; and would have to be reinput whenever it is required to change them. The implementation of this requires great care, but little ingenuity.

A simple implementation method is to introduce a special symbol, say "reload", to indicate that the next number or numbers are to be treated as a change of parameter; and if the number of parameters is variable, an "endreload" symbol ~~may also be~~ introduced.

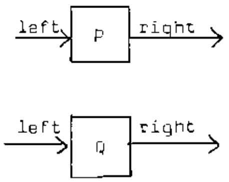
X8. Same as X4, except that the parameters V_j are to be reloaded by the number immediately following a "reload" symbol. The definition of $MULT_{j+1}$ needs to be changed.

$$MULT_{j+1}(v) = left_j?x \longrightarrow$$
$$\underline{\text{if}}\ x = \text{reload}\ \underline{\text{then}}\ (left?y \longrightarrow MULT_{j+1}(y))$$
$$\underline{\text{else}}\ (mid_j?y \longrightarrow mid_{j+1}!(v \times x + y) \longrightarrow MULT_{j+1}(v))$$

The network constructed from these nodes will not work unless the reload signal is sent at the same time on all the $left_j$ channels.

4.4 Pipes

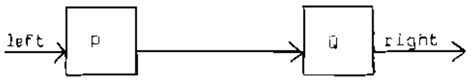
In this section we shall confine attention to processes with only two channels in their alphabet, namely an input channel "left" and an output channel "right". Such processes are called <u>pipes</u>, and they may be pictured:



These processes may be joined together so that the right channel of P is connected to the left channel ~~of Q, and the~~ sequence of messages output by P and input by Q on this internal channel is concealed from their common environment. The result of the connection is denoted

$$P \gg Q$$

and may be pictured as the series



This picture shows the concealment of the connecting channel by not giving it a name. It also shows that all messages input on the left channel of $(P \gg Q)$ are input by P, and all messages output on the right channel of $(P \gg Q)$ are output by Q. Finally $(P \gg Q)$ is itself a pipe, and may again be placed in series with other pipes:

$(P \gg Q) \gg R$, $(P \gg Q) \gg (R \gg S)$, etc

The same facts are expressed by the alphabet constraints:

$$\alpha(P >> Q) = \alpha left(P) \cup \alpha right(Q);$$

and a further constraint states that the connected channels are capable of transmitting the same kind of message:

$$\alpha right'(P) = \alpha left'(Q)$$

X1 A pipe which outputs each input value multiplied by four:

QUADRUPLE = DOUBLE >> DOUBLE

X2 A process which inputs cards of eighty characters and outputs them in sequence tightly packed in lines of 125 characters each. The card boundaries are indicated by an extra space.

UNPACK >> PACK

X3 Same as X2, except that each pair of consecutive asterisks is replaced by " ↑ "

(UNPACK >> SQUASH) >> PACK

X4 Same as X2, except that the reading of cards may continue when the printer is held up, and later the printing ~~can continue~~ when ~~the card~~ reader is held up.

UNPACK >> (BUFFER >> PACK)

X5 Same as X4, except that only one line of text is buffered:

UNPACK >> PACK >> COPY

X6 A double buffer, which accepts up to two messages before requiring output of the first

$COPY \gg COPY$

4.4.1 Laws

The most useful algebraic property of chaining is associativity.

L1 $P \gg (Q \gg R) = (P \gg Q) \gg R$

The remaining laws show how input and output can be implemented in a pipe; they enable process descriptions to be simplified by a form of symbolic execution.

L2 $(right!v \longrightarrow P) \gg (left?y \longrightarrow Q(y)) = P \gg Q(v)$

If one of the processes is determined to communicate with the other, but the other is prepared to communicate axternally, it is the external communication that takes place first.

L3 $(right!v \longrightarrow P) \gg (right!w \longrightarrow Q) =$

$$right!w \longrightarrow ((right!v \longrightarrow P) \gg Q)$$

L4 $(left?x \longrightarrow P(x)) \gg (left?y \longrightarrow Q(y))$

$= left?x \longrightarrow (P(x) \gg (left?y \longrightarrow Q(y)))$

If both processes are prepared for external communication, then either may happen first:

L5 $(left?x \longrightarrow P(x)) \gg (right!w \longrightarrow Q)$

$$= (left?x \longrightarrow (P(x) \gg (right!w \longrightarrow Q))$$
$$\mid right!w \longrightarrow ((left?x \longrightarrow P(x)) \gg Q))$$

The same law L5 is equally valid when the operator $\gg$ is replaced by $\gg^{\dashv} \gg$, since pipes in the middle of a chain cannot communicate directly with the environment.

L6 $(\text{left?}x \longrightarrow P(x)) \gg R \gg (\text{right!}w \longrightarrow Q)$

$$= (\text{left?}x \longrightarrow (P(x) \gg R \gg (\text{right!}w \longrightarrow Q)) \;|\; \text{right!}w \longrightarrow ((\text{left?}x \longrightarrow P(x)) \gg R \gg Q))$$

Similar generalisations may be made to the other laws:

L7 If R is a chain of processes all starting with output to the right,

$$R \gg (\text{right!}w \longrightarrow Q) \;=\; \text{right!}w \longrightarrow (R \gg Q)$$

L8 If R is a chain of processes all starting with input from the left,

$$(\text{left?}x \longrightarrow P(x)) \gg R \;=\; \text{left?}x \longrightarrow (P(x) \gg R).$$

X1 Let us define

$R(y) = (\text{right!}y \longrightarrow \text{COPY}) \gg \text{COPY}$

$\therefore R(y) = (\text{right!}y \longrightarrow \text{COPY}) \gg (\text{left?}y \longrightarrow \text{right!}y \longrightarrow \text{COPY})$ def COPY

$= \text{COPY} \gg (\text{right!}y \longrightarrow \text{COPY})$ L2

X2 $\text{COPY} \gg \text{COPY}$

$= (\text{left?}x \longrightarrow \text{right!}x \longrightarrow \text{COPY}) \gg \text{COPY}$ def COPY

$= \text{left?}x \longrightarrow ((\text{right!}x \longrightarrow \text{COPY}) \gg \text{COPY})$ L4

$= \text{left?}x \longrightarrow R(x)$ def R(x)

X3 From ~~the last line of~~ X1 we deduce

$R(y) = (\text{left?}x \longrightarrow \text{right!}x \longrightarrow \text{COPY}) \gg (\text{right!}y \longrightarrow \text{COPY})$

$= (\text{left?}x \longrightarrow (\text{right!}x \longrightarrow \text{COPY}) \gg (\text{right!}y \longrightarrow \text{COPY}) \;|\; \text{right!}y \longrightarrow (\text{COPY} \gg \text{COPY})))$ L5

$= (\text{left?}x \longrightarrow \text{right!}y \longrightarrow R(x) \;|\; \text{right!}y \longrightarrow \text{left?}x \longrightarrow R(x))$ L3, X2

This shows that a double buffer, after input of its first message is prepared either to output that message or to input a second message before doing so.

4.4.2 Implementation

In the implementation of (P >> Q) three cases are distinguished.

(1) If communication can take place on the internal connecting channel, it does so immediately, without consideration of the external environment.

(2) Otherwise, if the environment is interested in communication on the left channel, this is dealt with by P.

(3) Or if the environment is interested in the right channel, this is dealt with by Q.

```
chain (P,Q) =
  if P("right) ≠ "BLEEP & Q("left) ≠ "BLEEP
      then chain (cdr(P("right)),
                 Q(cons("left, cons(car(P("right)), NIL))))
else  λx. if x = "right
          then if Q("right) = "BLEEP then "BLEEP
                else cons (car(Q("right)),
                           chain(P, cdr(Q("right))))
else if atom(x) then "BLEEP
else if car(X) = "left
        then if P(x) = "BLEEP then "BLEEP
                else  chain (P(car(cdr (x))),Q)
else "BLEEP
```

4.4.3 Livelock

The chaining operator connects two processes by just one channel; and so it introduces no risk of deadlock. If both P and Q are non-stopping, then (P >>Q) will not stop either. Unfortunately there is a new danger that the processes P and Q will spend the whole time communicating with each other, so that (P >>Q) never again communicates with the external world. This phenomenon is known as livelock, and is illustrated by the following trivial example:

$$P = (right!1 \longrightarrow P)$$

$$Q = (left?x \longrightarrow Q).$$

$(P >> Q)$ is obviously a useless process: it is even worse than STOP, in that like an endless loop it may consume unbounded computing resources without achieving anything. A less trivial example is $(P >> Q)$, where

$$P = (right!1 \longrightarrow P \mid left?x \longrightarrow P1(x))$$

$$Q = (left?x \longrightarrow Q \mid right!1 \longrightarrow Q1)$$

In this example, livelock derives from the mere possibility of infinite internal communication; it exists even though the choice of external communication to the left and right is offered on every possible occasion, and even though after such external communications the subsequent behaviour of $(P >> Q)$ would not suffer from livelock. Livelock is a consequence of concealing the messages on the internal channel; it is a special case of divergence, which is discussed in 3.5.2.

A simple method to prove $(P >> Q)$ is free of livelock is to show that P is <u>left-guarded</u> in the sense that it can never output an infinite series of messages to the right without interspersing inputs from the left. To ensure this, we must prove that the length of the sequence output to the right is at all times bounded above by some well-defined function f of the sequence of values input from the left; or more formally, we define

$$P \text{ is left-guarded} \equiv$$
$$\exists f.\quad P\ \underline{\text{sat}}\ (\#right \leqslant f(left))$$

Left-guardedness is often simply obvious from the text of P, using the law L1.

L1 If every recursion used in the definition of P is guarded by an input from the left, then P is left-guarded.

L2 If P is left-guarded then $(P >> Q)$ is free of livelock.

Exactly the same reasoning applies to right-guardedness of the second operand of >> .

L3 If Q is right-guarded then $(P \gg Q)$ is free of livelock.
Examples

X1 The following are left-guarded by L1

COPY, DOUBLE, SQUASH, BUFFER

X2 The following are left-guarded in accordance with the original definition, because

UNPACK <u>sat</u> $\#\text{right} \leq \#(\wedge/\text{left}) + \#\text{left}$

PACK <u>sat</u> $\#\text{right} \leq \#\text{left}$

X3 BUFFER is <u>not</u> right-guarded, since it can input arbitrarily many messages from the left without ever outputting to the right.

4.4.4 Specifications

A specification of a pipe can often be expressed as ~~a relation~~ S(left,right) between the sequence of messages input on the left channel and the sequence of messages output on the right. When two pipes are connected in series, the sequence "right" produced by the left operand is equated with the sequence "left" consumed by the right operand; and this common sequence is then concealed. All that is known of the concealed sequence is that it exists. But we also need to avert the risk of livelock. Thus we explain the rule

L1 If P <u>sat</u> S(laft,right)

and Q <u>sat</u> T(left,right)

and if P is left-guarded <u>or</u> Q is right-guarded

then $(P \gg Q)$ <u>sat</u> $\exists s.\ P(\text{left},s) \wedge Q(s,\text{right})$.

This states that ths relation between "left" and "right" which is maintained by $(P \gg Q)$ is the normal relational composition of the relation for P with the relation for Q.

X1 DOUBLE <u>sat</u> right $\overset{1}{\leq}$ double*(left)

DOUBLE is left-guarded <u>and</u> right-guarded.

$\therefore$ (DOUBLE $\gg$ DOUBLE) <u>sat</u>

$\exists s.\ (s \overset{1}{\leq} \text{double}^*(\text{left}) \wedge \text{right} \overset{1}{\leq} \text{double}^*(s))$

$\equiv$ right $\overset{2}{\leq}$ double*(double*(left))

$\equiv$ right $\overset{2}{\leq}$ quadruple*(lsft)

X2 Let us use recursion together with >> to give an alternative definition of a buffer.

$$\mathrm{BUFF} = \mu X.(\mathrm{left}?x \longrightarrow (X \gg (\mathrm{right}!x \longrightarrow \mathrm{COPY})))$$

We wish to prove that

$$\mathrm{BUFF}\ \underline{\mathrm{sat}}\ (\mathrm{right} \leq \mathrm{left})$$

Assume that

$$X\ \underline{\mathrm{sat}}\ \#\mathrm{left} \geq n \vee \mathrm{right} \leq \mathrm{left}.$$

We know that

$$\mathrm{COPY}\ \underline{\mathrm{sat}}\ \mathrm{right} \leq \mathrm{left}$$

$$\begin{aligned} \therefore\ (\mathrm{right}!x \longrightarrow \mathrm{COPY})\ \underline{\mathrm{sat}}\ ((\mathrm{right} = \mathrm{left} = \langle\rangle \\ \vee (\mathrm{right} \geq \langle x\rangle \wedge \mathrm{right}' \leq \mathrm{left})) \\ \Longrightarrow \mathrm{right} \leq \langle x\rangle\ \mathrm{left} \end{aligned}$$

Since the right operand is right-guarded, by L1 and the assumption

$$\begin{aligned} (X \gg (\mathrm{right}!x \longrightarrow \mathrm{COPY}))\ \underline{\mathrm{sat}}\ (\exists s.(\#\mathrm{left} \geq n \vee s \leq \mathrm{left}) \\ \wedge\ \mathrm{right} \leq \langle x\rangle^\frown s) \\ \Longrightarrow (\#\mathrm{left} \geq n \vee \mathrm{right} \leq \langle x\rangle^\frown \mathrm{left}) \end{aligned}$$

$$\begin{aligned} \therefore\ \mathrm{left}?x \longrightarrow (\ \ldots\ )\ \underline{\mathrm{sat}}\ \mathrm{right} = \mathrm{left} = \langle\rangle \\ \vee (\mathrm{left} \geq \langle x\rangle \\ \wedge (\#\mathrm{left}' \geq n \vee \mathrm{right} \leq \langle x\rangle\ \mathrm{left}')) \\ \Longrightarrow \#\mathrm{left} \geq n{+}1 \vee \mathrm{right} \leq \mathrm{left}. \end{aligned}$$

The desired conclusion follows by tha proof rule for recursive processes.

4.4.3 Buffers and Protocols

A buffer is a process which outputs on the right exactly the same sequence of messages as it has input from the left, though possibly after some delay; furthermore, when non-empty, it is always ready to output on the right. More formally, we define a buffer to be a process P which never stops, which is free of livelock, and which meets the specification

$$P\ \underline{\mathrm{sat}}\ (\mathrm{right} \leq \mathrm{left})$$

It follows that all buffers are left-guarded

X1 The following processes are buffers.

COPY, (COPY $\gg$ COPY), BUFF, BUFFER.

Buffers are clearly useful for storing information which is waiting to be processed. But they are even more useful as specifications of the desired behaviour of a communications protocol, which is intended to deliver messages in the same order in which they have been submitted. Such a protocol consists of two processes, a transmitter T and a receiver R, which are connected in series (T $\gg$ R). If the protocol is correct, clearly (T $\gg$ R) must be a buffer.

In practice, the wire that connects the transmitter to the receiver is quite long, and the messages which are sent along it are subject to corruption or loss. Thus the behaviour of the wire itself can be modelled by a process WIRE, which may behave not quite like a buffer. It is the task of the protocol designer to ensure that in spite of the bad behaviour of the wire, the system as a whole acts as a buffer; i.e. that:

(T $\gg$ WIRE $\gg$ R) is a buffer.

A protocol is usually built in a number of layers (T_1, R_1), $(T_2, R_2), \ldots, (T_n, R_n)$, each one using the previous layer as its WIRE:

$$T_n \gg \ldots \gg (T_2 \gg (T_1 \gg \mathrm{WIRE} \gg R_1) \gg R_2) \gg \ldots R_n$$

Of course, when the protocol is implemented in hardware, all the transmitters are collected into a single transmitter at one end and all the receivers at the other, in accordance with the changed bracketing:

$$(T_n \gg \ldots \gg T_2 \gg T_1) \gg \mathrm{WIRE} \gg (R_1 \gg R_2 \gg \ldots \gg R_n)$$

The law of associativity of $\gg$ guarantees that this regrouping does not change the behaviour of the system.

The following laws are useful in proving the correctness of protocols.

L1 If P and Q are buffers,

so are $(P \gg Q)$

and $(left?x \longrightarrow (P \gg (right!x \longrightarrow Q)))$

L2 If $T \gg R = (left?x \longrightarrow (T \gg (right!x \longrightarrow R)))$

then $(T \gg R)$ is a buffer.

The following is a generalisation of L2.

L3 If for some function f and for all z

$(T(z) \gg R(z)) = (left?x \longrightarrow$

$(T(f(x,z)) \gg (right!x \longrightarrow R(f(x,z)))))$

then $T(z) \gg R(z)$ is a buffer for all z.

X1 The following are buffers by L1.

COPY $\gg$ COPY, BUFFER $\gg$ COPY, COPY $\gg$ BUFFER,
BUFFER $\gg$ BUFFER

X2 It has been shown in 4.4.1 X1 and X2 that

$(COPY \gg COPY) = (left?x \longrightarrow (COPY \gg (right!y \longrightarrow COPY)))$

By L2 it is therefore a buffer.

X3 Phase encoding

A phase encoder is a process T which inputs a stream of bits, and outputs $\langle 0,1 \rangle$ for each 0 input and $\langle 1,0 \rangle$ for each 1 input. A decoder R reverses this translation.

$T = left?x \longrightarrow right!x \longrightarrow right!(1-x) \longrightarrow T$

$R = left?x \longrightarrow left?y \longrightarrow$ <u>if</u> $y = x$ <u>then</u> FAIL

<u>else</u> $(right!x \longrightarrow R)$

We wish to prove by L2 that $(T \gg R)$ is a buffer

$(T \gg R) = left?x \longrightarrow ((right!x \longrightarrow right!(1-x) \longrightarrow T)$

$\gg (left?x \longrightarrow left?y \longrightarrow \underline{if}\ y = x\ \underline{then}\ FAIL$

$\underline{else}\ (right!x \longrightarrow R)))$

$= left?x \longrightarrow (T \gg \underline{if}(1-x) = x\ \underline{then}\ FAIL$

$\underline{else}\ (right!x \longrightarrow R))$

$= left?x \longrightarrow (T \gg (right!x \longrightarrow R))$

Therefore $(T \gg R)$ is a buffer.

X4 Bit stuffing

The transmitter T faithfully reproduces the input bits from left to right, except that after three consecutive 1-bits which have been output, it inserts a single extra 0. Thus ~~the input 01011110~~ is output as 010111010. The receiver R removes these extra zeroes. Thus $(T \gg R)$ must be proved to be a buffer. The construction of T and R, and the proof of their correctness, are left as an exercise.

4.5 Subordination

Let P and Q be processes with

$\alpha P \subseteq \alpha Q$

In the combination $(P \parallel Q)$, each action of P can occur only when Q permits it to occur; whereas Q can engage independently in the actions of $(\alpha Q - \alpha P)$, without the permission and without the knowledge of its partner P. Thus P serves Q as a subordinate process, where Q acts as a master or main process. When communications between a subordinate process and a main process are to be concealed from their common environment, we use the asymmetric notation:

$P // Q$

This notation is used only when $\alpha P \subseteq \alpha Q$; and then

$\alpha(P // Q) = (\alpha Q - \alpha P)$

It is usually convenient to give the subordinate process a name, say m, which used in the main process for all interactions with its subordinate. The naming technique described in 2.6 can be readily extended to communicating processes, by introducing compound channel names. These take the form $\langle m,c \rangle$ (abbreviated to m.c), where m is a process name and c is the name of one of its channels.

$$\text{Let } strip_m(\langle m.c,\ v \rangle) = \langle c,v \rangle$$

for all $\langle c,v \rangle \in \alpha P$. Then

$$m{:}\ P = strip_m^{-1}(P)$$

Examples

X1 doub:DOUBLE // Q.

The subordinate process acts as a simple subroutine called from within the main process Q. Inside Q, the value of $2 \times e$ may be obtained by a successive output of the argument e on the left channel of doub, and input of the result on the right channel:

$$doub.left!e \longrightarrow (doub.right?x \longrightarrow \dots)$$

X2 One subroutine may use another as a subordinate, and do so several times.

QUADRUPLE =

(doub:DOUBLE

$//(\mu X.left?x \longrightarrow doub.left!x \longrightarrow$

$\quad doub.right?y \longrightarrow doub.left!y \longrightarrow$

$\quad doub.right?z \longrightarrow right!z \longrightarrow X)$

This is designed itself to be used as a subroutine:

quad:QUADRUPLE // ...

X3 A conventional program variable named m may be modelled as a subordinate process.

m:VAR // Q

Inside the main process Q, the value of m can be assigned, read, and updated by input and output m:=3 is implemented by m.left!3

x:=m is implemented by m.right?x $\longrightarrow$ ...

m:=m+3 is implemented by

m.right?y $\longrightarrow$ m.left!(y+3) $\longrightarrow$ ...

A subordinate process may be used to implement a data structure with a more elaborate behaviour than just a simple variable.

X4 (q:BUFFER // Q)

The subordinate process serves as an unbounded queue named q. The output "q.left!v" adds v to one end of the queue, and "q.right?y" removes an element from the other end, and gives its value to y. If the queue is empty, the queue will not respond, and the system may deadlock.

X5 A stack with name st is declared:

st:STACK // Q

Inside the main process Q, "st.left!v" can be used to push the value v onto the stack, and "st.right?x" will pop the top value. To deal with the possibility that the stack is empty, a choice construction can be used:

(st.right?x $\longrightarrow$ Q1(x)

| st.empty $\longrightarrow$ Q2)

If the stack is non-empty, the first alternative is selected; if empty, deadlock is avoided and the second alternative is selected.

A subordinate process with several channels may be used by several concurrent processes, provided that they do not use the same channels.

X6 A process P is intended to communicate a stream of values to R; these values are to be buffered by a subordinate buffer process named "b", so that output from P will not be delayed when R is not ready for input. P uses channel b.left for its output and R uses b.right for its input.

(b:BUFFER // (P || R)).

note that if P attempts to input from an empty buffer, the system will not necessarily deadlock; P will simply be delayed until Q next outputs a value to the buffer.

The subordination operator may be used to define subroutines by recursion. Each level of recursion (except the last) declares a new local subroutine to deal with the recursive call(s).

X7 Factorial

```
μX. left?n ⟶ (if n = 0 then (right!1 ⟶ X)
    else (f: X //
          (f.left!(n−1)
         ⟶ f.right?y ⟶ right!(n × y) ⟶ X)))
```

This is a boringly familiar example of recursion, expressed in an unfamiliar but rather cumbersome notational framework. A less familiar idea is that of using recursion together with subordination to implement an unbounded data structure. Each level of the recursion stores a single component of the structure, and declares a new local subordinate data structure to deal with the rest.

X8 Unbounded finite set

A process which implements a set inputs its members on its left channel. After each input, it outputs a YES if it has already input the same value, and NO otherwise.

```
     SET  =  left?x ⟶ right!NO ⟶
                  (rest:SET // LOOP)
where  LOOP  =
     left?y ⟶ (if y = x then right!YES ⟶ LOOP
                 else (rest.left!y ⟶
                          rest.right?z ⟶ right!z ⟶ LOOP))
```

The set starts empty; consequently on input of its first member x it immediately outputs NO. It then declares a subordinate process called "rest", which is going to store all members of the set except x. The LOOP is designed to input subsequent members of the set. If the newly input member is equal to x, the answer YES is sent back immediately on the right channel. Otherwise, the new member is passed on for storage by "rest". Whatever answer (YES or NO) is sent back by "rest" is passed back again, and the LOOP repeats.

X9 Binary tree

A more efficient representation of a set is as a binary tree, which relies on some given total ordering $\leq$ over its elements. Each node stores its earliest inserted element, and declares <u>two</u> subordinate trees, one to store elements smaller than the earliest, and one to store the bigger elements. The external specification of the tree is the same as X8.

$$\text{TREE} = \text{left?x} \longrightarrow \text{right!NO} \longrightarrow (\text{smaller:TREE} /\!/ \text{bigger:TREE} /\!/ \text{LOOP})$$

The design of the LOOP is left as an exercise.

4.5.1 Laws

The following obvious laws govern communications between a process and its subordinates.

L1 $(m:(c?x \longrightarrow P(x))) /\!/ (m.c!v \longrightarrow Q)$

$$= (m:P(v)) /\!/ Q$$

L2 $(m:(d!v \longrightarrow P)) /\!/ (m.d?x \longrightarrow Q(x))$

$$= (m:P) /\!/ Q(v)$$

L3 $(m:(c?x \longrightarrow P1(x) \mid d?y \longrightarrow P2(y))) /\!/ (m.c!v \longrightarrow Q)$

$$= (m:P1(v)) /\!/ Q$$

L4 $m:P /\!/ (m:Q /\!/ R) = (m:Q /\!/ R)$

L5 If m and n are distinct names

$$m:P /\!/ (n:Q /\!/ R) = n:Q /\!/ (m:P /\!/ R)$$

CHAPTER FIVE

SEQUENTIAL PROCESSES

5.1 Introduction

The process STOP is defined as one that never engages in any action. It is not a useful process, and probably results from a deadlock or other design error, rather than a deliberate choice of the designer. However, there is one good reason why a process should do nothing more, namely that it has already accomplished everything that it was designed to do. Such a process is said to terminate successfully. In order to distinguish between this and STOP, it is convenient to regard successful termination as a special event, denoted by the symbol "$\surd$" (pronounced "success"). A sequential process is defined as one which has $\surd$ in its alphabet; and naturally this can only be the last event in which it engages. For this reason we stipulate that $\surd$ cannot be an alternative in the choice construct:

$$(x{:}A \longrightarrow Px) \text{ is invalid if } \surd \in A.$$

$SKIP_A$ is defined as a process which does nothing but terminate successfully.

$$\alpha SKIP_A = A \cup \{\surd\}.$$

As usual, we shall frequently omit the subscript ~~alphabet.~~

X1. A vending machine that is intended to serve only one customer with chocolate or toffee:

$$\begin{aligned} VMONE = (coin \longrightarrow (&choc \longrightarrow SKIP \\ &| toffee \longrightarrow SKIP)) \end{aligned}$$

In designing a process to solve a complex task, it is frequently useful to split the task into two subtasks, one of which must be completed successfully before the other begins. If P and Q are sequential processes with the same alphabet, their sequential composition

$$P;Q$$

is a process which first behaves like P; but when P terminates successfully, (P;Q) continues by behaving like Q. If P never terminates successfully, neither does (P;Q).

X2. A vending machine designed to serve exactly two customers, one after the other.

$$VMTWO = VMONE;VMONE$$

A process which repeats similar actions as often as required is known as a loop; it can be defined as a special case of recursion.

$$*P = \mu X.(P;X)$$

$$= P;P;P;....$$

$$\alpha(*P) = \alpha P - \{\checkmark\}$$

Clearly such a loop is intended never to terminate successfully; that is why it is convenient to remove ✓ from its alphabet.

X3. A vending machine designed to serve any number of customers

$$VMCT = *VMONE$$

This is identical to 1.1.3.X3.

A sequence of symbols is said to be a <u>sentence</u> of a process P if P terminates successfully after engaging in the corresponding sequence of actions. The set of all such sentences is called the <u>language</u> accepted by P. Thus the notations introduced for describing sequential processes may also be used to define the kind of simple language which might be used for communication between a human being and a computer.

X4. A sentence of "pidgingol" consists of a noun clause followed by a predicate. A predicate is a verb followed by a noun clause. A verb is either "bites" or "scratches". The definition of a noun clause is given more formally below.

$$\alpha PIDGINGOL = \{a, the, cat, dog, bites, scratches\}$$

$$PIDGINGOL = NOUNCLAUSE;PREDICATE$$

$$PREDICATE = VERB;NOUNCLAUSE$$

$$VERB = (bites \longrightarrow SKIP \mid scratches \longrightarrow SKIP)$$

$$NOUNCLAUSE = ARTICLE;NOUN$$

$$ARTICLE = (a \longrightarrow SKIP \mid the \longrightarrow SKIP)$$

$$NOUN = (cat \longrightarrow SKIP \mid dog \longrightarrow SKIP)$$

An example sentence of pidgingol:

"the cat scratches a dog"

To describe languages with an unbounded number of sentences, it is necessary to use some kind of iteration or recursion.

X5. A noun clause which may contain any number of adjectives:

$$\text{NOUNCLAUSE} = \text{ARTICLE}; \mu X.(\text{furry} \longrightarrow X \mid \text{prize} \longrightarrow X \mid \text{cat} \longrightarrow \text{SKIP} \mid \text{dog} \longrightarrow \text{SKIP})$$

Example of a nounclause:

"the furry furry prize dog"

X6. A process which accepts any number of "a"s followed by a "b" and then the same number of "c"s.

$$A^nBC^n = \mu X.(b \longrightarrow \text{SKIP} \mid a \longrightarrow (X;(c \longrightarrow \text{SKIP})))$$

If a "b" is accepted first, the process terminates; no "a"s and no "c"s are accepted, so their numbers are tha same. If the second branch is taken, the accepted sentence starts with "a" and ends with "c", and between these is the sentence accepted by the "recursive call" on the process X. If we assume thet the recursive ~~call behaves~~ correctly, then so will ~~the non-~~recursive call on A^nBC^n.

X7. A process which first behaves like A^nBC^n, but then accepts a "d" followed by the same number of "e"s.

$$A^nBC^nDE^n = ((A^nBC^n);d \longrightarrow \text{SKIP}) \,\|\, C^nDE^n$$

where $C^nDE^n = f^{-1}(A^nBC^n)$

for f which maps "c" to "e", "d" to "b", and "e" to "c".

The notations for defining a language by means of an accepting process are as powerful as those of regular expressions. The use of recursion introduces some of the power of context free grammar, but not all. A process can only define those languages that can be parsed from left to right without backtracking or look-ahead. This is because the use of the choice operator requires that the first event of each alternative is different from all its other first events. Consequently, it is not possible to use the construction of X5 to define a nounclause in which the word "prize" can be either a noun or an adjective or both, e.g. "the prize dog", "the furry prize". However, the introduction of parallel composition makes the process notation more powerful than context-free grammars, which cannot define the language of X7.

X8. A process which accepts any interleaving of "down"s and "up"s, except that it terminates successfully on the first occasion that the number of "down"s exceeds the number of "up"s:

$$POS = (down \longrightarrow SKIP \mid up \longrightarrow (POS;POS))$$

If the first symbol is "down", the task of POS is immediately accomplished. But if the ~~first symbol is "up", it is then~~ necessary to accept two more "down"s than "up"s. The only way of achieving this is first to accept one more "down" than "up"; and then again to accept one more "down" than "up". Thus two successive recursive calls on POS are needed, one after the other.

X9. The process C_0 behaves like CT_0 (1.1.4.X2)

$$C_0 = (around \longrightarrow C_0 \mid up \longrightarrow C_1)$$

$$C_{n+1} = POS;C_n \qquad \text{for all } n \geqslant 0.$$

$$= \underbrace{POS;\ldots;POS;}_{n \text{ times}}POS;C_0$$

5.2 Laws

The laws for sequential composition are similar to those for catenation (1.6.1), with SKIP playing the role of the unit.

L1. $SKIP;P = P;SKIP = P$

L2. $(P;Q);R = P;(Q;R)$

L3. $(x:A \longrightarrow Px);Q = (x:A \longrightarrow ((Px);Q))$

The law for the choice operator has corollaries:

L4. $(a \longrightarrow P);Q = a \longrightarrow (P;Q)$

L5. $STOP;Q = STOP$

When sequential processes are composed in parallel, the combination terminates successfully just when both components do so

L6. $SKIP_A \| SKIP_B = SKIP_{A \cup B}$

A successfully terminating process participates in no other event offered by a concurrent partner.

L7. $((x:A \longrightarrow Px) \| SKIP_B) = (x:(A-B) \longrightarrow (Px \| SKIP_B))$

In a concurrent combination of a sequential with a nonsequential process, when does the combination terminate successfully? If the alphabet of the sequential process wholly contains that of its partner, termination of the partnership is determined by that of the sequential process, since the partner can do nothing when its partner has finished.

L8. $STOP_A \| SKIP_B = SKIP_B$ if $\checkmark \tilde{\in} A \wedge A \subseteq B$.

We shall avoid a number of problems by stipulating that all parallel combinations of sequential and nonsequential processes must conform to the alphabet constraint of L8.

The laws L1 to L3 may be used to prove the claim made in 5.1.X9 that C_0 behaves like CT_0. This is done by showing that C satisfies the set of guarded recursive equations used to define CT. The equation for CT_0 is the same as that for C_0:

$$C_0 = (around \longrightarrow C_0 \mid up \longrightarrow C_1) \qquad \text{~~def~~ } C_0$$

For $n > 0$, we need to prove

$$C_n = (up \longrightarrow C_{n+1} \mid down \longrightarrow C_{n-1})$$

Proof. LHS $= POS;C_{n-1}$ def C_n

$= (down \longrightarrow SKIP \mid up \longrightarrow POS;POS);C_{n-1}$ def POS

$= (down \longrightarrow (SKIP;C_{n-1}) \mid up \longrightarrow (POS;POS);C_{n-1})$ L3

$= (down \longrightarrow C_{n-1} \mid up \longrightarrow POS;(POS;C_{n-1}))$ L1,L2

$= (down \longrightarrow C_{n-1} \mid up \longrightarrow POS;C_n)$ def C_n

$=$ RHS def C_n

Since C obeys the same set of guarded recursion equations as CT, they are the same.

This proof has been written out in full, in order to illustrate the use of the laws, and also in order to allay suspicion of circularity. What seems most suspicious is that the proof does not use induction on n. In fact, any attempt to use induction on n will fail, because the very definition of CT_n contains the process CT_{n+1}. Fortunately, an appeal to the law of unique solutions is both simple and successful.

In order to preserve the validity of the law of unique solutions, it is necessary to state that

SKIP is not guarded

P;Q is guarded if P is.

5.3 Traces

The first and only action of the process SKIP is successful termination

$$\text{traces (SKIP)} = \{\langle\,\rangle, \langle\checkmark\rangle\}$$

To define sequential composition of processes, it is convenient first to define sequential composition of their individual traces. If s and t are traces and s does not contain ✓

$$(s;t) = s$$

$$(s^\frown\langle\checkmark\rangle);t = s^\frown t$$

(The ✓ at the end of s acts as "glue" to join s and t. In the absence of glue, t falls off.)

L1 $\quad \text{traces}(P;Q) = \{ s;t \mid s \in \text{traces}(P) \wedge t \in \text{traces}(Q) \}$

The whole purpose of the ✓ symbol is that it should terminate the trace in which it occurs

L2 $\quad s \in \text{traces}(P) \wedge s \text{ contains } \checkmark \implies \bar{s}_0 = \checkmark$

To preserve the validity of this law, it is essential to impose the restriction mentioned in L8:

$(P \,\|\, Q)$ is valid only if

$$\alpha P \subseteq \alpha Q \vee \alpha Q \subseteq \alpha P$$

$$\vee\ \checkmark \in \alpha P \cap \alpha Q \vee \checkmark \in \overline{\alpha P} \wedge \overline{\alpha Q}$$

5.4 Sequential Programs

In this section we shall introduce the most important aspects of conventional sequential programming, namely assignments, conditionals, and loops. To simplify the formulation of useful laws, some unusual notations will be introduced.

The essential feature of conventional computer programming is assignment. If x is a program variable and e is an expression and P a process

$$(x:=e;P)$$

is a process which behaves like P, except that the initial value of x is defined to be the initial value of the expression e. Assignment by itself can be defined:

$$(x:=e) = (x:=e;SKIP)$$

Single assignment generalises easily to multiple assignment. Let x stand for a list of distinct variables

$$x = x_o, x_1, \ldots, x_{n-1} .$$

Let e stand for a list of expressions

$$e = e_o, e_1, \ldots, e_{n-1} .$$

Provided that the lengths of the two lists are the same

$$x:=e$$

assigns the initial value of e_i to x_i, for all i. Note that <u>all</u> the e_i are evaluated before <u>any</u> of the assignments are made, so that if y occurs in the expression g

$$y:=f;z:=g$$

is quite different from

$$y,z:=f,g .$$

Let b be an expression that evaluates to a Boolean truthvalue (either true or false). If P and Q are processes,

$$P \not\mathrel{\triangleleft} b \not\mathrel{\triangleright} Q \qquad\qquad (P \; \underline{\text{if}} \; b \; \underline{\text{else}} \; Q)$$

is a process which behaves like P if the initial value of b is true, or like Q if the initial value of b is false. The notation is novel, but less cumbersome than the traditional

> **if** b **then** P **else** Q.

For similar reasons, the traditional loop

> **while** b **do** Q

will be written

> $b{*}Q$

This may be defined by recursion:

$$b{*}Q \;=\; \mu X.(Q;X) \lhd b \rhd SKIP$$

X1. A process that behaves like CT_n (1.1.4.X2)

$$\mu X.(around \rightarrow X \mid up \rightarrow (n:=1;X))$$
$$\lhd n = 0 \rhd$$
$$(up \rightarrow (n:=n+1;X) \mid down \rightarrow (n:=n-1;X))$$

X2. A process that behaves like CT_0

> $n:=0;X1$

X3. A process that behaves like POS (5.1.X8)

$$n:=1;(n > 0){*}(up \rightarrow n:=n+1$$
$$\mid down \rightarrow n:=n-1)$$

X4. A process which divides a natural number x by a positive number y, assigning the quotient to q and the remainder to r

$$QUOT \;=\; (q:=x \div y;\; r:=x-q\times y)$$

X5. A process with the same effect as X4, which computes the quotient by repeated subtraction:

$$LONGQUOT = (q:=0;r:=x;((r \geq y){*}(q:=q+1;r:=r-y)))$$

5.5 Laws

In the laws for assignment x and y stand for lists of distinct variables; e, f(x), f(e) stand for lists of expressions, possibly containing occurrences of variables in x or y; and f(e) contains e_i wherever f(x) contains x_i for all indices i.

L1. $(x:=x) \;=\; \mathrm{SKIP}$

L2. $(x:=e;\; x:=f(x)) \;=\; (x:=f(e))$

L3. If x,y is a list of distinct variables

$$(x:=e;\; y:=f(x)) \;=\; (x,y:= e,f(e))$$

When $\triangleleft b \triangleright$ is considered as a binary infix operator, it possesses several familiar algebraic properties.

L4-6. $\triangleleft b \triangleright$ is idempotent, associative, and distributes through $\triangleleft c \triangleright$

L7. $P \triangleleft \mathrm{true} \triangleright Q \;=\; P$

L8. $P \triangleleft \mathrm{false} \triangleright Q \;=\; Q$

L9. $P \triangleleft \neg b \triangleright Q \;=\; Q \triangleleft b \triangleright P$

L10. $P \triangleleft b \triangleright (Q \triangleleft b \triangleright R) \;=\; P \triangleleft b \triangleright R$

L11. $P \triangleleft a \triangleleft b \triangleright c \triangleright Q \;=\; (P \triangleleft a \triangleright Q) \triangleleft b \triangleright (P \triangleleft c \triangleright Q)$

L12. $x:=e;(P \triangleleft b(x) \triangleright Q)$

$$= (x:=e;P) \triangleleft b(e) \triangleright (x:=e;Q)$$

To deal effectively with assignment in concurrent processes, it is necessary to impose a restriction that no variable assigned in one concurrent process can ever be used in another. To enforce this restriction, we introduce two new categories of variable into the alphabets of sequential processes.

var(P) : the set of variables that may be assigned within P

val(P) : the set of variables that may be used in expressions within P.

These are related by inclusion:

$$\mathrm{var}(P) \subseteq \mathrm{val}(P) \subseteq \alpha P$$

Similarly, we define val(e) as the set of variables appearing in the expression e.

Now if P and Q are to be joined by $||$, we stipulate that

$$\mathrm{var}(P) \cap \mathrm{val}(Q) \;=\; \mathrm{var}(Q) \cap \mathrm{val}(P) = \emptyset$$

Under this condition, it does not matter whether an assignment takes place before or after a parallel split.

L13. $(x:=e;P) \| Q = (x:=e;(P \| Q))$

provided that $x \subseteq var(P) - val(Q)$

and $val(e) \cap var(Q) = \emptyset$

An immediate consequence of this is

$$(x:=e;P) \| (y:=f;Q) = (x,y:=e,f;(P \| Q))$$

provided that $x \subseteq var(P) - val(Q) - val(f)$

and $y \subseteq var(Q) - val(P) - val(e)$.

This shows how the alphabet restriction rules ensure that assignments within one component process of a concurrent pair cannot interfere with assignments within the other. In an implementation, sequences of assignments may be carried out either together or in any interleaving, without making any difference to the externally observable actions of the process (except possibly to improve their timing — but we have chosen to ignore such details).

Finally, concurrent combination distributes through the conditional:

L14. $P \| (Q \not< b \not> R) = (P \| Q) \not< b \not> (P \| R)$

provided $val(b) \cap var(P) = \emptyset$

5.6 Specifications

A specification of a sequential process must describe not only the traces of the events which occur, but also the relationship between these traces, the initial values of the program variables, and their final values. To denote the initial value of a program variable x, we simply use the variable name x by itself. To denote the final value, we decorate the name with a superscript $\checkmark$, as in $x^{\checkmark}$. The value of $x^{\checkmark}$ is not defined until the process is terminated, i.e., until the last event of the trace is $\checkmark$.

X1. A process which performs no action, but adds one to the value of x, and terminates successfully with the value of y unchanged:

$$tr = \langle\rangle \vee tr = \langle\checkmark\rangle \wedge x^{\checkmark} = x+1 \wedge y^{\checkmark} = y$$

X2. A process which performs an event whose symbol is the initial value of the variable x, and then terminates successfully, leaving the final values of x and y unchanged:

$$tr = \langle\rangle \vee tr = \langle x\rangle \vee (tr = \langle x, \checkmark\rangle \wedge x^{\checkmark} = x \wedge y^{\checkmark} = y)$$

X3. A process which stores the identity of its first event as the final value of x:

$$\# tr \leq 2 \wedge (\# tr = 2 \Longrightarrow (tr = \langle x^{\checkmark}, \checkmark\rangle \wedge y^{\checkmark} = y))$$

The correct working of a process often depends on some precondition S(x) on the initial values of the program variables x. This can be expressed by writing S(x) as the antecedent of the specification.

X4. A process which divides a nonnegative x by a positive y, and assigns the quotient to q and the remainder to r:

$$DIV = \quad y > 0 \Longrightarrow$$
$$(tr = \langle\rangle \vee (tr = \langle\checkmark\rangle \wedge q^{\checkmark} = x \div y \wedge r^{\checkmark} = x - q^{\checkmark} \times y \wedge y^{\checkmark} = y \wedge x^{\checkmark} = x)$$

Without the precondition, this specification would be impossible to meet in its full generality.

X5. Here are some more complex specifications which will be used later

$$DIVLOOP = (tr = \langle\rangle \vee (tr = \langle\checkmark\rangle \wedge r = (q^{\checkmark} - q) \times y + r^{\checkmark}$$
$$\wedge r^{\checkmark} < y \wedge x^{\checkmark} = x \wedge y^{\checkmark} = y))$$

$$T(n) = r < n \times y$$

All variables in this ~~specification are~~ intended to denote natural numbers, so subtraction is undefined if the second operand is greater than the first.

We shall now formulate the laws which underlie proofs that a process satisfies its specification. Let $S(x, tr, x^{\checkmark})$ be a specification. In order to prove that SKIP satisfies this specification, clearly the specification must be true when the trace is empty; furthermore, it must be true when the trace is $\langle\checkmark\rangle$ and the final values of all variables $x^{\checkmark}$ are equal to their initial values. These two conditions are also sufficient, as stated in the following law.

L1. If $S(x, \langle\rangle, x^{\checkmark})$

and $S(x, \langle\checkmark\rangle, x)$

then SKIP <u>sat</u> $S(x, tr, x^{\checkmark})$.

X6. The strongest specification satisfied by SKIP is

$$SKIP_A \ \underline{sat} (tr = \langle\rangle \ \vee \ tr = \langle\checkmark\rangle \wedge x^{\checkmark} = x)$$

where x is a list of all variables in A and $x^{\checkmark}$ is a list of their ticked variants.

X7. SKIP $\underline{sat}$ $(r < y \Rightarrow (T(n+1) \Longrightarrow DIVLOOP))$

(1) Replacing tr by $\langle\rangle$ in the specification gives

$$r < y \wedge T(n+1) \Longrightarrow \langle\rangle = \langle\rangle \vee \ldots\ldots$$

(2) Replacing tr by $\langle\checkmark\rangle$ and final values by initial values gives

$$r < y \wedge T(n+1) \Longrightarrow (\langle\checkmark\rangle = \langle\rangle \ \vee (\langle\checkmark\rangle = \langle\checkmark\rangle \ \wedge x = x \ \wedge y = y$$
$$\wedge \ r = (a-q) \times y + r \wedge r < y))$$

Both of ~~these are tedious tautologies.~~

If e is a list of expressions, we define $\mathfrak{D}$ e as a predicate stating that the values of all operands of e are within the domains of their operators. For example, in natural number arithmetic

$$\mathfrak{D}(x \div y) \ = \ y > 0.$$
$$\mathfrak{D}(y+1, \ z+y) = true$$
$$\mathfrak{D}(r - y) \ = \ y \leqslant r$$

It is a precondition of successful assignment (x:=e) that the expressions (e) on the right hand side must be defined. In this case, if P satisfies a specification S(x),(x:=e;P) satisfies the same specification, after it has been modified to reflect the fact that the initial value of x is e.

L2. If P $\underline{sat}$ S(x)

then (x:=e;P) $\underline{sat}$ $(\mathfrak{D}e \Longrightarrow S(e))$

The law for simple assignment can be derived from this on replacing P by SKIP, and using X6:

L2' x_0:=e $\underline{sat}$ $(\mathfrak{D}e \wedge tr \neq \langle\rangle \Longrightarrow tr = \langle\checkmark\rangle \wedge x_0^{\checkmark} = e \wedge x_1^{\checkmark} = x_1 \wedge \ldots.)$

X8. $\text{SKIP} \ \underline{\text{sat}} \ (tr \neq \langle\rangle \Longrightarrow tr = \langle\checkmark\rangle \wedge q^{\checkmark} = q \wedge r^{\checkmark} = r \wedge y^{\checkmark} = y \wedge x^{\checkmark} = x)$

$\therefore \ (r := x - q \times y; \text{SKIP}) \ \underline{\text{sat}} \ (x \geqslant q \times y \wedge tr \neq \langle\rangle \Longrightarrow$

$tr = \langle\checkmark\rangle \wedge q^{\checkmark} = q \wedge r^{\checkmark} = x - q \times y \wedge y^{\checkmark} = y \wedge x^{\checkmark} = x)$

$\therefore \ (q := x \div y; r := x - q \times y) \ \underline{\text{sat}} \ (y > 0 \wedge x \geqslant (x \div y) \times y \wedge tr \neq \langle\rangle \Longrightarrow$

$tr = \langle\checkmark\rangle \wedge q^{\checkmark} = x \div y \wedge r^{\checkmark} = x - q \times y \wedge y^{\checkmark} = y \wedge x^{\checkmark} = x)$

The specification on the last line is equivalent to DIV which was defined in X4 .

X9. Assume X $\underline{\text{sat}} \ (T(n) \Longrightarrow \text{DIVLOOP})$

$\therefore \ (r := r - y; X) \ \underline{\text{sat}} \ (y \leqslant r \Longrightarrow (r - y < n \times y \Longrightarrow (tr = \langle\rangle$

$\vee \ tr = \langle\checkmark\rangle \wedge (r - y) = \ldots)))$

$\therefore \ (q := q + 1; r := r - y; X) \ \underline{\text{sat}} (y \leqslant r \Longrightarrow (r < (n+1) \times y \Longrightarrow \text{DIVLOOP}'))$

where $\text{DIVLOOP}' = (tr = \langle\rangle \vee (tr = \langle\checkmark\rangle \wedge (r - y) = (q^{\checkmark} - (q+1)) \times y + r^{\checkmark}$

$\wedge \ r^{\checkmark} < y \wedge x^{\checkmark} = x \wedge y^{\checkmark} = y))$

By elementary algebra of natural numbers

$y \leqslant r \Longrightarrow (\text{DIVLOOP}' \equiv \text{DIVLOOP})$

$\therefore \ (q := q + 1; r := r - y; X) \ \underline{\text{sat}} \ (y \leqslant r \Longrightarrow (T(n+1) \Longrightarrow \text{DIVLOOP}))$

For general sequential composition, a much more complicated law is required, in which the traces of the components are sequentially composed, and the initial state of the second component is identical to the final state of the first component. ~~However,~~ the values of the variables in this intermediate state are not observable; only the existence of such values is assured.

L3. If P $\underline{\text{sat}}$ $S(x, tr, x^{\checkmark})$

and Q $\underline{\text{sat}}$ $T(x, tr, x^{\checkmark})$

then $(P;Q) \ \underline{\text{sat}} \ (\exists y, s, t. \ tr = (s;t)$

$\wedge \ S(x, s, y)$

$\wedge \ T(y, t, x^{\checkmark}))$

In this law, x is a list of all variables in the alphabet of P and Q; $x^{\checkmark}$ is a list of their superscripted variants, and y a list of the same number of fresh variables.

The specification of a conditional is the same as that of the first component if the condition is true, and the same as that of the second component if false.

L4. If P sat S and Q sat T

then $(P \triangleleft b \triangleright Q)$ sat $(b \wedge S \vee \neg b \wedge T)$

An alternative form of this law is sometimes more convenient

L4' If P sat $(b \Longrightarrow S)$ and Q sat $(\neg b \Longrightarrow S)$

then $(P \triangleleft b \triangleright Q)$ sat S.

X10. Let COND = $(q{:}=q+1;r{:}=r-y;X) \triangleleft r \geqslant y \triangleright$ SKIP

and X sat $(T(n) \Longrightarrow$ DIVLOOP).

Then COND sat $(T(n+1) \Longrightarrow$ DIVLOOP).

The two sufficient conditions for this conclusion have been proved in X7 and X9 .

The proof of a loop uses the recursive definition given in 5.4 and the law for recursion (1.10.2 L6). If R is the intended specification of the loop, we must find a specification S(n) such that S(0) is always true, and also

$$(\forall n.\ S(n)) \Longrightarrow R.$$

A general method to construct S(n) is to find a predicate T(n,x), which describes the conditions on the initial state x such that the loop is certain to terminate in less than n repetitions. Then define:

$$S(n) = (T(n,x) \Longrightarrow R)$$

Clearly, no loop can terminate in less than no repetitions, so if T(n,x) is correctly defined T(0,x) must be false, and S(0) must be true.

The result of the proof of the loop will be $\forall n.\ S(n)$, i.e.,

$$\forall n.\ (T(n,x) \Longrightarrow R)$$

Since n is chosen as a variable which does not occur in R, this is equivalent to

$$(\exists n.\ T(n,x)) \Longrightarrow R.$$

No stronger specification can possibly be met, since $\exists n.T(n,x)$ is the precondition under which the loop terminates in some finite number of iterations.

Finally, we must prove that the body of the loop meets its specification. Since the recursive equation for a loop involves a conditional, this task splits into two. Thus we derive the general law:

L5. If $\neg T(0,x)$

and SKIP sat $(\neg b \Longrightarrow (T(n,x) \Longrightarrow R))$

and X sat $(T(n,x) \Longrightarrow R) \Longrightarrow ((Q;X)$ sat $(b \Longrightarrow (T(n+1,x) \Longrightarrow R)))$.

then $(b*Q)$ sat $((\exists n.T(n,x)) \Longrightarrow R)$

X11. We wish to prove that the program for long division by repeated subtraction (5.4.X5) meets its specification DIV. The task splits naturally in two. The second and more difficult part is to prove that the loop meets some suitably formulated specification, namely

$(r \geq y) * (q:=q+1;r:=r-y)$ sat $(y > 0 \Longrightarrow \text{DIVLOOP})$

First we need to formulate the condition under which the loop terminates in less than n iterations.

$T(n) = r < n \times y$

here $T(0)$ is obviously false; the clause

$\exists n.\ T(n)$

is equivalent to

$y > 0$

which is the precondition under which the loop terminates. The remaining steps of the proof of the loop have already been taken in X7 and X9. The rest of the proof is a simple exercise.

5.7 Implementation

The initial and final states of a sequential process can be represented as a function which maps each variable name onto its value. A sequential process is defined as a function which maps its initial state onto its subsequent behaviour. Successful termination is represented by the atom "SUCCESS. A process which is ready to terminate will accept this symbol, which it maps, not onto another process, but onto the final state of its variables.

The process SKIP takes an initial state as a parameter, accepts "SUCCESS as its only action, and delivers its initial state as its final state.

```
SKIP  =  λs.   λy. if y ≠ "SUCCESS then "BLEEP else  s
```

An assignment is similar, except that its final state is slightly changed.

```
      assign(x,e)  =  λs,   λy. if y ≠ "SUCCESS then "BLEEP
                            else update (s,x,e)
where  update(s,x,e)  =  λy. if y = x then eval (e,s)
                                       else s(y)
```

and eval (e,s) is the result of evaluating the expression e in state s.

Here, for simplicity we have implemented only the single assignment. Multiple assignment is a little more complicated.

To implement sequential composition, it is necessary first to test whether the first operand is successfully terminated. If so, its final state is passed on to the second operend. If not, the first action is that of the first operand

```
sequence (P;Q)  =   λs.
if P(s) ("SUCCESS) ≠ "BLEEP then
          Q(P(s)("SUCCESS))
else  λy. if P(s)(y) = "BLEEP then "BLEEP
             else sequence (P(s)(y), Q)
```

The implementation of the conditional is as a conditional:

```
condition (P,b,Q) =  λs. if  eval(b,s) then P(s) else Q(s)
```

The implementation of the loop is left as an exercise.

CHAPTER SIX

SHARED RESOURCES

6.1 Introduction

In chapter 4.5 we introduced the concept of a subordinate process, whose sole task is to meet the needs of a single main process; and for this we have defined the notation

$(m:R \,//\, S)$

Suppose now that S contains or consists of two concurrent processes $(P \| Q)$, and both P and Q require the services of the same subordinate process (m:R). Unfortunately, it is not possible for P and Q both to communicate with (m:R) along the same channels, because these channels would have to be in the alphabet of both P and Q; and then the definition of $\|$ would require that communications with (m:R) take place only when both P and Q communicate the same message simultaneously — which is far from the required effect. What is needed is some way of interleaving the communications between P and (m:R) with those between Q and (m:R). In this way (m:R) serves as a resource shared between P and Q; each of them uses it at a time when the other is not doing so.

When the identity of all the sharing processes is known in advance, it is possible to arrange that each ~~such process uses~~ a different ~~set of~~ channels to communicate with the shared resource. This technique was used in the story of the dining philosophers: each fork was shared among two neighbouring philosophers, and the footman was shared among all five. Another example was 4.5 X6, in which a buffer was shared between two processes, one of which used only the left channel and the other used only the right channel. But this method is not adequate for a subordinate process intended to serve the needs of a main process which splits into an arbitrary number of concurrent subprocesses. This chapter introduces techniques for sharing a resource among many processes, even when their number and identities is not known in advance. It is illustrated by examples drawn from the design of an operating system.

5.2 Sharing by interleaving

The problem described in 5.1 arises from the use of the combinator $\|$ to describe the concurrent behaviour of processes; and this problem can often be avoided by using instead the interleaving form of concurrency (P $|||$ Q). Here, P and Q have the same alphabet and their communications with external (shared) processes are arbitrarily interleaved. Of course, this prohibits direct communication between P and Q; but indirect communication can be re-established through the services of a shared subordinate process of appropriate design.

X1 Shared Subroutine

doub:DOUBLE // (P $|||$ Q)

Here, both ~~P and Q may contain calls on the~~ subordinate process

(doub.left!v ; doub.right?x)

Even though these pairs of communications from P and Q are arbitrarily interleaved, there is no danger that one of the processes will accidentally obtain an answer which should have been received by the other. To ensure this, all subprocesses of the main process must observe a strict alternation of communications on the left channel with communications on the right channel of the shared subordinate process. For this reason, it seems worthwhile to introduce a specialised notation, whose exclusive use will guarantee observance of the required discipline. I suggest a notation reminiscent of a traditional procedure call in a high level language, except that the value parameters are preceded by ! and the result parameters by ?, thus

doub!x?y =

(doub.left!x ; doub.right?y)

X2 Shared data structure

In an airline flight reservation system, bookings are made by many reservation clerks, whose actions are interleaved. Each reservation adds a passenger to the flight list, and returns an indication whether that passenger was already booked or not. For this oversimplified example, the set implemented in 4.5 X8 will serve as a shared subordinate process, named by the flight number:

AG109:SET // (.... (CLERK $|||$ CLERK $|||$ ....) ...)

Each CLERK books a passenger by the call

AG109!pass no?x

which stands for

(AG109.left! pass no ;AG109.right?x)

In these two examples, each occasion of use of the shared resource involves exactly two communications, one to send the parameters and the other to receive the results; after each pair of communications, the subordinate process returns to a state in which it is ready to serve another process, or the same one again. But frequently we wish to ensure that a whole series of communications take place between two processes, without danger of interference by a third process. For example, a single expensive output device may have to be shared among many concurrent processes. On each occasion of use, a number of lines constituting a file must be output consecutively, without any danger of interleaving of lines sent by another process. For this purpose, the output of each file must be preceded by an "acquire" which obtains exclusive use of the resource; and on completion, the resource must be made available again by a "release".

X3 Shared line printer

$$
\begin{aligned}
LP \;=\; & \text{acquire} \rightarrow \\
& \quad \mu X.(\text{left?}l \rightarrow h!l \rightarrow X \\
& \qquad \mid \text{release} \rightarrow LP)
\end{aligned}
$$

Here, h is the channel which connects LP to the hardware of the line printer. After acquisition, the process LP copies successive lines from its left channel to its hardware, until a release signal returns it to its original state, in which it is available for use by other processes. This process is used as a shared resource

$lp{:}LP \,//\, \ldots\, (P \;|||\; Q)\, \ldots$

Inside P or Q, the output of the series of lines constituting a file is bracketed by an "lp.acquire" and "lp.release":

$lp.acquire \rightarrow \ldots\; lp.left!\text{"A.JONES"} \ldots$

$\qquad lp.left!nextline \rightarrow \ldots\;\; lp.release$

X4 An improvement on X3

When a line printer is shared between many users, the content of each file will be manually separated after output from the previous and the following files. For this purpose, the printing paper is usually divided into pages, which are separated by perforations; and the hardware of the printer allows an operation "throw", which moves the paper rapidly to the end of the current page - or better, to the next outward-facing fold in the paper stack. To assist in separation of output, files should begin and end on page boundaries, and a complete row of asterisks should be printed at the end of the last page of the file, and at the beginning of the first page. To prevent confusion, no complete line of asterisks is permitted to be printed in the middle of a file:

```
LP = (h!throw → h!asterisks →
        acquire → h!asterisks →
        μX.(left?l → if l ≠ asterisks then (h!l → X)
                                       else X
           |release → LP)
```

This version of LP is used in exactly the same way as the previous one.

The use of the signals "acquire" and "release" prevent arbitrary interleaving of lines from distinct files, without introducing the danger of deadlock. But if more than one resource is to be shared in this fashion, the risk of deadlock cannot be ignored.

X5 Deadlock

Ann and Mary are good but impecunious cooks; they share a pot and a pan, which they acquire, use and release as they need them.

UTENSIL = (acquire ⟶ use ⟶ use ⟶ ... ⟶ release ⟶ UTENSIL)

pot:UTENSIL // pan:UTENSIL // (ANN ||| MARY)

Ann cooks in accordance with a recipe which requires a pot first and then a pan, whereas Mary needs a pan first, then a pot

ANN = ... pot.acquire; ... pan.acquire; ...

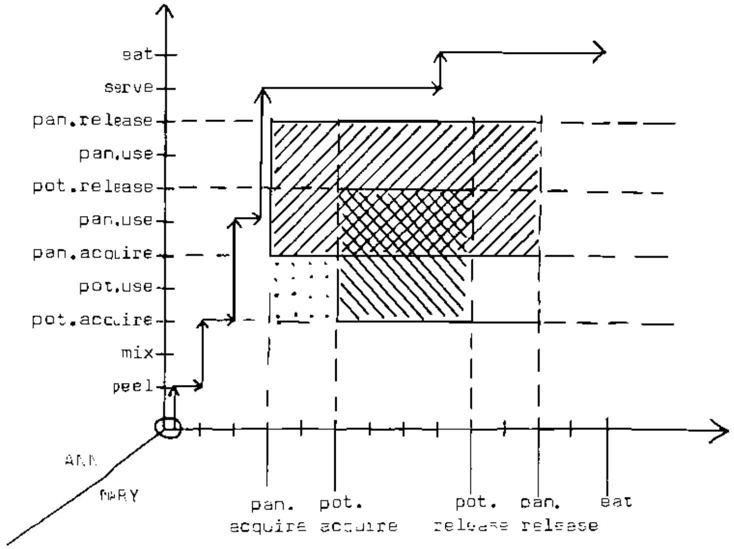
MARY = ... pan.acquire; ... pot.acquire; ...

Unfortunately, they decide to prepare a meal at about the same time. Each of them acquires her first utensil; but when she needs her second utensil, she finds she cannot have it, because it is being used by the other.

The story of Ann and Mary can be visualised on a two-dimensional plot, where the life of Ann is displayed along the vertical axis, and Mary's life on the horizontal. The system ~~starts in the~~ bottom left hand corner, at the beginning of both their lives. Each time Ann performs an action, the system moves one step upward. Each time Mary performs an action, the system moves one step right. The trajectory shown on the graph shows a typical interleaving of Ann and Mary's actions. Fortunately, this trajectory reaches the top right hand corner of the graph where both cooks are enjoying their meal.

But this happy outcome is not certain. Because they cannot simultaneously use a shared utensil, there are certain rectangular regions in the state space through which the trajectory cannot pass. For example in the region hatched ▨ both cooks would be using the pan, and this is not possible. Similarly, exclusion on the use of the pot prohibits entry into the region hatched ▧ . Thus if the trajectory reaches the edge of one of these forbidden regions, it can only follow the edge upward (for a vertical edge) or rightward (for a horizontal edge). During this period, one of the cooks is waiting for release of a utensil by the other.

Now consider the zone marked with dots. ⁙ If ever the trajectory enters this zone, it will inevitably end in deadlock at the top right hand corner of the zone. The purpose of the picture is to show that the danger of deadlock arises solely as a result of a re-entrancy of the forbidden region which faces towards the origin - other reentrancies are quite safe. The picture also shows that the only sure way of preventing deadlock is to extend the forbidden region to cover the danger zone, and so remove the reentrancy. One technique would be to introduce an additional resource (say the stove), which must be acquired before either utensil, and must not be ~~released until both utensils have been released.~~ This solution is similar to the one imposed by the footman in the story of the dining philosophers (2.5.3). An easier solution is to insist that any cook who is going to want both utensils must acquire the pan first (example due to E.W. Dijkstra).



The solution of the previous example generalises to any number of users and any number of resources. Provided there is a fixed order in which all users acquire the resources they want, there is no risk of deadlock. Users should release their resources as soon as they have finished with them; the order of release does not matter. Users may even acquire resources out of order, provided that at time of acquisition they have already released all resources which are later in the standard ordering. Observance of this discipline of resource acquisition and release can often be checked by a visual scan of the text of the user processes.

To formulate the relevant theorem more accurately, we let

a_j stand for the acquisition of the resource with rank j in the standard ordering

r_j stand for release of this resource.

Clearly, each resource must be acquired and released in alternation

$$\text{ALTERNATION} = \forall j.\ 0 \leqslant tr.a_j - tr.r_j \leqslant 1.$$

Furthermore a resource may be acquired only if it is ranked higher than any resource currently acquired but not yet released:

$$\text{ORDERED} = \forall i \quad \overline{tr}_0 = a_i \Longrightarrow$$
$$i > \max \left\{ j \mid \overline{tr}'.a_j - \overline{tr}'.r_j = 1 \right\}$$

A user process must satisfy both these specifications:

USER_i <u>sat</u> ALTERNATION $\wedge$ ORDERED

The behaviour of each resource for all $i \leqslant n$ is a strict alternation of acquisition and release, so their concurrent composition may be specified:

RESOURCES <u>sat</u> ALTERNATION

The system as a whole is defined

$$\text{SYSTEM} = \text{RESOURCES} \parallel \text{ALLUSERS}$$

where $\text{ALLUSERS} = (\text{USER}_0 \,|||\, \text{USER}_1 \,|||\, \ldots \,|||\, \text{USER}_n)$

L1 Under the conditions defined above, if for all i $(FORECAST. \parallel USER_i)$ never stops, then neither does the system.

5.3 Shared storage

The purpose of this section is to argue against the use of shared storage; the section may be omitted.

The behaviour of systems of concurrent processes can readily be implemented on a single conventional stored program computer, by a technique known as timesharing, by which a single processor executes each of the processes in alternation, with process change on occurrence of interrupt from an external device or a regular timer. In this implementation, it is very easy to allow the concurrent processes to share locations of common storage, which are accessed and assigned simply by means of the usual machine instructions within the code for each of the processes.

A location of shared storage can be modelled in our theory as a shared variable (4.2.X7) with the appropriate symbolic name

$$(count{:}VAR \mathbin{//} (count!0 \longrightarrow (P \parallel Q)))$$

Shared storage must be clearly distinguished from the local storage described in 5.4. The simplicity of the laws for reasoning about sequential processes derives solely from the fact that each variable is updated by at most one process; and these laws do not deal with the many dangers that arise from arbitrary interleaving of assignments from different processes.

These dangers are most clearly illustrated by the following example.

X1 Interference

The shared variable "count" is used to keep a count of the total number of occurrences of some important event. On each occurrence of the event, the relevant process P or Q attempts to update the count by the pair of communications

$count.right?x \,;\, count.left!(x + 1)$

unfortunately, these two communications may be interleaved by a similar pair of communications from the other process, resulting in the sequence:

```
          count.right?x ; count.right?y ;
          count.left!(y + 1) ; count.left!(x + 1)
```

As a consequence, the value of the count is incremented only by one instead of two. This kind of error is known as interference, and it is an easy mistake in the design of processes which share common storage. Further, the actual occurrence of the fault is highly non-deterministic; it is not reliably reproducible, and so it is almost impossible to diagnose the error by conventional testing techniques. As a result, there are several operating systems in common use, which regularly produce slightly inaccurate summaries, statistics, and accounts.

A possible solution to this problem is to make sure that no change of process takes place during a sequence of actions which must be protected from interleaving. Such a sequence is known as a critical region. On an implementation by a single processor, the required exclusion is often achieved by inhibiting all interrupts for the duration of the critical region. This solution has an undesirable effect in delaying response to interrupts; and worse, it fails completely as soon as a second processing unit is added to the computer.

A better solution was suggested by E.W. Dijkstra in his introduction of the binary exclusion semaphore. A semaphore may be described as a process which engages alternately in actions named P and V.

$$SEM = (P \longrightarrow V \longrightarrow SEM)$$

This is declared as a shared resource

(mutex: SEM // ...)

Each process, on entry into a critical region, must send the signal

mutex.P

and on exit from the critical region must engage in the event

mutex.V

Thus, the critical region in which the count is incremented will appear:

```
    mutex.P ;
    count.right?x ; count.left!(x + 1) ;
    mutex.V
```

Provided that all processes observe this discipline, it is impossible for two processes to interfere with each other's updating of the count. But if any process omits a P or a V, or gets them in the wrong order, the effect will be chaotic, and will risk a disastrous or (perhaps worse) a subtle error.

A much more robust way to prevent interference is build the required protection into the very design of the shared storage, taking advantage of knowledge of the intended pattern of usage. For example, if a variable is to be used only for counting, then the operation which increments it should be a single atomic operation

count.up

and the shared resource should be designed like CT_0 (1.1.4 X2)

$$count{:}CT_0 \mathbin{//} (\ldots\ P \mathbin{|||} Q\ \ldots)$$

In fact there are good reasons for recommending that each shared resource be specially designed for its purpose, and that pure storage should never be shared in the design of a system using concurrency. This not only avoids the grave dangers of accidental interference; it also produces a design that can be implemented readily on networks of distributed processing elements as well as single-processor and multiprocessor computers with physically shared store.

6.4 Multiple resources

In the previous section, we described how a number of concurrent processes with different behaviour could share a single subordinate process. Each sharing process observes a discipline of alternating output and input, or alternating acquire and release signals, to ensure that at any given time the resource is used by at most one of the potentially sharing processes. Such resources are known as "serially reusable". In this section we introduce arrays of processes to represent multiple resources with identical behaviour; and indices in the array ensure that each element communicates safely with the process that has acquired it.

X1 Reentrant Subroutine

A shared subroutine that is serially reusable can be used by only one calling process at a time. If the execution of the subroutine requires a considerable calculation, there could be corresponding delays to the calling proceeses. If several processors are available to perform the calculations, there is good reason to allow several instances of the subroutine to proceed concurrently on different processors. A subroutine capable of several concurrent instances is known as "re-entrant"

$$\text{doub}:(\|_{i<27}(i:\text{DOUBLE}))\ /\!/\ \ldots$$

A typical call of this subroutine could be

doub.3.left!30; doub.3.right?y

The use of the index 3 ensures that the result of the call is obtained from the same instance of doub to which the arguments were sent, even though some other concurrent process may at the same time call another instance of the array, resulting in an interleaving of the mesaages:

doub.3.left.30, .. doub.2.left.20, ..

doub.3.right.60, .. doub.2.right.40, ..

When a process calls a reentrant subroutine, it really does not matter which element of the array responds to the call; any one that happens to be frea will be equally good. So rather than specifying a particular index 2 or 3, a calling process should leave the selection arbitrary, by using the construct

$$\square_{i\geqslant 0}(\text{doub}.i.\text{left}!30;\ \text{doub}.i.\text{right}?y)$$

This still observes the essential discipline that the same index is used for sending the arguments and receiving the result.

In the example shown above, there is an arbitrary limit of twenty seven simultaneous activations of the subroutine. Since it is fairly easy to arrange that a single processor can divide its attention among a much larger number of processes, it is more elegant to avoid such arbitrary limits, and declare an infinite array of concurrent processes

$$doub: \left(\|_{i \geq 0}\ i{:}D \right)$$

where D can now be designed to serve only a single call and then stop:

$$D \;=\; left?x \longrightarrow right!(x+x) \longrightarrow STOP$$

A subroutine with no bound on its reentrancy is known as a procedure.

The intention in using a procedure is that the effect of each call

$$\square_{i \geq 0} (doub.i.left!x\,;\ doub.i.right?y)$$

should be identical to the call of a subordinate process D declared right next to the call:

$$(doub{:}\ D\, //\, (doub.left!x;\ doub.right?y))$$

This latter is known as a "local" procedure call, since it models execution of the procedure on the same processor as the calling process; whereas the call of a shared procedure is known as a "remote" call, since it models execution on a separate possibly distant processor. Since the effect of remote and local calls is intended to be the same, the reasons for using the remote call can only be political or economic – e.g., to keep the code of the procedure secret, or to run it on a machine with special facilities which are too expensive to provide on the machines on which using processes run.

A typical example of an expensive facility is a high-volume backing store, such as a disc or bubble memory.

X2 Shared backing storage

A storage medium is split into B sectors which can be read and written independently. Each sector can store one block of information which it inputs on the left and outputs on the right. Unfortunately the storage medium is implemented in a technology with destructive read-out, so that each block written can be read only once. Thus each sector behaves like COPY rather than VAR. The whole store is an array of such sectors, indexed by numbers less than B:

$$BSTORE = \underset{i<B}{\|} i:COPY$$

This store is intended for use as a subordinate process

(back:BSTORE // ...)

within its scope, the store may be used:

back.i.left!bl ... back.i.right?y ...

The backing store may also be shared by concurrent processes. In this case, the action

$$\underset{i<B}{[]} \text{back.i.left!bl}$$

will simultaneously acquire an arbitrary free sector with number i, and write the value of bl into it. Similarly, back.i.right?x will simultaneously read the content of sector i into x and release this sector for use on another occasion, very possibly by another process.

X3 Two line printers

Two identical line printers are available to serve the needs of a collection of using processes. They both need the kind of protection from interleaving that was provided by LP (6.2.X4). We therefore declare an array of two instances of LP, ~~each of which is~~ ~~indexed by a natural~~ number indicating its position in the array

LP2 = (0:LP || 1:LP).

This array may itself be given a name for use as a shared resource

(lp:LP2 // ...)

Each instance of LP is now prefixed twice, by a name and by an index; thus communications with the using process take the form:

lp.0.acquire, lp.1.left."...", ...

As in the case of a procedure, when a process needs to acquire one of an array of identical resources, it really cannot matter which element of the array is selected on a given occasion. Any element which is ready to respond to the "acquire" signal will be acceptable. A general choice construction will make the required arbitrary choice:

$\square_{i \geqslant 0}$ (lp.i.acquire $\longrightarrow$ ... lp.i.left!l ...; lp.i.release)

Here, the initial lp.i.acquire will acquire whichever of the two LP processes is ready for this event. If neither is ready, the acquiring process will wait; if both are ready, the choice between them is non-deterministic. After the initial acquisition, the bound variable i takes as its value the index of the selected resource, and all subsequent communications will be correctly directed to that same resource.

When a shared resource has been acquired for temporary use within ~~another process, the resource is~~ intended to behave exactly like a locally declared subordinate process, communicating only with its using subprocess. Let us therefore adapt the familiar notation for subordination, and write

(myfile::lp // ... myfile.left!x ...)

instead of the much more cumbersome construction:

$\square_{i \geqslant 0}$ (lp.i.acquire $\longrightarrow$ ... lp.i.left!x ...; lp.i.release)

Here, the local name "myfile" has been introduced to stand for the indexed name "lp.i", and the technicalities of acquisition and release have been conveniently suppressed. The new :: notation is called "remote subordination"; it is distinguished from the familiar : notation in that it takes a process <u>name</u> on its right, instead of a process.

X4 Two output files

A using process requires simultaneous use of two line printers to output two files, f1 and f2

f1::lp // f2::lp // ... f1.left!l1;f2.left!l2 ...

Here, the using process interleaves output of lines to the two different files; but each line is printed on the appropriate printer.

Of course, deadlock will be the certain result of any attempt to declare <u>three</u> printers simultaneously; it is also a likely result of declaring two printers simultaneously in each of two concurrent processes.

A formal definition of the new operator

$$(n::m \mathbin{/\!/} P)$$

must ensure that whichever actual resource is acquired, that particular resource will not be used again within its scope P. Let j be the identity of the selected resource. First we define P'_j as behaving like P, except that it is not allowed to communicate with m.j:

$$\text{traces}(P'_j) = \{ tr \mid tr \in P \wedge tr \upharpoonright \{ x \mid \exists y.(x = m.j.y) \} = \langle\rangle \}$$

Now we can safely replace communications between P'_j and the process named n by communications with the real resource m.j:

$$P''_j = f^{-1}(P')$$

where $f(lp.j.x) = n.x$

$f(y) = y$ if y does not begin with lp.j.

It remains only to insert ~~the necessary~~ acquire and release signals, and allow j to range over all natural numbers

$$(n::m \mathbin{/\!/} P) = \underset{j \geq 0}{\Box} (m.j.\text{acquire} \longrightarrow P''_j \,;\, m.j.\text{release})$$

This is a rather elaborate definition; but one should not be surprised. The definition gives a mechanistic model of how global resources can be safely shared among many processes, with a guarantee that each instance is wholly protected from every other.

X5 Scratchfile

A scratchfile is used for output of a sequence of blocks. When the output is complete, the file is rewound, and the entire sequence of blocks is read back from the beginning. The scratchfile will then give only "empty" signals; no further reading or writing is possible

```
$SCRATCH = WRITE_{\langle\rangle}$
$WRITE_s = (left?x \longrightarrow WRITE_{s\frown\langle x\rangle}$
              $| rewind \longrightarrow READ_s)$
$READ_{\langle x\rangle\frown s} = (right!x \longrightarrow READ_s)$
$READ_{\langle\rangle} = (empty \longrightarrow READ_{\langle\rangle})$
```

This may conveniently be used as a subordinate process:

```
(myfile:SCRATCH // ... myfile.left!v ... myfile.rewind ...
                  ... (myfile.right?x ⟶ ...
                      | myfile.empty ⟶ ...) ...)
```

X6 Scratch files on backing store

The scratchfile described in X5 can be readily implemented by holding the stored sequence of blocks in the main store of a computer. But if the blocks are large and the sequence is long, this could be an uneconomic use of main store, and it would be better to store the blocks on a backing store. Since each block in a scratchfile is read and written only once, a backing store (X2) with destructive read-out will suffice. An ordinary scratchfile (held in main store) is used to hold the sequence of indices of the sectors of backing store on which the corresponding actual blocks of information are held.

```
$BSCRATCH = (pagetable:SCRATCH //$
    $\mu X.(left?x \longrightarrow (\Box_{i<B}\, back.i.left!x \longrightarrow$
                          $pagetable.left!i \longrightarrow X$
        $| rewind \longrightarrow pagetable.rewind \longrightarrow$
              $\mu Y.(pagetable.right?i \longrightarrow$
                    $back.i?x \longrightarrow right!x \longrightarrow Y$
                 $| pagetable.empty \longrightarrow empty \longrightarrow Y))$
```

This process is intended to be used as a subordinate process, in exactly the same way as SCRATCH:

```
    (myfile:BSCRATCH // ...)
```

But this raises a problem with the naming operator, which will cause communications of BSCRATCH with "back" to be renamed "myfile.back". We must therefore prohibit such double-named events, and redefine the prefixing operator so that it does not add a second name to an event which already has a name. With this redefinition, communications of (myfile:BSCRATCH) with the process (back:BSTORE) can be correctly maintained.

But there are worse problems. In the definition of the subordination operator // we made a stipulation that the alphabet of the subordinate process shall be wholly contained in the alphabet of its scope. Even if this restriction were relaxed, we would face a prohibition against sharing the backing store between a subordinate process and its scope: this prohibition would (for example) prevent simultaneous use of two separate instances of BSCRATCH

```
    f1:BSCRATCH // f2:BSCRATCH // ...
```

All the problems can be happily avoided by the use of remote subordination, as shown in the next example.

X7 Scratch filing system

A scratch filing system shares a single backing store among an arbitrary number of simultaneously active virtual resources. The first requirement is to insert acquire and release signals into the BSCRATCH of X6,

```
    VSCR  =  acquire ⟶
                 (BSCRATCH || μX.(empty ⟶ X
                                 | release ⟶ STOP))
```

For simplicity, we do not allow this resource to be released until it is known to be empty: any attempt to do so will deadlock.

Next, we construct an infinite array of these processes.

$$VSCRS = \underset{i \geqslant 0}{|||} \ i{:}VSCR$$

Note that we have taken advantage of the new convention for prefixing, so that communications of (i:VSCR) with (back:STORE) are not indexed with i. To allow the backing store to be shared among all the virtual scratch files, these communications have been interleaved by use of the $|||$ combinator instead of $||$. To confine the use of the backing store to the scratch filing system, it can be declared as a subordinate process:

$$VSCRSYS = (back{:}BSTORE \,//\, VSCRS)$$

Finally, the whole system can be declared and used:

$$vscr{:}VSCRSYS \,//\, \ldots.\ (f1{::}vscr \,//\, \ldots$$
$$f2{::}vscr \,//\, \ldots)\ldots)\ldots$$

This use of remote subordination allows "vscr" to be used with complete freedom, in exactly the same way as a local scratchfile SCRATCH; however, there remains a risk of deadlock if there are no free blocks left when all the virtual scratch files are still expanding; or if a using process misuses the scratchfile, for example by failing to read to the end before releasing it. This second danger could be averted by redesign of VSCR, so that it will accept a release signal at any time, and then read back the unread blocks, thereby releasing them for subsequent reuse. The requisite modifications are left as an exercise. A major reason for the size and complexity of operating systems is the need to deal with arbitrary misbehaviour on the part of using processes.

A simple batch processing operating system has the task of running jobs submitted by a number of programmers. Each job requires resources such as readers, storage, and printers. In a multiprogrammed operating system, many jobs will be running concurrently; and it is inevitable that jobs will sometimes have to wait to acquire needed resources which are currently being used by other jobs. Any resources which have been acquired by a waiting job will be idle during the wait. This leads to a decrease in the effective utilisation of resources, and a decrease in the effective throughput of jobs by the operating system. The reduction in efficiency due to sharing of resources among too many competing users is known as thrashing. In an operating system, thrashing must be avoided with as much care as deadlock – the extreme version of thrashing in which no progress is made at all.

One way to avoid thrashing is to ensure that each job only acquires one resource at a time; so that if there is any waiting involved, no other resource remains idle. Thus ideally each job will first acquire a reader to input all its cards; it will then release the reader and acquire main storage to run the program; and when the program has released its main storage, it will acquire a printer for the output. Of course, this scheme will not work unless there is storage less precious than main store to hold the whole of the input and the whole of the output between the three phases in the progress of the job. For this backing store is normally used, and the technique is known as spooling.

X8 Spooled printer

A single virtual printer uses a temporary scratch file (X7) to store blocks output by its using process. When the using process signals its release, then an actual printer (X3) is acquired to output the content of the temporary file:

```
VLP  =  (temp::vscr //
          μX.left?x ⟶ temp.left!x ⟶ X
             | release ⟶ temp.rewind ⟶
              (actual::lp //
              μY.(temp.right?y ⟶ actual.left!y ⟶ Y
                  | temp.empty ⟶ STOP)))
```

The requisite unbounded array is defined

```
VLPs  =  |||  i:(acquire ⟶ VLP)
         i≥0
```

If we want the actual line printers to be used only in spooling mode, we can declare them local to the spooling system:

```
OUTSPOOL  =  (lp:LP2 // VLPs)
```

This in its turn is declared subordinate to the using jobs

```
    lp:OUTSPOOL // (...(myfile::lp // ...)... ||| ...)
```

Note that spooled line printers are declared using exactly the same notation as if spooling were not in use, and the actual line printers were used directly, as in

```
    lp:LP2 // (...(myfile::lp // ...)... ||| ...)
```

X9 Input/output system

A similar spooling system INSPOOL may be designed for card readers. The scratch filing system must be shared between INSPOOL and OUTSPOOL, which must therefore be interleaved:

```
    IOSYSTEM = (vscr:VSCRSYS // (lp:OUTSPOOL
                                 ||| cr:INSPOOL))
```

X10 A multiprogrammed operating system

Let JOB be a standard program which inputs the cards of a single user's job on in.right, executes the job in accordance with the instructions on the cards, and outputs the results on out.left. JOB is designed to terminate successfully after a reasonable time, independent of the content of the cards.

The JOB program can be adapted to process a whole batch of users' jobs, one after the other. For each job, a card reader and a printer is acquired from the IOSYSTEM

```
    BATCH  =  *(in::cr // out::lp // JOB)
```

A multiprogrammed operating system can run (say) four such batches simultaneously, all sharing the same IOSYSTEM

$$\text{OPSYS} = \text{IOSYSTEM} \mathbin{//} (\mathop{|||}_{i<4} \text{BATCH})$$

In this design, the scratch filing system is shared only between the INSPOOL and OUTSPOOL processes, and is not accessible to JOB. Thus one of the major risks of deadlock mentioned in X7 can be averted. The risk of deadlock due to exhaustion of backing storage can be reduced by stopping further input of cards whenever the store is nearly full. (If the backing store is large enough, no one will want to submit more jobs when it is full, because they would have to wait too long for their results). If the number of free sectors continues to decline, stop one or more of the running jobs, preferably one that is producing more output than the input consumed. The control limits for invoking these policies should be set low enough to ensure that deadlock is considerably less frequent than breakdown of the hardware on which the operating system runs.

6.5 Scheduling

When a limited number of resources is shared among a greater number of potential users, there will always be the possibility that some aspiring users will have to wait to acquire a resource until some other process has released it. If at the time of release there are two or more processes waiting to acquire it, the choice of which waiting process will acquire the resource is non-deterministic. In itself, this is of little concern; but suppose, by the time a resource is released again, yet another process has joined the set of waiting processes. Since the choice between waiting process is again non-deterministic, the newly joined process may be the lucky one chosen. If the resource is heavily loaded, this may happen again and again. As a result, some of the processes may happen to be delayed forever, or at least for a wholly unpredictable and unacceptable period of time. This is known as the problem of "infinite overtaking".

One solution to the problem is to ensure that all resources are lightly used. This may be achieved either by providing more resources, or by rationing their use, or by charging a heavy price for the services provided. In fact, these are the only solutions in the case of a resource which is consistently under heavy load. Unfortunately, even a resource which is on average lightly loaded will quite often be heavily used for long periods (rush hours or peaks). The problem can sometimes be mitigated by differential charging to try to smooth the demand, but this is not always successful or even possible. During the peaks, it is inevitable that, on the average, using processes will be subject to delay. It is important to ensure that these delays are reasonably consistent and predictable – you would much prefer to know that you will be served within the hour, than to wonder whether you will have to wait one minute or one day.

The task of deciding how to allocate a resource among waiting users is known as scheduling. In order to schedule successfully, it is necessary to know which processes are currently waiting for allocation of the resource. For this reason, the acquisition of a resource cannot any longer be regarded as a single atomic event. It must be split into two events:

please, which requests the allocation

thankyou, which accompanies the actual allocation of the resource.

For each process, the period between the "please" and the "thankyou" is the period during which the process has to wait for the resource. In order to identify the requesting process, we will index each occurrence of "please", "thankyou" and "release" by a different natural number. The requesting process acquires its number on each occasion by the same construction as remote subordination (6.2 X3):

$$\square_{i \geq 0}\ (\text{res.i.please; res.i.thankyou; ...; res.i.release})$$

A simple and effective method of scheduling a resource is to allocate it to the process which has been waiting longest for it. This policy is known as "first come first served" (FCFS) or "first in first out" (FIFO). It is the queuing discipline observed by passengers who form themselves into a line at a bus stop.

In a place such as a bakery, where customers are unable or unwilling to form a line, there is an alternative mechanism to achieve the same effect. A machine is installed which issues tickets with strictly ascending serial numbers. On entry to the bakery, a customer takes a ticket. When a server is ready, he calls out the lowest ticket number of a customer who has taken a ticket but not yet been served. This is known as the "bakery algorithm", and is described more formally below. We assume that up to R customers can be served simultaneously.

X1 The bakery algorithm

we need to keep the following counts

p	customers who have said "please"
t	customers who have said "thankyou"
r	customers who have released their resources.

Clearly, at all times $r \leq t \leq p$. Also, at all times, p is the number that will be given to the next customer who enters the bakery, and t is the number of the next customer to be served. Also, $p - t$ is the number of waiting customers, and $R + r - t$ is the number of waiting servers. All counts are initially zero, and can revert to zero again when they are all equal.

One of the main tasks of the algorithm is to ensure that there is never simultaneously a free resource and a waiting customer; whenever such a situation arises, the very next event must be a thankyou.

```
BAKERY = $B_{0,0,0}$

$B_{p,t,r}$ = if $0 < r = t = p$ then BAKERY
              else if $R + r - t > 0 \wedge p - t > 0$
                        then $t.\text{thankyou} \longrightarrow B_{p,t+1,r}$
               else $(p.\text{please} \longrightarrow B_{p+1,t,r}$
                        $\mid (\square_{i<t}\, i.\text{release} \longrightarrow B_{p,t,r+1}))$
```

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP TECHNICAL MONOGRAPHS

SEPTEMBER 1983

This is a series of technical monographs on topics in the field of computation. Copies may be obtained from the Programming Research Group, (Technical Monographs), 8–11 Keble Road, Oxford, OX1 3QD, England.

PRG-2 Dana Scott
Outline of a Mathematical Theory of Computation

PRG-3 Dana Scott
The Lattice of Flow Diagrams

PRG-5 Dana Scott
Data Types as Lattices

PRG-6 Dana Scott and Christopher Strachey
Toward a Mathematical Semantics for Computer Languages

PRG-7 Dana Scott
~~*Continuous Lattices*~~

PRG-8 Joseph Stoy and Christopher Strachey
OS6 – an Experimental Operating System for a Small Computer

PRG-9 Christopher Strachey and Joseph Stoy
The Text of OSPub

PRG-10 Christopher Strachey
The Varieties of Programming Language

PRG-11 Christopher Strachey and Christopher P. Wadsworth
Continuations: A Mathematical Semantics for Handling Full Jumps

PRG-12 Peter Mosses
The Mathematical Semantics of Algol 60

PRG-13 Robert Milne
The Formal Semantics of Computer Languages and their Implementations

PRG-14 Shan S. Kuo, Michael H. Linck and Sohrab Saadat
A Guide to Communicating Sequential Processes

PRG-15 Joseph Stoy
The Congruence of Two Programming Language Definitions

PRG-16 C. A. R. Hoare, S. D. Brookes and A. W. Roscoe
A Theory of Communicating Sequential Processes

PRG-17 Andrew P. Black
Report on the Programming Notation 3R

PRG-18 Elizabeth Fielding
The Specification of Abstract Mappings and their implementation as B^+-trees