

# MECHANIZABLE PROOFS ABOUT PARALLEL PROCESSES

J.M. Cadiou and J.J. Lévy

IRIA-LABORIA

78 - Rocquencourt France

## Abstract

This paper attempts to use formal semantics of a class of parallel processes in order to carry out mechanizable proofs about them. The formalism used is LCF (Logic for Computable Functions, Milner [22]), with slight extensions. The processes we consider communicate by sharing memory, rather than by signals on communication lines. Parallelism is treated as non-determinism. We state properties such as mutual exclusion of critical sections, absence of deadlocks, determinacy, and we show examples of proofs.

## Introduction

The area of formal semantics and correctness of sequential programs has recently been strongly influenced by Scott's mathematical theory of computation (Scott [26]). From the work of Scott and Strachey [29], Strachey [30], one can see how a rich theory of formal semantics for sequential programs is now developing. Scott's computational induction principle has been used as the basis of a Logic for Computable Functions (LCF) by Milner [21],[22]. This system has been successfully used for proving the correctness of a compiler (Milner and Weyhrauch [24]). Furthermore, it has been shown that computation induction was as powerful as all other methods for proving a wide class of properties of sequential programs (Manna, Ness, Vuillemin [19], Vuillemin [31]).

The present paper is, in part, an attempt to carry some of these ideas over to the area of parallel programs. Some steps in this direction have already been made by Kahn [15] and Milner [23], and we have drawn heavily on their ideas. Related work can also be found in Bekic [3].

Various kinds of proofs about parallel programs have appeared in the literature. Some of them are based on a generalization of Floyd's method (Floyd [9], Manna [18]) to parallel programs (Ashcroft and Manna [2], Lauer [17], Elspas & al [8]). Ashcroft [1] makes use of relations to express the properties to be proved. The absence of deadlocks is in general treated separately from other correctness properties,

much like termination is treated separately from partial correctness in sequential programs. However the property of absence of deadlocks is in general expressed indirectly : for example, one shows that some "state" will repeat itself. Another kind of proof is found in Habermann [10],[11]. There, proofs are done at a higher level, and are not directly formalizable in a logical system. Yet another approach is found in Hoare [12], where some proofrules for parallel processes are proposed, in an attempt to extend to parallel programs the axiomatic formalism developed in Hoare [13] for sequential programs.

We approach the problem by expressing the semantics of parallel processes as a mathematical function, and then stating the property to prove as a property of that function in the form of an LCF sentence. For example, the fact that a process is deadlock free is expressed by stating that a certain string of characters is infinite. The proof of the property can then be carried out within the logic.

The processes that we consider can be thought of as non deterministic programs. Their execution is dependent upon an external oracle (choice function) which determines what operation is to be performed next. The various processes communicate by shared memory rather than by signals on communication lines. The basic underlying constraint is that if two operations want to access the memory at the same time, one of them will get serviced before the other.

In the first part of this paper we discuss simple parallel processes consisting of two concurrent sequential programs. In the second part we consider more general processes, including processes allowing parallel recursive calls of processes.

## I Simple parallel processes

We first give a mathematical semantic description of simple parallel processes. Then we discuss how to express and prove certain properties of these processes within LCF. The properties considered include mutual exclusion of critical sections, absence of deadlocks and determinacy.

## 1 Semantics

The concurrent sequential programs are specified in terms of basic (not interruptable) operations. They operate on a common memory, which means that they can share variables. The scheduling of the operations of the process is done by an oracle (or choice function) which determines which of the two sequential programs is to advance next. In the first paragraph we discuss the semantics of the sequential programs ; in the second paragraph we show how the ideas extend to the parallel case.

### 1.1. Sequential programs

Let  $P$  be a sequential program operating on a memory state  $S$ . We view  $S$  as a mapping from names (variables) into values.  $P$  consists of labelled elementary operations : each operation in  $P$  has a distinct label belonging to some alphabet  $L$ .

We wish to isolate a special operation, corresponding to the usual STOP statements. We do this by having a special predicate,  $\text{Stop} : L \rightarrow \{\text{tt}, \text{ff}, \perp\}$ , such that  $\text{Stop}(l) \equiv \text{tt}$  iff  $l$  labels a STOP operation. Notice that we must have  $\text{Stop}(\perp) \equiv \perp$  if  $\text{Stop}$  is to be a monotonic predicate.  $P$  is specified by two functions:

a) The next operation function, which is a mapping  $\text{Succ} : L \times S \rightarrow L$ , where, for  $l \in L$  and  $s \in S$ ,  $\text{Succ}(l, s)$  denotes the next operation of  $P$  to be performed after the execution of operation  $l$  on state  $s$ .

b) The next state function, which is a mapping  $\sigma : L \times S \rightarrow S$ , where, for  $l \in L$  and  $s \in S$ ,  $\sigma(l, s)$  denotes the state resulting from the execution of operation  $l$  on state  $s$ .

These two functions are sufficient to describe the input-output action of  $P$  as a state transformation function. The final state  $M(l, s)$  of  $P$  started at operation  $l$  on state  $s$  is recursively defined by :

$$(1) M(l, s) \Leftarrow \text{Stop}(l) \rightarrow s, M(\text{Succ}(l, s), \sigma(l, s)),$$

where we use the notation " $p \rightarrow A, B$ " for the sequential conditional connective "if  $p$  then  $A$  else  $B$ ". Notice that we cannot prove that the final state is the least fixpoint of this equation, since we have no other formal definition of how  $P$  operates. Our point of view here is that all correct implementations of  $P$  will have to verify (1).

\*)  $\text{tt}, \text{ff}, \perp$  denote the logical values True, False and Undefined.  $\equiv$  denotes equality.  $\perp$  also denotes the undefined (or bottom) element of any domain. We assume that  $L$  is a flat domain.

This type of semantic description of a program is satisfactory for many purposes when one only deals with sequential programs. It is, in fact, quite similar to the classical recursive equations obtained when one translates a flowchart program following McCarthy [20].

However, for one thing, it fails to capture the difference between two programs which loop forever but with a different behavior : say one of them prints an infinite number of A's, the other an infinite number of B's.

Another well known problem is that the knowledge of the state transformation function of two sequential programs is not sufficient to determine the state transformation resulting of their execution in parallel. (See Milner [23] for a discussion of these ideas).

Clearly then, one must have more information about the histories of the computations of the sequential programs in order to resolve these ambiguities.

To solve the problem, we propose to add a third function to the basic description of  $P$ , a mapping  $\text{out} : L \times S \rightarrow D$ , where  $D$  is some set. For all  $l \in L$ ,  $s \in S$ ,  $\text{out}(l, s)$  denotes the output produced by operation  $l$  on state  $s$ . One can think of  $D$  as an alphabet and of the output as a write operation on an external line printer. We assume that  $D$  possesses a special element  $\perp$ , which can be interpreted as a blank character.

Then we might describe the semantics of  $P$  with the following recursive definition :

$$(2) \text{TR}(l, s) \Leftarrow \text{Stop}(l) \rightarrow \text{Out}(l, s), \perp, \\ \text{Out}(l, s). \text{TR}(\text{Succ}(l, s), \sigma(l, s))$$

$\text{TR}(l, s)$  traces the output of the individual operations when they are executed.

More formally,  $\text{TR}$  will be a mapping of  $L \times S \rightarrow D^\omega$ , where  $D^\omega$  is the set of finite and (one-way) infinite strings of elements of  $D$ . The operation "." used in (2) is the concatenation, a mapping of  $D \times D^\omega \rightarrow D^\omega$ . The empty string is denoted  $\perp$ . Then we have the two selector functions  $\text{hd} : D^\omega \rightarrow D$  and  $\text{tl} : D^\omega \rightarrow D^\omega$ . These operations satisfy the following axioms :

$$\text{hd}(\perp) \equiv \text{tl}(\perp) \equiv \perp.$$

$$\forall u \in D^\omega : \perp.u \equiv \perp, \perp.u \equiv u, \text{hd}(u).\text{tl}(u) \equiv u$$

$$\forall \alpha \in D, u \in D^\omega : \text{hd}(\alpha.u) \equiv \alpha, \text{tl}(\alpha.u) \equiv u.$$

$D^\omega$  is further structured as a partially ordered set by introducing the prefix ordering :  $u \subseteq v$  iff  $u$  is a prefix of  $v$ . An infinite string is defined as the least upper bound of an infinite sequence of increasing strings. All this construction can be made

formally.  $D^\omega$  is thus structured as a complete partially ordered set (i.e. a partially ordered set with a minimal element, and such that every ascending chain has a least upper bound). The operations  $\cdot$ ,  $hd$ ,  $tl$  can be shown to be continuous in the sense of Scott [26], and this is enough to guarantee that (2) has a least fixpoint, obtained as a least upper bound of increasing approximations (See Cadiou [5]). The domain  $D^\omega$  was first introduced by Kahn [15].

The basic output function  $Out(l,s)$  can be specified in various ways, according to one's purposes. For example if one is interested in tracing what happens in location  $x$  of the memory, one will choose :

$\forall l \forall s \text{ Out}(l,s) \equiv s(x)$ . If one wants to trace the whole history of control and states through which the program goes, one will choose  $\forall l \forall s \text{ Out}(l,s) \equiv \langle l,s \rangle$ . Let us denote  $U$  the tracing function corresponding to that last choice. By (2), we have :

$$(3) U(l,s) \leq \text{Stop}(l) \rightarrow \langle l,s \rangle \cdot 1, \\ \langle l,s \rangle \cdot U(\text{Succ}(l,s), \sigma(l,s)).$$

It is interesting to note that any tracing function of  $P$  can be reconstructed from  $U$ : let  $Out(l,s)$  be an arbitrary output function,  $TR(l,s)$  its corresponding tracing function as defined by (2). Then one can prove that :

$$TR(l,s) \equiv \text{Extract}(Out(U(l,s))),$$

where, for any  $f \in L \times S \rightarrow D$ , and any  $u \in (L \times S)^\omega$ ,

$$\text{Extract}(f,u) \leq \text{Stop}((hdu)_1) \rightarrow f((hdu)_1, (hdu)_2) \cdot 1, \\ f((hdu)_1, (hdu)_2) \cdot \text{Extract}(f,tl u).$$

$( )_1$  and  $( )_2$  are respectively the first and second projection functions <sup>\*</sup>).

Similarly the state transformation function  $M$  defined by (1) can be retrieved from  $U$ . In fact,  $U$  contains all the information we have about  $P$ , since  $\text{Stop}(l) \equiv ff \Rightarrow \text{Succ}(l,s) \equiv (hd \text{ tl } U(l,s))_1$ , and  $\sigma(l,s) \equiv (hd \text{ tl } U(l,s))_2$ .

## 1.2. Simple parallel processes

The ideas of the previous paragraph generalize easily to simple parallel processes. A simple parallel process consists of two sequential programs  $P_1$  and  $P_2$  operating on the same memory state  $S$ . The label sets  $L_1$  and  $L_2$  of  $P_1$  and  $P_2$  are supposed disjoint.

In order to determine which of  $P_1$  and  $P_2$  must advance, we use an oracle. An oracle, or choice sequence, is a string in  $\Omega = B^\omega$ , where  $B = \{tt, ff, \perp\}$

Then a natural way of defining the state transformation function  $\mathcal{M}$ , as a function of  $L_1 \times L_2 \times S \times \Omega \rightarrow S$  is :

$$(4) \mathcal{M}(l_1, l_2, s, \omega) \leq \text{Stop}(l_1) \rightarrow M(l_2, s), \\ \text{stop}(l_2) \rightarrow M(l_1, s), \\ hd \omega \rightarrow \mathcal{M}(\text{Succ}(l_1, s), l_2, \sigma(l_1, s), tl \omega), \\ \mathcal{M}(l_1, \text{Succ}(l_2, s), \sigma(l_2, s), tl \omega).$$

In this equation,  $M$  is the sequential state transformation function of (1); we have dropped the indices from  $\text{Succ}_1$ ,  $\text{Succ}_2$ ,  $\sigma_1$  and  $\sigma_2$  since they have disjoint domains.

Similarly, we can write a tracing function  $\mathcal{TR}$  corresponding to any function  $Out : L_1 \times L_2 \times S \times B \rightarrow D$ . Just as in the sequential case, there is a most general one,  $\mathcal{U}$ , satisfying :

$$(5) \mathcal{U}(l_1, l_2, s, \omega) \leq \text{Stop}(l_1) \rightarrow U(l_2, s), \\ \text{Stop}(l_2) \rightarrow U(l_1, s), \\ hd \omega \rightarrow \langle l_1, l_2, s, hd \omega \rangle \cdot \mathcal{U}(\text{Succ}(l_1, s), l_2, \sigma(l_1, s), tl \omega), \\ \langle l_1, l_2, s, hd \omega \rangle \cdot \mathcal{U}(l_1, \text{Succ}(l_2, s), \sigma(l_2, s), tl \omega).$$

$\mathcal{U}$  is a mapping of  $L_1 \times L_2 \times S \times \Omega \rightarrow \{L_1 \times L_2 \times S \times B\}^\omega$ , if we consider  $\{L_1 \times S\}^\omega$  and  $\{L_2 \times S\}^\omega$  as subsets of  $\{L_1 \times L_2 \times S \times B\}^\omega$  which can be done in a number of ways. Since  $\text{Succ}$  and  $\sigma$  can be reconstructed from  $U$   $\mathcal{U}$  can be expressed in terms of  $U$ . However, we will not pursue this further here, since part II will provide us with more powerful tools to do it.

Example : Let us give, as example, the semantic description of the following simple parallel process, which is classical in studies of the mutual exclusion problem.

<sup>\*</sup>) one needs the axioms  $(\perp)_1 \equiv (\perp)_2 \equiv \perp$

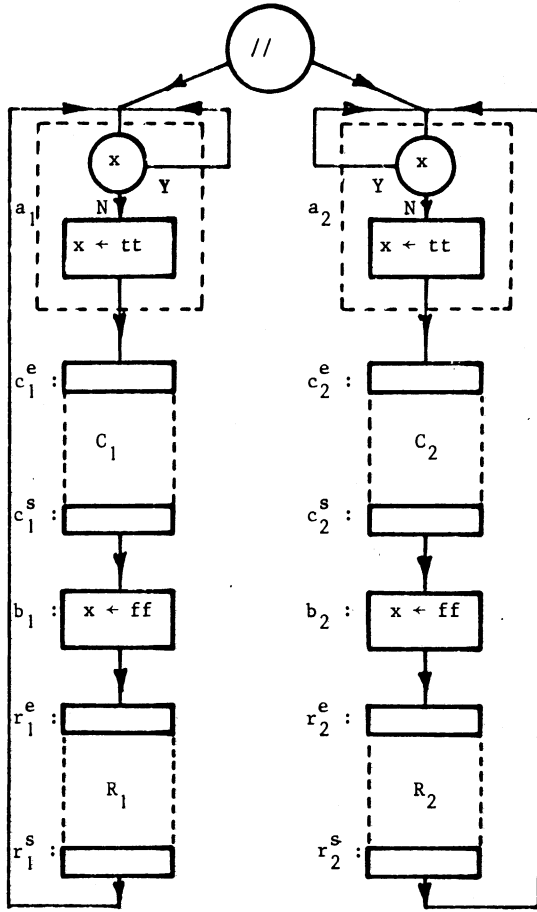


Figure 1

The notations used are fairly standard. The dotted rectangles around a group of instructions mean that the group is not interruptable, and must be considered as an indivisible operation. Therefore a dotted rectangle has a single label. The portions  $C_1, C_2, R_1, R_2$  are interruptable. They stand, respectively, for the critical sections and the rest of the programs. The only assumptions about them is that they are single-entry, single-exit pieces of programs which do not access location  $x$ . The label of the first (entry) instruction in  $C_1$  is  $c_1^e$ , the label of the last (exit) instruction in  $C_1$  is  $c_1^s$ , and similarly for  $C_2, R_1, R_2$ .

The label alphabet  $L_1$  of the left sequential program  $P_1$  is  $\{a_1\} \cup \mathcal{C}_1 \cup \mathcal{R}_1$ , where  $\mathcal{C}_1$  and  $\mathcal{R}_1$  are the label alphabets of  $C_1$  and  $R_1$  respectively.

The basic semantic functions  $\text{Succ}$  and  $\sigma$  of  $P_1$  satisfy :

$$\left\{ \begin{array}{l} \text{Succ}(a_1, s) \equiv s(x) \rightarrow a_1, c_1^e \\ \text{Succ}(c, s) \in \mathcal{C}_1, \forall c \in \mathcal{C}_1 \setminus \{c_1^s\} \\ \text{Succ}(c_1^s, s) \equiv b_1 \\ \text{Succ}(b_1, s) \equiv r_1^e \\ \text{Succ}(r, s) \in \mathcal{R}_1, \forall r \in \mathcal{R}_1 \setminus \{r_1^s\} \\ \text{Succ}(r_1^s, s) \equiv a_1 \end{array} \right.$$

$$\left\{ \begin{array}{l} \sigma(a_1, s) \equiv s(x) \rightarrow s, \lambda z. (z = x \rightarrow tt, s(z)) \\ \sigma(c, s)(x) \equiv s(x), \forall c \in \mathcal{C}_1 \\ \sigma(b_1, s) \equiv \lambda z. (z = x \rightarrow ff, s(z)) \\ \sigma(r, s)(x) \equiv s(x), \forall r \in \mathcal{R}_1 \end{array} \right.$$

The equations are analogous for  $P_2$ .

In this case, the tracing function  $\mathcal{TR}$  is defined by :

$$(6) \quad \mathcal{TR}(l_1, l_2, s, \omega) \Leftarrow$$

$$\begin{aligned} \text{hd} \omega \rightarrow \text{Out}(l_1, l_2, s, tt) \mathcal{TR}(\text{Succ}(l_1, s), l_2, \sigma(l_1, s), \text{tl} \omega), \\ \text{Out}(l_1, l_2, s, ff) \mathcal{TR}(l_1, \text{Succ}(l_2, s), \sigma(l_2, s), \text{tl} \omega). \end{aligned}$$

In the remainder of part I we discuss the problem of stating and proving properties of simple parallel processes.

## 2 Mutual exclusion

The mutual exclusion problem has been extensively studied by many authors (Dijkstra [7], Knuth [16], Habermann [11], Bredt [4], to mention just a few). The problem is to ensure that two pieces of code of  $P_1$  and  $P_2$  never get executed simultaneously. These pieces of code are called critical sections. We denote them  $C_1$  and  $C_2$  respectively. Once  $P_1$  has entered  $C_1$ , and until it leaves it,  $P_2$  is forbidden to enter  $C_2$ , and vice-versa.

Here, we are interested in proving that a proposed simple parallel process has the mutual exclusion property with respect to  $C_1$  and  $C_2$ . The problem of proving that  $P_1$  and  $P_2$  eventually execute their critical section is discussed separately (in the next section).

There are several ways to express the mutual exclusion problem in the logic. For example, we can use a special output function  $\text{Out}(l_1, l_2, s, t)$  which will output some character, say  $\alpha$ , iff  $l_1$  and  $l_2$  are simultaneously in  $C_1$  and  $C_2$ . Then we say that the process has the desired property iff the corresponding

tracing function is a string with no  $\alpha$ 's.

Formally, we define :

$$\text{Out}(l_1, l_2, s, t) \Leftarrow \langle l_1, l_2 \rangle \in \mathcal{C}_1 \times \mathcal{C}_2 \rightarrow \alpha, I,$$

and use the corresponding tracing function  $\mathcal{CR}$  as defined by (6). We will say that the mutual exclusion property is satisfied for  $P_1$  and  $P_2$  with respect to  $C_1$  and  $C_2$  iff the string produced by  $\mathcal{CR}$  only contains blank characters. It is easy to see that such a string can only be  $\perp$  since  $I. \perp \equiv \perp$ . Hence the property to prove is :

$$(7) \forall \omega \mathcal{CR}(a_1, a_2, s_0, \omega) \equiv \perp,$$

where  $a_1$  and  $a_2$  are the first operations of  $P_1$  and  $P_2$  and  $s_0$  the state in which the process is started.

We will sketch the proof of (7) in the case of the process of Figure 1. In fact, as usual in inductive proofs, we prove a more general property, namely :

$$(8) \forall l_1, l_2, s, \omega D(l_1, l_2, s) :: \mathcal{CR}(l_1, l_2, s, \omega) \equiv \perp$$

where the notation  $\forall x P :: A \equiv B$  stands for :

$$\lambda x. (P \rightarrow A, \perp) \equiv \lambda x. (P \rightarrow B, \perp). \text{ (See Milner [21])}.$$

$D(l_1, l_2, s)$  is the predicate defined by :

$$(9) D(l_1, l_2, s) \equiv (l_1 \in \{a_1\} \cup \mathcal{R}_1 \wedge l_2 \in \{a_2\} \cup \mathcal{R}_2) \rightarrow tt, \\ (l_1 \in \{b_1\} \cup \mathcal{C}_1 \wedge l_2 \in \{a_2\} \cup \mathcal{R}_2) \rightarrow s(x), \\ (l_1 \in \{a_1\} \cup \mathcal{R}_1 \wedge l_2 \in \{b_2\} \cup \mathcal{C}_2) \rightarrow s(x), \\ ff.$$

All the functions and predicates used here can be defined (or axiomatized) as continuous objects, since  $L_1$  and  $L_2$  are flat domains<sup>\*</sup>, and therefore (8) is an admissible property. For defined values of the parameters, the values of  $D(l_1, l_2, s)$  can be conveniently represented by the table of Figure 2.

$l_1 \backslash l_2$	$a_2$	$\mathcal{C}_2$	$b_2$	$\mathcal{R}_2$
$a_1$	tt	s(x)	s(x)	tt
$\mathcal{C}_1$	s(x)	ff	ff	s(x)
$b_1$	s(x)	ff	ff	s(x)
$\mathcal{R}_1$	tt	s(x)	s(x)	tt

Figure 2 :  $D(l_1, l_2, s)$

<sup>\*</sup> A flat domain is a set  $S$  with a partial order  $\subseteq$  where the only pairs in the order are  $\perp \subseteq a, a \subseteq a, \forall a \in S$ .

We prove (8) by computational induction on the recursive definition (6) of  $\mathcal{CR}$ <sup>\*\*</sup>.

The base case is trivial. The induction step easily follows from the facts that :

$$\partial(s(x)) \equiv tt, D(l_1, l_2, s) \equiv tt \vdash \text{Out}(l_1, l_2, s, t) \equiv I,$$

$$D(\text{Succ}(l_1, s), l_2, \sigma(l_1, s)) \equiv tt,$$

$$D(l_1, \text{Succ}(l_2, s), \sigma(l_1, s)) \equiv tt,$$

where  $\partial(S(x)) \equiv tt$  means that  $s(x)$  is not  $\perp$ .

In turn, these are easily, albeit tediously, derivable from the definitions of Out, D, Succ and  $\sigma$ . For example, let us show:  $\partial(S(x)) \equiv tt, D(l_1, l_2, s) \equiv tt \vdash$

$$D(\text{Succ}(l_1, s), l_2, \sigma(l_1, s)) \equiv tt$$

in the interesting case  $l_1 \equiv a_1$ .

$$\text{We have: } \begin{cases} \text{Succ}(a_1, S) \equiv s(x) \rightarrow a_1, c_1^e, \\ \sigma(a_1, S)(x) \equiv s(x) \rightarrow s(x), tt. \end{cases}$$

Let us denote  $D(\text{Succ}(l_1, s), l_2, \sigma(l_1, s))$  by  $D'$ .

Assume  $D(l_1, l_2, s) \equiv tt$ . We argue by cases on  $s(x)$ :

$s(x) \equiv \perp$  leads to a contradiction with  $\partial(s(x)) \equiv tt$ .

$s(x) \equiv tt$  implies  $D' \equiv D(a_1, l_2, s')$ , with  $s'(x) \equiv tt$ .

Hence  $D' \equiv tt$ .

$s(x) \equiv ff$  implies  $D' \equiv D(c_1^e, l_2, s')$  with  $s'(x) \equiv tt$ .

Furthermore,  $D(l_1, l_2, s) \equiv tt$  implies, in this case,  $l_2 \in \{a_2\} \cup \mathcal{R}_2 \equiv tt$ . But then  $D' \equiv s'(x) \equiv tt$ .

We will not bother the reader with the rest of the proof, which is equally easy.

This proof seems complicated for such a simple property. However, all the cases are quite easy to prove, and well within to power of existing automatic proving systems. Furthermore, the proof brings out all the cases in a systematic way, including the undefined cases (Here for instance the property does not depend on the definedness of the oracle). Also, the predicate  $D(l_1, l_2, S)$  summarizes the facts about the computation which are relevant to the property to prove (for example : "when  $P_1$  enters  $C_1$ ,  $x$  is True", etc...). Therefore, if one intuitively understands why the property is true, one should be able to guess  $D(l_1, l_2, S)$  without too much trouble.

<sup>\*\*</sup> See Manna, Ness, Vuillemin [19] for a discussion of computational induction. Essentially, if  $F(x) \Leftarrow \tau\{F\}(x)$  is a recursive definition, to prove  $P(F)$ , one shows  $P(\perp)$  and  $\forall f\{P(f) \supset P(\tau\{F\})\}$ . For this to be valid,  $P$  must be an admissible property.

Notice that if one wants to do an informal proof of the correctness of this program, one almost invariably proceeds by examining some property of a "control state" consisting of  $l_1, l_2$  and the value of  $x$ . This "control state" can only take a finite number of different values, and therefore we have essentially a finite (hence decidable) problem to solve. There would not seem to be too much point in using a theorem prover when a decision procedure can be used (essentially a determination of unreachable vertices in a graph). However, we must emphasize that, from a rigorous point of view, the assumption that the "control state"  $\langle l_1, l_2, x \rangle$  contains all the "relevant" information needs to be justified. It does, for example, depend on the hypothesis that  $C_1, R_1, C_2, R_2$  do not modify  $x$ . Indeed, the fact that the predicate  $D(l_1, l_2, s)$  was sufficient to carry out the LCF proof constitutes such a justification. One of the advantages of a completely formal proof system is that it brings out all the hypothesis that are necessary for the property to hold, and which might easily be overlooked in an informal proof.

There are other ways of expressing the mutual exclusion property in the logic, but we will not expand on this further here.

### 3. Absence of deadlocks

We will now discuss another property, the absence of deadlocks in a process. Again we will focus the discussion on specific examples, but the methods used are quite general.

We first consider the process of Figure 3.

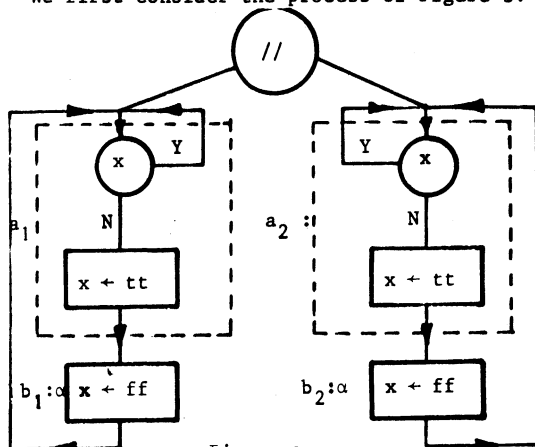


Figure 3

Intuitively what is meant by absence of deadlock is that some sort of activity goes on for ever. The sort of activity one wants to consider depends on the problem. For example if this process starts execution with  $x \equiv tt$  all it can do is loop in the initial tests.

It will do so forever, provided the oracle is always defined.

A more interesting problem is to try to prove that if the process starts executing with  $x \equiv ff$  then at least one of the operations  $b_1$  or  $b_2$  will get executed an infinite number of times. Now this is not true in general: for example if the oracle is undefined, nothing gets executed; moreover, if the oracle consists of  $tt$  followed by an infinite sequence of  $ff$ 's, the process will execute neither  $b_1$  or  $b_2$ . Therefore the property will require some restriction on the oracle  $\omega$ .

A natural restriction on  $\omega$  is that it have no right part consisting solely of  $tt$ 's or solely of  $ff$ 's. In other words,  $\omega$  should in some sense never "forget" one of its processes. Also,  $\omega$  should be infinite defined, which means that it should always return either  $tt$  or  $ff$ , and never "hesitate" for ever between its two processes. In order to express this property formally within our system, we introduce some recursively defined predicates and functions:

$$(10) \begin{cases} T(\omega) \leq hd\omega \rightarrow tt, T(tl\omega) \\ F(\omega) \leq hd\omega \rightarrow F(tt\omega), tt \end{cases} \quad T, F : B^\omega \rightarrow \{tt, \perp\}$$

$T$  looks for a  $tt$  in  $\omega$  and  $F$  looks for a  $ff$ . Certainly we want  $T(\omega) \equiv F(\omega) \equiv tt$ , but that is not enough. We also want the subsequent portion of  $\omega$  to have the same property. However, this cannot be expressed as a continuous predicate  $P$  in  $B^\omega \rightarrow \{tt, \perp\}$ . For if we had  $P(\omega) \equiv tt$  for some infinite string  $\omega$ , then by continuity of  $P$ , we would also have  $P(\alpha) \equiv tt$  for some finite prefix  $\alpha$  of  $\omega$ .

But this cannot be, because we want  $P$  to be true only of infinite oracles. In fact this argument will hold for any finite truth value range. But we can solve the problem by taking  $B^\omega$  itself as a "truth value" range.

Let us define  $\mathcal{G}, \mathcal{F} : B^\omega \rightarrow B^\omega$ , by:

$$(11) \begin{cases} \mathcal{G}(\omega) \leq T(\omega). \mathcal{G}(\omega) \\ \mathcal{F}(\omega) \leq F(\omega). \mathcal{F}(\omega) \end{cases}$$

Then the property we want can be expressed by  $\mathcal{G}(\omega) \equiv \mathcal{F}(\omega) \equiv tt^*$ , where  $tt^*$  is the infinite string consisting of  $tt$ 's. (we can define  $tt^*$  in the logic by  $tt^* \leq tt.tt^*$ ). We will denote this property  $\mathcal{J}(\omega)$ .

Returning to our example, we wish to show that, if  $\mathcal{J}(\omega)$  is satisfied, then either  $b_1$  or  $b_2$  will

get executed an infinite number of times. \*)

We can express this by taking a tracing function  $\mathcal{TR}$ , of the family described by (6), which outputs  $\alpha$  whenever  $b_1$  or  $b_2$  is executed, and saying that :

$$\mathcal{F}(\omega) \vdash \mathcal{TR}(a_1, a_2, ff, \omega) \equiv \alpha^*$$

Then, if we set :

$$X(\omega) \equiv \mathcal{TR}(a_1, a_2, ff, \omega)$$

$$Y(\omega) \equiv \mathcal{TR}(b_1, a_2, tt, \omega)$$

$$Z(\omega) \equiv \mathcal{TR}(a_1, b_2, tt, \omega), \text{ we can show that}$$

$X, Y, Z$  are defined by the recursive system :

$$(12) \begin{cases} X(\omega) \Leftarrow hd\omega \rightarrow Y(tl\omega), Z(tl\omega) \\ Y(\omega) \Leftarrow hd\omega \rightarrow \alpha.X(tl\omega), Y(tl\omega) \\ Z(\omega) \Leftarrow hd\omega \rightarrow Z(tl\omega), \alpha.X(tl\omega) \end{cases}$$

(Justifying formally the transformation of (6) into (12) for this particular case requires an instance of the general theorem  $\lambda x. (\mu F. \tau) \equiv \mu F. (\lambda x. \tau)$ , which is provable in LCF.  $\mu$  here denotes the fixpoint operator).

Our problem is now to prove that :

$$(13) \mathcal{C}(\omega) \equiv \mathcal{F}(\omega) \equiv tt^* \vdash X(\omega) \equiv \alpha^*$$

We sketch the proof here :

First we show :

(14)  $\delta(\underline{T}(\omega)) :: \underline{Y}(\omega) \equiv \alpha.X(\underline{\theta}(\omega))$ , by parallel computational induction on the underlined functions.  $\delta$  here is the monotonic "is defined" predicate on  $\{tt, ff, \perp\}$ , axiomatized by  $\delta(\perp) \equiv \perp$ ,  $\delta(tt) \equiv \delta(ff) \equiv tt$ .  $T$  is defined by (10),  $X, Y$  by (12), and  $\theta(\omega)$  by :

$$(15) \theta(\omega) \Leftarrow hd\omega \rightarrow tl\omega, \theta(tl\omega).$$

Intuitively, (14) says that, because of the property  $\mathcal{F}(\omega)$ ,  $Y$  as defined in (12), must eventually call  $X$  after outputting  $\alpha$ . In terms of the process, it says that if control is in  $b_1, a_2$  with  $x \equiv ff$ , it must eventually advance to  $a_1, a_2$ . The proof of (14) is as follows :

- Base case : trivial since  $(\perp :: a \equiv b)$  always holds.
- Inductive step :

$$\text{Assume } P(\omega) : \delta(t(\omega)) :: y(\omega) \equiv \alpha.X(h(\omega)),$$

$$\text{Show } P'(\omega) : \delta(t'( \omega)) :: y'(\omega) \equiv \alpha.X(h'(\omega)),$$

\*) Notice that, in general, it is not true that they both do : for example, the oracle  $\omega$  defined by  $\omega \Leftarrow tt.ff.tt.\omega$  satisfies the axiom, but  $b_2$  never gets executed. This means that there can be infinite waiting of one process.

$$\begin{aligned} \text{where : } t'(\omega) &\equiv hd\omega \rightarrow tt, t(tl\omega), \\ y'(\omega) &\equiv hd\omega \rightarrow \alpha.X(tl\omega), y(tl\omega), \\ h'(\omega) &\equiv hd\omega \rightarrow tl\omega, h(tl\omega). \end{aligned}$$

We do case analysis on  $hd\omega$  :

- .  $hd\omega \equiv \perp$ ,  $P'$  reduces to  $\perp :: \perp \equiv \alpha.X(\perp)$ , which holds.
- .  $hd\omega \equiv tt$ ,  $P'$  reduces to  $tt :: \alpha.X(tl\omega) \equiv \alpha.X(tl\omega)$ , which holds.
- .  $hd\omega \equiv ff$ ,  $P'$  reduces to :  
 $\delta(t(tl\omega)) :: y(tl\omega) \equiv \alpha.X(h(tl\omega))$ , which is  $P(tl\omega)$ , and thus holds by the inductive hypothesis.

Similarly, one can prove :

$$(16) \delta(\underline{F}(\omega)) :: \underline{Z}(\omega) \equiv \alpha.X(\underline{\eta}(\omega)), \text{ where :}$$

$$\eta(\omega) \Leftarrow hd\omega \rightarrow \eta(tl\omega), tl\omega, \text{ and, using (12) :}$$

$$(17) \delta(\underline{T}(\omega)) :: \delta(\underline{F}(\omega)) :: X(\omega) \equiv \alpha.X(u(\omega)),$$

where  $u(\omega) \equiv hd\omega \rightarrow \theta(tl\omega), \eta(tl\omega)$ .

Then using (11), we can deduce :

$$(18) \mathcal{C}(\omega) \equiv \mathcal{F}(\omega) \equiv tt^* \vdash X(\omega) \equiv \alpha.X(u(\omega)).$$

We can now use an instance of a general theorem which says that for any  $X, u \in D^\omega \rightarrow D^\omega$ , and any  $\alpha \in D$ , from  $Q(\omega) \vdash X(\omega) \equiv \alpha.X(u(\omega))$  and  $Q(\omega) \vdash Q(u(\omega))$ , one can deduce  $Q(\omega) \vdash X(\omega) \equiv \alpha^*$ . This theorem is provable in LCF.

We won't show the detailed proof that :

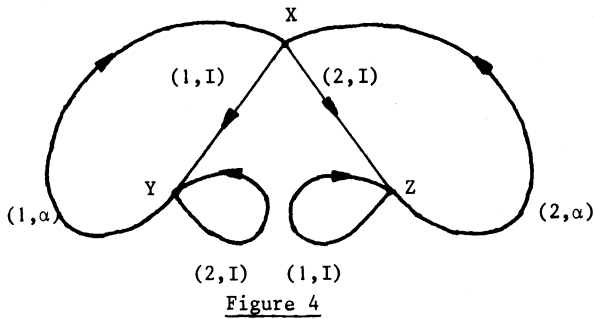
$\mathcal{F}(\omega) \vdash \mathcal{F}(u(\omega))$ , (where  $\mathcal{F}(\omega)$  is  $\mathcal{C}(\omega) \equiv \mathcal{F}(\omega) \equiv tt^*$ ), but it is easy using (10), (11) and the definition of  $\theta(\omega)$  and  $\eta(\omega)$ , and so the theorem applies.

This essentially proves (13), which was our statement of the absence of deadlock property.

Now, as it turns out, this proof is a special instance of a systematic proof procedure which enables us to prove absence of deadlock whenever a process has a finite number of "control states", in the sense which we discussed in the previous section. When this happens, the process can be described by a finite number of recursive equations of the form  $E_i : X_i(\omega) \Leftarrow hd\omega \rightarrow \alpha.X_{i_1}(tl\omega), \beta_i.X_{i_2}(tl\omega)$ , such that every unknown on the right also appear on the left. Such a system of equations can be translated into a finite graph in the following way :

Each unknown  $X_i$  in the recursive system is represented by a node of the graph. To each equation of the form  $E_i$  will correspond the two arcs :  $X_i \rightarrow X_{i_1}$ , labelled  $(1, \alpha_i)$ , and  $X_i \rightarrow X_{i_2}$ , labelled  $(2, \beta_i)$ .

For example, Figure 4 represents the graph associated with our previous example.

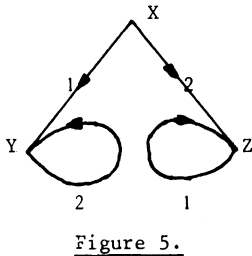


Now the property  $\mathcal{F}(\omega)$  has a remarkable expression in terms of these graphs. It says that if the graph has a cycle of which all the arcs have first label 1, then no acceptable computation path can loop forever in that cycle (we say that such a cycle is of Type 1). Similarly for cycles containing only 2's (cycles of Type 2). All other paths represent possible computations.

This observation suggests the following property to characterize whether a process representable by such a graph is deadlock free<sup>\*</sup>). Let us suppose that there is one "start" node X, and one non blank character  $\alpha$  used in the equations in such a way that the process is deadlock free iff  $X(\omega) \equiv \alpha^*$ .

Then if we call G the connected component of the graph which contains X, and G' the graph obtained from G by deleting all the arcs with second label  $\alpha$ , the process will be deadlock-free iff all the cycles of G' are either of Type 1 or of Type 2.

Figure 5 represents the graph G' for our previous example.



It has two trivial cycles, one of type 1, one of type 2.

We leave to the reader the (easy) proof of the property.

<sup>\*</sup>) Thanks are due to L. Hyafil for his active contribution to the solution of this problem.

In fact, it is possible to show that if a process can be represented by a finite system of equations of the form  $E_i$  above, then the property of absence of deadlock is provable within the logic. We will not discuss this further, partly because the interest of the general proof techniques presented here lies precisely in the fact that they apply to processes which do not have a finite number of "states".

In order to illustrate this point we will sketch the proof of the absence of deadlock of the process represented in Figure 6. This process is an adaptation of an example of Hoare [14].

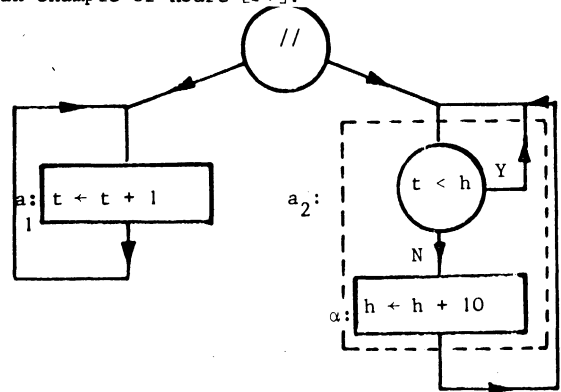


Figure 6 : an alarm clock.

Roughly speaking, one can think of t as being the time, and h as being the hour at which an alarm will ring. After the alarm has rung, h is reset to some later hour, here h + 10. The expected behavior of this system is that the alarm should ring an infinite number of times.

Using the same techniques as above, the problem can be formulated as follows :

Prove that :

- (19)  $\mathcal{F}(\omega) \vdash \text{isint}(t) :: \text{isint}(h) :: X(t, h, \omega) \equiv \alpha^*$ , where  $X(t, h, \omega)$  is defined by :
- (20)  $X(t, h, \omega) \leq \text{hd}\omega \rightarrow X(t+1, h, \text{tl}\omega)$ ,  
 $t < h \rightarrow X(t, h, \text{tl}\omega)$ ,  
 $\alpha.X(t, h+10, \text{tl}\omega)$ .

$\mathcal{F}(\omega)$  is the property of the oracle introduced in our last example and isint is a continuous predicate characterizing the domain of the integers (See Newey [25]).

An outline of a proof is the following :  
 Rewrite (20) as :

- (21)  $\begin{cases} X(t, h, \omega) \leq \text{hd}\omega \rightarrow X(t+1, h, \text{tl}\omega), Y(t, h, \text{tl}\omega) \\ Y(t, h, \omega) \leq t < h \rightarrow X(t, h, \omega), \alpha.X(t, h+10, \omega) \end{cases}$

Then, prove :

(22)  $\vdash \text{isint}(t) :: F(\omega) :: X(t, h, \omega) \equiv Y(U(t, \omega), h, V(\omega))$ ,  
where  $F$  is the same as in (10), and  $U, V$  are defined by

$$\begin{cases} U(t, \omega) <= \text{hd}\omega \rightarrow U(t+1, \text{tl}\omega), t \\ V(\omega) <= \text{hd}\omega \rightarrow V(\text{tl}\omega), \text{tl}\omega. \end{cases}$$

This says that the process cannot stay forever in the loop  $t \leftarrow t+1$  of the left branch.  $U$  and  $V$  give the value of the parameters  $t$  and  $\omega$  upon exit of the loop.

Then, prove :

(23)  $\vdash \text{isint}(t) :: \text{isint}(h) :: T(\omega) :: F(\omega) ::$

$$X(t, h, \omega) \equiv \alpha. X(\Theta(t, h, \omega), H(t, h, \omega), \Omega(t, h, \omega)),$$

where  $T$  and  $F$  are as in (10), and  $\Theta, H, \Omega$  are defined by :

$$(24) \begin{cases} \Theta(t, h, \omega) <= U(t, \omega) < h \rightarrow \Theta(U(t, \omega), h, V(\omega)), U(t, \omega) \\ H(t, h, \omega) <= U(t, \omega) < h \rightarrow H(U(t, \omega), h, V(\omega)), h+10 \\ \Omega(t, h, \omega) <= U(t, \omega) < h \rightarrow \Omega(U(t, \omega), h, V(\omega)), V(\omega) \end{cases}$$

This says that the process cannot stay forever stuck in the innerloop of the right branch.  $\Theta, H, \Omega$  give the values of the parameters  $t, h, \omega$  upon exit of the loop.

Then one essentially applies the general theorem which we used in the last example on (18) to finish the proof.

The structure of the proof is very similar to the finite case, although the steps are much harder to prove.

At this point, we should make the following comment : the existential quantifier is not allowed to appear in admissible predicates in LCF, and so, instead of stating : " $\exists \theta, h, \omega Q(\theta, h, \omega)$ ", one must actually exhibit the  $\theta, h, \omega$  which make  $Q(\theta, h, \omega)$  true. This is the purpose of system (24), for example. Yet, the general theorem which we use at this point to deduce  $X(x) \equiv \alpha^*$  from  $X(x) \equiv \alpha X(g(x))$  does not require  $g$  to be recursive. It is not yet clear whether this really complicates the proof, but it seems to be a potential problem for LCF-like systems.

#### 4 Other properties

There are many other properties of simple parallel processes which can be formalized and proved within our system, such as, for instance, determinacy and equivalence.

Let us take the following example : we will show that if an assignment  $A$  commutes with all the operations of a sequential program  $P$ , then the result of executing  $A$  and  $P$  in parallel is the same as the result of executing  $P$  after  $A$ .

Let us suppose that  $A$  is the assignment  $s \leftarrow fs$ . The commutativity property is expressed by :

$$\mathcal{C}(A, P) \begin{cases} \forall l \in L, s \in S : \sigma(l, fs) \equiv f(\sigma(l, s)) \\ \forall l \in L, s \in S : \text{Succ}(l, fs) \equiv \text{Succ}(l, s) \end{cases}$$

We also need the hypothesis that the oracle is defined, which we can state as  $\mathcal{D}(\omega) \equiv \text{tt}^*$ , where :

$$\mathcal{D}(\omega) <= \delta(\text{hd}\omega) \cdot \mathcal{D}(\text{tl}\omega).$$

$\delta$  is the same monotonic "isdefined" predicate used in Eq. (14).

Then the property to prove can be stated as :

$$\mathcal{C}(A, P), \mathcal{D}(\omega) \equiv \text{tt}^* \vdash M(a, fs) \equiv \mathcal{M}(A, a, s, \omega)$$

where " $a$ " is the label of the starting operation of  $P$  and  $M, \mathcal{M}$  are defined by :

$$(25) \quad M(l, s) \equiv \text{Stop}(l) \rightarrow s, M(\text{Succ}(l, s), \sigma(l, s))$$

$$(26) \quad \mathcal{M}(A, l, s, \omega) \equiv \text{Stop}(l) \rightarrow fs,$$

$$\text{hd}\omega \rightarrow M(l, fs),$$

$$\mathcal{M}(A, \text{Succ}(l, s), \sigma(l, s), \text{tl}\omega).$$

(25) and (26) are the appropriate instances of Eq. (1) and (4) respectively for defining the sequential and parallel state transformation functions of  $P$  and  $A//P$  respectively.

We show :

$$\mathcal{C}(A, P) \vdash M(l, fs) \supseteq m(A, l, s, \omega)$$

by computational induction on  $m$  using (26), and :

$$\mathcal{C}(A, P), \mathcal{D}(\omega) \equiv \text{tt}^* \vdash m(l, fs) \subseteq \mathcal{M}(A, l, s, \omega), m(l, s) \subseteq M(l, s)$$

by computational induction on  $m$  using (25). These two properties imply the desired result. The details of the proof are straightforward.

## II. Parallel Processes

In this part, we attempt to formalize the semantics of more general parallel processes, and to give methods for proving properties about them.

The processes that we now consider generalize those of the previous part in that they can have a varying, possibly unbounded, degree of parallelism. In particular, calls of processes within processes, possibly recursively, are allowed. However, the various parts of a process still communicate via the memory.

In section 5 we present the semantics we use, and in section 6 we show examples of proofs, including the proof of the correctness of a parallel recursive program to compute the factorial function.

## 5 Semantics

Following Milner [23], we want to think of a process as an object which operates on a memory state with the help of an oracle : it first performs an action, which is an indivisible operation, and then returns a new process. Since it may have used some of the oracle to determine which was the first action to perform, it must also return a new oracle. One way to express this is to consider the semantics of a process to be in the functional domain  $P$  satisfying the equation :

$$(27) \quad P = S \rightarrow \Omega \rightarrow S \times \Omega \times P.$$

The justification that such a domain  $P$ , consisting of continuous functions, can be constructed comes from the fundamental work of Scott [26],[27],[28].

Given a state  $s \in S$  and an oracle  $\omega \in \Omega$ , a process  $p \in P$  will return a triple  $\langle s', \omega', p' \rangle$  with  $s' \in S$ ,  $\omega' \in \Omega$ ,  $p' \in P$ , i.e. :

$$ps\omega \equiv \langle s', \omega', p' \rangle$$

If we denote  $( )_1$ ,  $( )_2$ ,  $( )_3$  the three projections functions, then

$$s' \equiv (ps\omega)_1, \omega' \equiv (ps\omega)_2, p' \equiv (ps\omega)_3, \text{ and}$$

we can write :

$$(28) \quad p \equiv \lambda s \lambda \omega \langle (ps\omega)_1, (ps\omega)_2, (ps\omega)_3 \rangle.$$

In fact, this structure turns out not to be quite sufficient. One needs to have a special process, called STOP, and to be able to test the equality  $p = \text{STOP}$  with a continuous predicate  $\text{Stop} : P \rightarrow \{\text{tt}, \text{ff}, \perp\}$ . For this to be possible, STOP must be an isolated point in  $P$ . We obtain this by slightly modifying (27) to

$$(29) \quad P = \{\text{STOP}\} + [S \rightarrow \Omega \rightarrow S \times \Omega \times P].$$

This now says that a process is either the undefined element  $\perp$  or STOP or a triple of the form (28). It is then possible to axiomatize the continuous predicate Stop in such a way that  $\text{Stop}(\perp) \equiv \perp$ ,  $\text{Stop}(\text{STOP}) \equiv \text{tt}$ , and, for any other process  $p$  :

$$\text{Stop}(p) \equiv \text{ff}.$$

Let us define two useful combinators on processes : the first one is the composition combinator. It is a mapping  $*$  :  $P \times P \rightarrow P$  which, given two processes  $p$  and  $q$ , yields a process  $p * q$ , the sequential composition of  $p$  and  $q$ , defined by :

$$(30) \quad p * q \leq \text{Stop}(p) \rightarrow q, \\ \lambda s \lambda \omega \langle (ps\omega)_1, (ps\omega)_2, (ps\omega)_3 * q \rangle.$$

The second one is the parallel combinator, a mapping  $//$  :  $P \times P \rightarrow P$ , defined by :

$$(31) \quad p // q \leq \text{Stop}(p) \rightarrow q, \\ \text{Stop}(q) \rightarrow p, \\ \lambda s \lambda \omega. \text{hd}\omega \rightarrow \langle (p\text{stl}\omega)_1, (p\text{stl}\omega)_2, (p\text{stl}\omega)_3 // q \rangle, \\ (q\text{stl}\omega)_1, (q\text{stl}\omega)_2, p // (q\text{stl}\omega)_3 \rangle.$$

Intuitively, what  $p // q$  does, when neither  $p$  nor  $q$  are Stop, is ask the oracle which component must get control first,  $p$  or  $q$ . If it is  $p$ , then  $p // q$  does the first action of  $p$  and returns a process which consists of the new process of  $p$  in parallel with  $q$ . If it is  $q$ , then the symmetric course of action takes place.

Let us also define a state transformation function on a process, i.e a function  $M : P \times S \times \Omega \rightarrow S$ . Given a process  $p$ , a state  $s$  and an oracle  $\omega$ , we define  $M[p]s\omega$  as follows :

$$(32) \quad M[p]s\omega \leq \text{Stop}(p) \rightarrow s, \\ M[(ps\omega)_3](ps\omega)_1(ps\omega)_2.$$

Intuitively,  $M[p]s\omega$  represents the resulting state of the memory after  $p$  has finished its execution on starting state  $s$  and oracle  $\omega$ .

Let us illustrate these definitions with some simple examples. We will represent a simple assignment  $a := s + f(s)$  by the process  $a \equiv \lambda s \lambda \omega \langle fs, \omega, \text{STOP} \rangle$ . Thus :  $(as\omega)_1 \equiv fs$ ,  $(as\omega)_2 \equiv \omega$ ,  $(as\omega)_3 \equiv \text{STOP}$ , and (32) yields  $M[a]s\omega \equiv M[\text{STOP}]fs\omega \equiv fs$ , as expected.

Also, if  $a$  is the assignment  $s + f(s)$ ,  $b$  the assignment  $s + g(s)$ , we can use (30) to get  $a * b$  :

$$a * b \equiv \lambda s \lambda \omega \langle fs, \omega, \text{STOP} * b \rangle \equiv \lambda s \lambda \omega \langle fs, \omega, b \rangle.$$

Then :  $M[a*b]s\omega \equiv M[b]fs\omega \equiv gfs$ , as expected.

Now, let us see how to use (31) to get  $a // b$  :

$$a // b \equiv \lambda s \lambda \omega. \text{hd}\omega \rightarrow \langle fs, \text{tl}\omega, \text{STOP} // b \rangle, \\ \langle gs, \text{tl}\omega, a // \text{STOP} \rangle \\ \equiv \lambda s \lambda \omega. \text{hd}\omega \rightarrow \langle fs, \text{tl}\omega, b \rangle, \\ \langle gs, \text{tl}\omega, a \rangle.$$

We can write this

$$\begin{aligned} ((a // b)s\omega)_1 &\equiv \text{hd}\omega \rightarrow \text{fs}, \text{gs} \\ ((a // b)s\omega)_2 &\equiv \text{hd}\omega \rightarrow \text{tl}\omega, \text{tl}\omega \\ ((a // b)s\omega)_3 &\equiv \text{hd}\omega \rightarrow b, a \end{aligned}$$

because of the axioms  $(\perp)_1 \equiv (\perp)_2 \equiv (\perp)_3 \equiv \perp$ .

Then we have :

$$\begin{aligned} M[a//b]s\omega &\equiv M[(a//b)s\omega_3]((a//b)s\omega)_1((a//b)s\omega)_2 \text{ by (32)} \\ &\equiv \text{hd}\omega \rightarrow M[b]fs \text{ tl}\omega, \\ &\quad M[a]gs \text{ tl}\omega \\ &\equiv \text{hd}\omega \rightarrow \text{gfs}, \text{fgs}, \text{ as expected.} \end{aligned}$$

Here we have used the fact that  $M[\perp]s\omega \equiv \perp$ , which comes from  $\text{Stop}(\perp) \equiv \perp$ , and the fact that  $\text{Stop}(a//b) \equiv \text{ff}$ .

At this point, let us observe that the simple parallel processes considered in Part I are indeed special cases of those we are now modeling. To any sequential program  $S$  of paragraph 1.1, we associate the element  $\bar{S}$  of  $L \rightarrow P$  defined by :

$$\begin{aligned} \bar{S}(1) &\Leftarrow \text{STOP}(1) \rightarrow \text{STOP}, \\ &\quad \lambda s \lambda \omega < \sigma(1, s), \omega, \bar{S}(\text{Succ}(1, s)) >, \end{aligned}$$

where  $\sigma(1, s)$  and  $\text{Succ}(1, s)$  are the next state and next operation functions of the program  $S$ , and  $L$  is the label alphabet of  $S$ . Thus  $\bar{S}(1)$  is a process  $\in P$  in our present sense.

Then one can show that the state transformation function  $M$  as defined by (1) in Part I (denote it  $M_I$ ) is the same as the current state transformation function  $M$  defined by (32), i.e :

$$M_I(1, s) \equiv M(\bar{S}(1)) s\omega$$

The proof is straightforward by parallel computational induction on the two definitions.

In a similar way, one can now obtain the state transformation  $\mathcal{M}$  of a simple parallel process, as defined by (4), from the parallel combinator  $//$  and the state transformation function  $M$  of the present theory. More precisely :

$$\mathcal{M}(1_1, 1_2, s, \omega) \equiv M[\bar{S}_1(1_1) // \bar{S}_2(1_2)] s\omega$$

one could also define tracing functions in the present context and relate them to the tracing functions of part I in the same fashion.

## 6 Proofs

In the remainder of this paper we discuss some proofs of properties of parallel processes.

As an example of proof using computational induction, let us show the associativity of the composition of processes, namely :

$$(33) \quad \forall pqr : (p * q) * r \equiv p * (q * r).$$

The proof proceeds by computational induction on the two occurrences of '\*' designated by an arrow, using the recursive definition (30).

The base case is trivial.

For the inductive step, assume :

$$\forall pqr \text{ f}(p, q) * r \equiv \text{f}(p, q * r).$$

Show :

$$(34) \quad [\text{Stop}(p) \rightarrow q, \lambda s \lambda \omega < (ps\omega)_1, (ps\omega)_2, \text{f}(ps\omega)_3, q >] * r \equiv [\text{Stop}(p) \rightarrow q * r, \lambda s \lambda \omega < (ps\omega)_1, (ps\omega)_2, \text{f}(ps\omega)_3, q * r >]$$

We do case analysis on  $\text{Stop}(p)$  :

$\text{Stop}(p) \equiv \perp$  : trivial.

$\text{Stop}(p) \equiv \text{tt}$  : (34) is reduced to  $q * r \equiv q * r$ .

$\text{Stop}(p) \equiv \text{ff}$  :

$$\begin{aligned} \text{l.h.s.} &\equiv [\lambda s \lambda \omega < (ps\omega)_1, (ps\omega)_2, \text{f}((ps\omega)_3, q) >] * r \\ &\equiv \lambda s \lambda \omega < (ps\omega)_1, (ps\omega)_2, \text{f}((ps\omega)_3, q) * r > \text{ Def. of } * \\ &\equiv \lambda s \lambda \omega < (ps\omega)_1, (ps\omega)_2, \text{f}((ps\omega)_3, q * r) > \text{ Ind. Hyp.} \\ &\equiv \text{r.h.s.} \end{aligned}$$

□

Let us now indicate another useful tool for proofs, the structural induction principle on  $P$ . This principle can be justified within the system (for admissible predicates), by using computational induction on the recursive definition of the domain as a retract (characteristic function) in the general Scott domain  $D^\infty$ . (See Scott [27],[28]).

However, we will give here an informal presentation, without formal justification.

The domain  $P$  satisfying  $P = \{\text{STOP}\} + [S \rightarrow \Omega \rightarrow S \times \Omega \times P]$  can be built recursively via a sequence of approximations, much like the fixpoint of an ordinary recursive definition. Here, we have :

$$P_0 = \{\text{STOP}, \perp\} \text{ and, for } n > 0 :$$

$$P_n = \{\text{STOP}\} + [S \rightarrow \Omega \rightarrow S \times \Omega \times P_{n-1}].$$

Then  $P = \bigsqcup P_n$ . Now for every  $p \in P$  and for every  $n \geq 0$  it is possible to define the  $n$ -th projection of  $p$ ,  $p_n \in P_n$ . If  $p \in P_n$  then  $p_n \equiv p$ .

One can show that, for  $n > 0$  :

$$(35) \quad p_n \equiv \lambda s \lambda \omega < (ps\omega)_1, (ps\omega)_2, [(ps\omega)_3]_{n-1} >.$$

This suggests the following proof idea : to prove a property of  $p \in P$ , prove it for all its finite

projections. For this to be a valid proof, the property  $\mathcal{F}$  to prove must be admissible (See Manna, Ness, Vuillemin [19]), i.e one must have  $\forall n \mathcal{F}(p_n) \supset \mathcal{F}(p)$ . In particular, LCF well formed formulae are admissible.

For such properties, the structural induction principle can be stated in the following form :

$$(36) \quad \begin{array}{l} \vdash \mathcal{F}(\perp) \\ \vdash \mathcal{F}(\text{STOP}) \\ \forall p \mathcal{F}(jp) \vdash \forall p \mathcal{F}(\lambda s \lambda \omega \langle (ps\omega)_1, (ps\omega)_2, j(ps\omega)_3 \rangle) \\ \hline \forall p \mathcal{F}(p) \end{array}$$

Notice however that for finite p's, this induction principle is valid for any property  $\mathcal{F}$ . In other words, from the above premisses, one can infer :  $(\forall \text{ finite } p) \mathcal{F}(p)$ , whether or not  $\mathcal{F}$  is admissible. It is not clear yet how one can justify this within LCF, but it is certainly a valid deduction rule which can be added to the system.

Example : As an example of the use of structural induction, let us prove :

$$(37) \quad \forall p \quad p * \text{STOP} \equiv p$$

(i)  $\mathcal{F}(\perp)$  : trivial

(ii)  $\mathcal{F}(\text{STOP})$  : trivial

(iii) assume :  $\forall p(jp) * \text{STOP} \equiv jp$ .

show :  $q * \text{STOP} \equiv q$ ,

where  $q \equiv \lambda s \lambda \omega \langle (ps\omega)_1, (ps\omega)_2, j(ps\omega)_3 \rangle$ . We have :

$$\begin{aligned} q * \text{STOP} &\equiv \lambda s \lambda \omega \langle (qs\omega)_1, (qs\omega)_2, (qs\omega)_3 * \text{STOP} \rangle \\ &\equiv \lambda s \lambda \omega \langle (ps\omega)_1, (ps\omega)_2, j(ps\omega)_3 * \text{STOP} \rangle \\ &\equiv \lambda s \lambda \omega \langle (ps\omega)_1, (ps\omega)_2, j(ps\omega)_3 \rangle \text{ Ind. Hyp.} \\ &\equiv q \end{aligned}$$

We will now introduce several properties of processes and prove some facts about them. In the sequel,  $a$  will denote the individual action  $\lambda s \lambda \omega \langle fs, \omega, \text{STOP} \rangle$ .

Commutativity. We define a total predicate  $Q(a,p)$

with the following intuitive meaning : "  $a$  commutes with all the individual actions of  $p$ ". We first define it for finite  $p'_s$  by induction :

$$(38) \quad \begin{array}{l} \vdash Q(a, \text{STOP}), \vdash Q(a, \perp), \\ Q(a,p) \iff \forall s \forall \omega \{ [f(ps\omega)_1 \equiv (pfs\omega)_1] \wedge [(ps\omega)_2 \equiv (pfs\omega)_2] \wedge \\ \quad [(ps\omega)_3 \equiv (pfs\omega)_3] \wedge Q(a, (ps\omega)_3) \} \end{array}$$

To justify that this defines  $Q$  for all finite  $p'_s$  one can use (36) with the remark on finite  $p'_s$ , or observe directly that  $p \in P_n \Rightarrow (ps\omega)_3 \in P_{n-1}$  and use mathematical induction on  $n$ .

Then, for arbitrary  $p$ , define  $Q(a,p) \iff \forall n Q(a, p_n)$ . This definition is consistent in that it reduces to (38) for finite  $p'_s$ .

$Q$  has the following property :

Lemma 1 :

Let  $a, b$  be individual actions. Then, for every finite  $p$  :

$$Q(a,b) \wedge Q(a,p) \supset Q(a,b//p).$$

Proof :

The proof is easy by structural induction (36) on  $p$ , and is left to the reader.  $\square$

Strong Determinacy : Following the technique used in the previous paragraph, we now define a total predicate  $D(p)$  with the intuitive meaning : "the state transformation function of  $p$  does not depend on its oracle, provided it is defined".  $D(p)$  is defined as follows :

$$(39) \quad \begin{array}{l} \vdash D(\perp), \vdash D(\text{STOP}), \\ D(p) \iff \forall s \forall \omega \forall \omega' [ \text{Def}(\omega) \wedge \text{Def}(\omega') \supset M[p]s\omega \equiv M[p]s\omega' ] \\ \quad \wedge \forall s \forall \omega D((ps\omega)_3) \end{array}$$

for finite  $p'_s$ , and  $D(p) \iff \forall n D(p_n)$  for arbitrary  $p'_s$ .

$\text{Def}(\omega)$  is the predicate  $\mathcal{D}(\omega) \equiv tt^*$ , where

$\mathcal{D}(\omega) \iff \delta(\text{hd}\omega). \mathcal{D}(\text{tl}\omega)$  as defined in Section 4.

Def-preserving: We now define a total predicate  $\Delta(p)$  with the intuitive meaning "if  $p$  is given a defined oracle, it returns a defined oracle".  $\Delta(p)$  is defined as follows :

$$\begin{array}{l} \vdash \Delta(\text{STOP}), \\ \Delta(p) \iff \forall s \forall \omega [ \text{Def}(\omega) \supset \text{Def}((ps\omega)_2) ] \wedge \forall s \forall \omega \Delta((ps\omega)_3) \end{array}$$

for finite  $p'_s$ , and  $\Delta(p) \iff \forall n \Delta(p_n)$  for arbitrary  $p$ .

We can now state the following properties :

Lemma 2

$$\begin{array}{l} \forall p \quad Q(a,p), D(p), \Delta(p) \vdash \\ \quad \forall s \forall \omega \{ \text{Def}(\omega) \supset M[a//p]s\omega \equiv M[a*p]s\omega \} \end{array}$$

Informally, this states that under certain restrictions on  $p$  and if  $a$  commutes with all the individual actions of  $p$ , then executing  $a$  in parallel with  $p$  is equivalent to executing  $a$  before  $p$ , as far as final results are concerned.

The property to be proved is admissible and we can use computational induction on  $M$ , much in the same way as in section 4 where an analogous result was obtained for simple parallel processes. The proof here is slightly longer, although quite easy.  $\square$

**Lemma 3 :** For all finite  $p$  :

$$Q(a,p) \wedge D(p) \wedge \Delta(p) \supset D(a//p).$$

The proof is simple by structural induction on  $p$ .  $\square$

**Lemma 4 :** For all finite  $p$  :

$$\Delta(p) \supset \Delta(a//p).$$

Again the proof is easy and we omit it.  $\square$

### The recursive factorial

We now have all the basic tools to show the correctness, of the recursive factorial process shown in Figure 7.

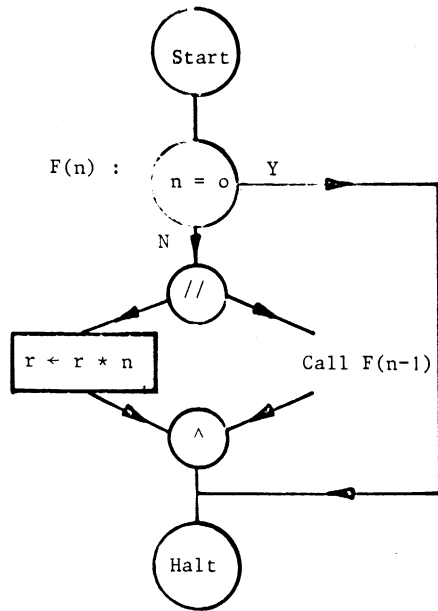


Figure 7 : the recursive factorial <sup>\*</sup>

In this process,  $n$  is a non negative integer parameter,  $r$  is a global variable and the operation  $r \leftarrow r * n$  is not interruptible.

It is possible to describe formally a language for parallel programs and then use the formal semantics of the language to obtain the mathematical process in  $P$  which represents such a program. This is however outside the scope of the present paper, and we will assume that the formal semantics of the parallel flowchart language used in Figure 7 is such that the corresponding object in  $P$  is described by the following recursive definition :

<sup>\*</sup>) This process is an adaptation of an example proposed by Milner [23].  $*$  denotes the multiplication on the integers.

$$(40) F(n) \Leftarrow \lambda r. \lambda \omega \langle r, \omega, [n=0 \rightarrow \text{STOP}, p(n)//F(n-1)] \rangle,$$

where  $p(n)$  is the individual action :

$$p(n) \equiv \lambda z \lambda \omega \langle r * n, \omega, \text{STOP} \rangle$$

We assume here that the state  $s$  is reduced to  $r$ , and that  $r$  is a non negative integer.

We will first show by mathematical induction on  $n$  that  $F(n)$  is finite. We need :

**Lemma 5 :** Let  $a$  be an individual action :

If  $p \in P_n$  then  $a//p \in P_{n+1}$ .

**Proof :** By mathematical induction on  $n$ . The base case is trivial. Now for  $n > 0$ , assume  $p \in P_{n-1} \Rightarrow a//p \in P_n$ .

Let  $p \in P_n$ . Then

$p \equiv p_n \equiv \lambda s \lambda \omega \langle (ps\omega)_1, (ps\omega)_2, [(ps\omega)_3]_{n-1} \rangle$ , by (35).  
Since  $(ps\omega)_3 \equiv [(ps\omega)_3]_{n-1}$ , we get that  $(ps\omega)_3 \in P_{n-1}$ .

Therefore, by induction hypothesis :  $a//p \in P_n$ .

Now,  $((a//p)s\omega) \equiv \text{hd}\omega \rightarrow p, a//p \text{stl}\omega)_3$ , by (31)  
 $\equiv \text{hd}\omega \rightarrow p_n, [a//p \text{stl}\omega)_3]_n$ .

Hence  $((a//p)s\omega)_3 \in P_n$  and  $a//p \in P_{n+1}$   $\square$

As an easy consequence of this we get that, for every  $n \geq 0$ ,  $F(n) \in P_{2n+1}$ . The proof is straightforward by induction on  $n$ .

We can now prove :

**Theorem :**  $\forall n, k \geq 0, \forall \omega \in \Omega$ ,

$\text{Def}(\omega) \supset M[F(n)]k\omega \equiv k * \text{Fact}(n)$ , where  $\text{Fact}(n)$

is the factorial function :

$$\text{Fact}(n) \Leftarrow (n=0) \rightarrow 1, n * \text{Fact}(n-1).$$

**Proof :** we prove the property :

$$\forall n \forall m Q(p(m), F(n)) \wedge D(F(n)) \wedge \Delta(F(n)) \wedge$$

$$(41) \text{Def}(\omega) \supset M[F(n)]k\omega \equiv k * \text{Fact}(n),$$

by mathematical induction on  $n$ . We will only outline the proof. The base case is easy. The inductive step, for  $n > 0$ , splits into four cases :

(i)  $Q(p(m), F(n))$ .

The key step here is to prove  $Q(p(m), p(n)//F(n-1))$  from the inductive hypothesis  $Q(p(m), F(n-1))$ . This is done by using Lemma 1 and the fact that  $F(n-1)$  is finite. One also needs  $Q(p(m), p(n))$  which is a direct consequence of the commutativity and associativity of the multiplication on the integers.

(ii)  $\Delta(F(n))$

This is where Lemma 4 gets used.

(iii)  $\text{Def}(\omega) \supset M[F(n)]k\omega \equiv k * \text{Fact}(n)$

Assume  $\text{Def}(\omega)$ . We have :

$$\begin{aligned}
M[F(n)]_{k\omega} &\equiv M[(F(n)k\omega)_3](F(n)k\omega)_1(F(n)k\omega)_2 \\
&\equiv M[p(n)/F(n-1)]_{k\omega}. && \text{Def (40)} \\
&\equiv M[F(n-1)]_{(k*n)\omega} && \text{Lemma 2} \\
&\equiv (k*n) * \text{Fact}(n-1) && \text{Ind. Hyp.} \\
&\equiv k * \text{Fact}(n).
\end{aligned}$$

(iv) D(F(n))

We use Lemma 3 and case iii above already proved.  $\square$

This shows both correctness and termination of the recursive factorial. The proof seem a bit complicated, and it is quite probable that it can be shortened. In fact, it may be that some form of Lemma 2 can be proved without the determinacy hypothesis.

Lemma 4 is technical and could be eliminated by adopting a slightly different formalism for processes. Actually, the formalism that we have at present somehow seems too general, in that it allows processes which can modify oracles in arbitrary ways, whereas intuition says that a process always returns some right portion of the oracle i.e, we should have  $\omega \equiv u \circledast (ps\omega)_2$ , where  $\circledast$  denotes the string concatenation. Furthermore, intuition says that  $(ps\omega)_1, (ps\omega)_2$  and  $(ps\omega)_3$  should not depend on that portion of the oracle which is at the right of u. These suggest possible improvements on the present formalism.

The above proof also calls for some comments on the logic. In order to carry it out mechanically, in the present form, one essentially needs to extend LCF by adding the structural induction principle for finite p and also allowing the possibility to define wffs which are not admissible. Of course, the system would have to check that such wffs only get used in proofs for finite p's. This might be convenient for some proofs even if they can be carried out also in pure LCF.

### Conclusion

This paper shows examples of proofs about parallel processes which can be carried out completely formally. The proofs generally appear to be more complicated than proofs about sequential programs. Therefore the need for complete formality appears even greater than for sequential programs. Proofs carried out manually are probably just as likely to contain bugs as the original code. Also the need for structure in the proof is essential, if we want to be able to prove meaningful things about sizeable parallel programs.

Evidently there is need for more research both at the level of semantics of parallel processes and at the level of proof structure. The present semantics should be extended to handle processes with communication lines, and possibly also modified to forbid "unnatural" behavior of processes, as suggested at the end of last section. Also, work should go into proving general properties of parallel programs, such as sufficient conditions for using facts about components of a program to deduce facts about the whole program.

### Acknowledgements

We are indebted to R.Milner for many stimulating discussions and ideas. It is clear that his views on various aspects of theory of computation have greatly influenced this work. Thanks are also due to G. Kahn who commented on an early version of the paper.

### References

- [1] Aschroft, E.A., "Proving Assertions about Parallel Programs". Report CS-73-01, Department of AACS, University of Waterloo (1973).
- [2] Aschroft, E.A., and Z.Manna, "Formalization of Properties of Parallel Programs", Machine Intelligence 6, Edinburgh University Press (1970).
- [3] Bekic, H., "Towards a Mathematical Theory of Processes", Technical report TR 25-125, IBM Laboratory, Vienna (1971).
- [4] Bredt, T.H., "The mutual Exclusion Problem", STAN-CS-70-173, Stanford University (1970).
- [5] Cadiou, J.M., "Recursive Definitions of Partial Functions and their Computations", Ph.D. Thesis, STAN-CS-266-72, Stanford University (1972).
- [6] Dijkstra, E.W., "Cooperating Sequential Processes", University of Eindhoven Report EWD 123 (1965).
- [7] Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control", CACM (8) p. 569 (1965).
- [8] Elspas, B., M.W. Green, K.N. Levitt and R.J. Waldinger, "Research in Interactive Program Proving Techniques", Stanford Research Institute Report (1972).

- [9] Floyd, R.W., "Assigning Meanings to Programs", Proceedings of a symposium in Applied Mathematics, vol 19 pp. 19-32, Mathematical Aspects of Computer Science, American Mathematical Society (1967).
- [10] Habermann, A.N., "On a Solution and a Generalization of the Cigarette Smokers' Problem", Technical report, Department of Computer Science Carnegie-Mellon University (1972).
- [11] Habermann, A.N., "Synchronization of Communicating Processes", CACM (15) 171-176 (1972).
- [12] Hoare, C.A.R., "Proofrules for Parallel Processes", Lectures given at the Matematisk Institutt, Oslo (1972).
- [13] Hoare, C.A.R., "An axiomatic Basis for Computer Programming", CACM (12) 576-580 (1969).
- [14] Hoare, C.A.R., "Monitors: an Operating System Structuring Concept", Internal Report, The Queens University of Belfast (1973).
- [15] Kahn, G., "A Preliminary Theory for Parallel Programs", Rapport n°6, IRIA-LABORIA (1973).
- [16] Knuth, D., "Additional Comments on a Problem in Concurrent Programming Control" CACM (9) (1966).
- [17] Lauer, H.C., "Correctness in Operating Systems", Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University (1972).
- [18] Manna Z., "The correctness of Programs", JCSS(3) 119-127 (1969).
- [19] Manna, Z., S. Ness, J. Vuillemin, "Inductive Methods for Proving Properties of Programs", Proceedings of an ACM Conference on Proving Assertions about Programs, Las Cruces, N.M (1972).
- [20] McCarthy, J., "A Basis for a Mathematical Theory of Computation", in Computer Programming and Formal systems, North Holland Publ. Co, Amsterdam (1963).
- [21] Milner, R., "Implementation and Applications of Scott's Logic for Computable Functions", Proceedings of an ACM Conference on Proving Assertions, about Programs, Las Cruces, N.M. (1972).
- [22] Milner, R., "Logic for Computable Functions- Description of a Machine Implementation", Artificial Intelligence Memo # 169, Computer Science Department, Stanford University (1972).
- [23] Milner, R., "An Approach to the Semantics of Parallel Programs", Unpublished Memo, Computer Science Department, University of Edinburgh (1973).
- [24] Milner, R., and R.W. Weyhrauch, "Proving Compiler Correctness in a Mechanized Logic", Machine Intelligence 7, Edinburgh University Press (1972).
- [25] Newey, M., "Axioms and Theorems for Integers, Lists and finite Sets in LCF", STAN-CS-330, Stanford University (1973).
- [26] Scott, D., "Outline of a Mathematical Theory of Computation", Proceedings of the 4th Annual Princeton Conference on Information Science and Systems (1970).
- [27] Scott, D., "Data Types as Lattices", Unpublished Lecture Notes, Amsterdam (1972).
- [28] Scott, D., "Continuous Lattices", Proceedings of the 1971 Dalhousie Conference, Springer Lecture Notes Series, Springer-Verlag, Heidelberg (1971).
- [29] Scott, D., and C. Strachey, "Towards a Mathematical Semantics for Computer Languages" Proceedings of the Symposium on Computers and Automata, Microwave Research Institute, Symposium Series, vol 21, Polytechnic Institute of Brooklyn (1971).
- [30] Strachey, C., "Continuations, a Mathematical Semantics which can deal with full Jumps", Unpublished Memo, Programming Research Group, Computing Laboratory Oxford University (1973).
- [31] Vuillemin, J., "Proof Techniques for Recursive Programs", Ph.D. Thesis, Computer Science Department, Stanford University (to appear) (1973).