The Semantics of a Simple Language for Parallel Programming

Gilles KAHN
IRIA-Laboria, Domaine de Voluceau, 78150 Rocquencourt, France
and
Commissariat à l'Energie Atomique, France

ABSTRACT

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for more formal (*i.e.* mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

The features of our mini-language are exhibited on the sample program S on Figure 1. The conventions are close to Algol1 and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type *integer channel* are declared at line (1), and for any simple type σ (boolean, real, etc. . .) we could have declared a σ *channel*. Then *processes* f, g and h are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes : at line (2) f is stated to communicate via two input lines that can carry integers, and one similar output line.
The body of a process is an usual Algol program except for invocation of *wait* until something on an input line (*e.g.* at (4)) or *send* a variable *on* a line of compatible type (*e.g.* at (5)). The process stays blocked on a *wait* until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.
In others words, processes communicate via first-in first-out (fifo) queues.
Calling instances of the processes is done in the body of the main program at line (6) where the actual names of he channels are bound to the formal parameters of the processes. The infix operator *par* initiates the concurrent activation of the processes. Such a style of programming is close to may systems using EVENT mechanisms ([1, 2, 3, 4]). A pictorial representation of the *program* is the schema P on Figure 2, where the nodes represent processes and the arcs communication channels between these processes.
What sort of things would we like to prove on a program like S? Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out (at line (7)) an alternating sequence of 0's and 1's forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole systems would stop.
The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

```
Begin
(1) Integer channel X, Y, Z, T1, T2;

(2) Process f(interger in U,V; interger out W);
    Begin integer I ; logical B;
    B := true;
    Repeat Begin
(4)      I := if B then wait(U) else wait (V);
(7)      print (I);
(5)      send I on W;
         B := not B;
    End;
    End;

    Process g(integer in U ; integer out V, W);
    Begin integer I; logical B;
    B := true;
    Repeat Begin
      I := wait (U);
      if B then send I on V else send I on W :
         B := not B;
    End;
    End;

(3) Process h(integer in U; integer out V;
              integer INIT);
    Begin integer I;
    send INIT on V;
    Repeat Begin
      I := wait (U);
      send I on V;
    End;
    End;

Comment : body of mainprogram;
(6) f(X,Y,Z) par g(X,T1,T2) par h(T1,Y,0)
      par h(T2,Z,1);
End;
```

Figure 1: *Sample parallel program S.*

2. PARALLEL COMPUTATION

Informally speaking, a parallel computation is organized in the following way: some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along

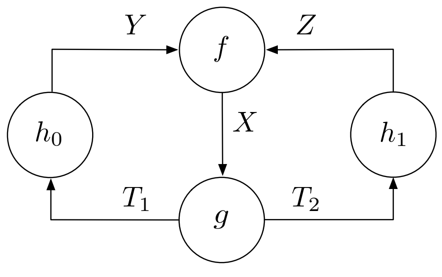


Figure 2: ***The schema P for the program S.***

its input lines, using some memory of its own, to produce output on some or all of its output lines. It is assumed that:

i) Communication lines are the *only way* by which computing station may communicate.

ii) A communication line transmits information within an unpredictable but *finite* amount of time.

Restrictions are imposed on the behaviour of computing stations:

iii) At any given time, a computing station is either computing or waiting for information on *one* of its input lines.

iv) Each computing station follows a sequential program. (We call here *sequential* program what is usually called *program* elsewhere).

Remarks: first, since several computing stations may be computing simultaneously, this model indeed exhibits some form of parallelism. Second, restriction iii) means that a computing station cannot be waiting on data coming from one *or* another of its input lines, or alternatively that no two computing stations are allowed to *send* data on the same *channel*. Third, we *do not* restrict the computing stations to have a finite memory.

The reader who is mathematically inclined wan think of set of Turing machines connected via-one-way tapes, where each machine can use its own working tape.
We formalize now the notion of parallel computation introduced above.

2.1 Syntax

A parallel program *schema* is an oriented graph with labeled nodes and edges, together with some supplementary edges (see Figure 3): incoming edges with only end vertices, meant to represent the input lines, and outcoming edges, with only origin vertices, the output lines.

2.2 Semantics

2.2.1 Outline

Edges in a schema are thought of as pipes: each edge is able to carry data of a given type D (*e.g.* : integer, boolean,

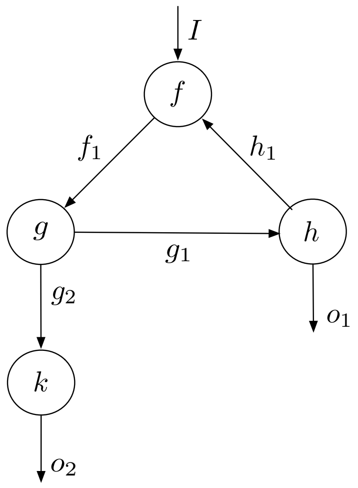


Figure 3: ***A parallel program schema.***

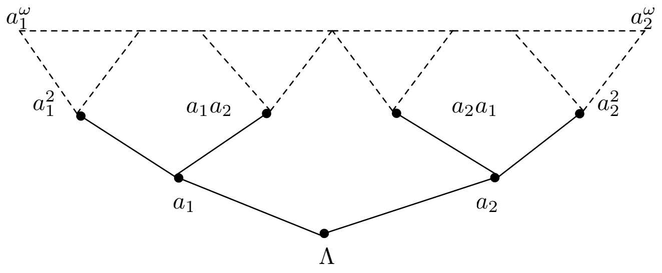


Figure 4: ***The complete partial order D^ω, when $D = \{a_1, a_2\}$.***

pointer, procedure, etc...).
An observer placed on the line witnesses its traffic, a (possibly infinite) sequence of objects of type D: it is called the *history* of the line. Since a computing station has its own memory, it is not a partial function from the domains of the inputs into the domain of the outputs, but rather a function from the *histories* of its input line into the histories of its output lines.

2.2.2 Sequence domains

Let D^ω be the set of finite or denumerably infinite sequences of elements over a set D. In D^ω we include the empty sequence Λ. The relation $\subseteq$ defined by $X \subseteq Y$ iff X is an initial segment of Y is a partial order on D^ω. The minimal element of D^ω is Λ. Any increasing chain ξ in D^ω: $X_1 \subseteq X_2 \subseteq \cdots \subseteq X_n \subseteq \cdots$ has a least upper bound which we call $\lim\limits_{D^\omega} \xi$. Hence, D^ω is a *complete partial order* (c.p.o.).

2.2.3 Domain of interpretation

To each edge e in a schema, we associate a set D_e, the type of the objects it may carry. The history of line e is then an element of D_e^ω.

2.2.4 Continuous mappings

A mapping f from a complete partial order A into a complete partial order B is *continuous* iff, for any increasing chain a of A

$$f(\lim_A a) = \lim_B f(a)$$

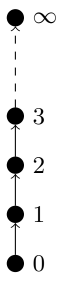
Note that a continuous mapping is also monotonic, *i.e.* $x \subseteq y \Rightarrow f(x) \subseteq f(y)$. The following mappings: F (for *first*), R (for *remainder*) and A (for *Append*) are examples of continuous mappings:

- F: to any sequence x in D^ω, F associates the (unit length) sequence constituted of the leftmost element of x.
- R: to any sequence x in D^ω, R associates the sequence of the right of its leftmost element.
- A: takes two arguments L_1 and L_2 in D^ω to produce the sequence: leftmost element of L_1 followed by L_2.

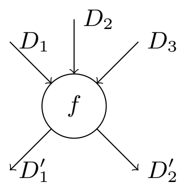
More precisely, F, R, and A obey the axioms[*]:

1) $R(\Lambda) = \Lambda$,	2) $A(\Lambda, X) = \Lambda$
3) $A(X, \Lambda) = F(X)$	4) $F(A(X,Y)) = F(X)$
5) $A(F(X), R(X)) = X$	6) $(X = \Lambda) \vee R(A(X,Y)) = Y$

For properties such as deadlock, we shall need to talk *formally* about the length of a sequence. An elegant way to do so within our formalism is to take the integers with their usual order and complete them with an extra element ∞ to obtain the complete partial order $\overline{\mathbb{N}}$ (see Figure 5(a)).
The mapping *length* from D^ω to $\overline{\mathbb{N}}$ which maps a sequence into its length is continuous; note also that *addition* in $\overline{\mathbb{N}}$ is continuous.



(a) *The c.p.o.* $\overline{\mathbb{N}}$.



(b) *Computing station.*

Figure 5: Schema of c.p.o and computing station.

2.2.5 *Computing stations*

We are now ready to interpret the nodes in a parallel schema. To each node with input lines carrying data in $D_1 \times D_2 \times \cdots \times D_n$ and producing data in $D'_1, D'_2, \cdots, D'_p$ we associate p *continuous* functions from $D_1^\omega \times D_2^\omega \times \cdots \times D_n^\omega$ into (respectively) $D_1'^\omega, D_2'^\omega, \cdots, D_p'^\omega$. For example, in Figure 5(b), we specify two continuous functions f_1 and f_2 in order to interpret node f:

$$f_1 : D_1^\omega \times D_2^\omega \times D_3^\omega \to D_1'^\omega$$
$$f_2 : D_1^\omega \times D_2^\omega \times D_3^\omega \to D_2'^\omega$$

EXAMPLE 1. *The process f of program S is associated to the continuous function f in $\mathbb{N}^\omega \times \mathbb{N}^\omega \to \mathbb{N}^\omega$ defined recursively by:*

$$f(U,V) = A(F(U), A(F(V), f(R(U), R(V)))).$$

[*]In the original article the axiom 2) is false, the correct axiom is $A(\Lambda, X) = X$.

The process g is associated to two functions, one per output line, defined recursively by:

$$g_1(U) = A(F(U), g_1(R(R(U)))) \quad \textit{and}$$
$$g_2(U) = A(F(R(U)), g_2(R(R(U))))$$

Similarly the function h maps $\mathbb{N}^\omega$ into $\mathbb{N}^\omega$:

$$h(U,x) = A(\{x\}, U)$$

where x is in $\mathbb{N}$ and the notation $\{x\}$ means the unit length sequence whose first element is x.

In theses examples, not much computation is actually performed on this inputs, but an arbitrary amount of computation could be performed by a computing station, requiring possibly an unbounded amount of memory. The restriction of the interpretation of nodes to continuous functions can be understood in concrete terms:

a) *Monotonicity* means that receiving more input at a computing station can only provoke it to send more output. Indeed this a crucial property since it allows parallel operation: *a machine need not have all of its input to start computing, since future input concerns only future output.*

b) Furthermore *continuity* prevents any station from deciding to send some output only after it has received an infinite amount of input.

Any process written in the simple programming language of Section 1 correspond to a set of continuous functions. The recursive definition of these functions is obtained by the usual method of McCarthy for converting flow-chart programs to recursive definitions.

3. FIXPOINT EQUATIONS

Rather than studying the behaviour of a complex machine, we want to study the properties of the solution of a set of equations. To each parallel *program* (*i.e.* interpreted schema) we associate a set Σ_p of equations on sequence domains in such a way that a set of sequences is a possible solution of the system *iff* it is a possible set of histories for the communication's lines of the program:

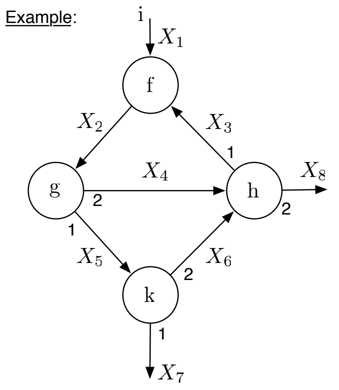
i) To every line e, of type D_e, associate to a variable X_e ranging over D_e.

ii) If $X_1, X_2, \cdots, X_n$ are the variables associated to the input lines and $i_1, \cdots, i_k$ are the sequences fed as inputs on the lines include the equations:

$$\begin{cases} X_1 = i_1 \\ \vdots \\ X_k = i_k \end{cases}$$

iii) For each node f, interpreted with the functions $f_1, \cdots, f_p$, with input variables $X_1, \cdots, X_n$ output $X'_1, \cdots, X'_p$ include p equations in Σ_P:

$$\begin{cases} X'_1 = f_1(X_1, \cdots, X_n) \\ \vdots \\ X'_p = f_p(X_1, \cdots, X_n) \end{cases}$$



$$\Sigma_P = \begin{cases} X_1 = i \\ X_2 = f(X_1, X_3) \\ X_5 = g_1(X_2) \\ X_4 = g_2(X_2) \\ X_7 = k_1(X_5) \\ X_6 = k_2(X_5) \\ X_8 = h_2(X_4, X_6) \\ X_3 = h_1(X_4, X_6) \end{cases}$$

Figure 6: *The program P and the associated system Σ_P.*

Clearly, the histories of the lines of the program P have to satisfy the system Σ_P. Σ_P is a set of fixpoint equations over c.p.o.'s, where the operators are continuous. It is a well-known mathematical result (see for example Milner [5]) that such a system admits a *unique minimal solution.* It is outside the scope of this paper to show that this minimal solution constitutes indeed the vector of histories of the communication's lines, given a suitable implementation. Such a proof can be found in Cadiou [6] in a similar set up.

The first property of this minimal solution gives us access to the most powerful rule of induction used in proving programs correct (see Manna, Ness, Vuillemin [7]), *i.e.* Scott's rule:

PROPERTY 1 (KLEENE). *The minimal solution $\{Y(X_1), Y(X_2), \cdots, Y(X_n)\}$ of the system $\Sigma_P = \{X_i = \tau_i(X_1, \cdots, X_n) \mid i \in [1, n]\}$ where the τ_i are terms built out of continuous operators is $\lim_{i \to +\infty} (X_1^i, \cdots, X_n^i)$ where*

$$X_i^0 = \Lambda (i \in [1, n])$$

(Strictly speaking there might be n different Λ's)

$$X_i^{j+1} = \tau_i(X_1^j, \cdots, X_n^j) \quad (i \in [1, n])$$

Scott's induction rule in this case can be stated as follows, if $\mathcal{P}$ is an admissible predicate (see Manna, Ness, Vuillemin [7]):

$$\frac{\mathcal{P}(\Lambda, \cdots, \Lambda) \qquad \mathcal{P}(X_1, \cdots, X_n) \supseteq \mathcal{P}(\tau_1(X_1, \cdots, X_n), \cdots, \tau_n(X_1, \cdots, X_n))}{\mathcal{P}(Y_1(X_1), \cdots, Y_n(X_n))}$$

A property of a parallel program is stated as a relation between the input sequences and the output sequences or in general between the histories of some communication lines. Since we may use Scott's rule, all the techniques for proving properties of recursive programs studied in Vuillemin [8] are available, in particular structural induction and recursion induction.

EXAMPLE 2. *The system Σ_S associated with program S is:*

$$\Sigma_S = \begin{cases} X = f(Y, Z) \\ Y = h(T_1, 0) \\ Z = h(T_2, 1) \\ T_1 = g_1(X) \\ T_2 = g_2(X) \end{cases}$$

where f, g_1, g_2 and h are given in 2.2.5.

As an illustration, let us prove that the history X which is exactly what S prints out, is an infinite alternating sequence of 0's and 1's. In other words, if $\mathcal{X}$ is the minimal fixpoint of $\mathcal{X} = A(\{0\}, A(\{1\}, \mathcal{X}))$ then $X = \mathcal{X}$.

The system Σ_S can be reduced to a single fixpoint equation:

$$X = f(h(g_1(X), 0), h(g_2(X, 1))) \tag{1}$$

Using the definition of f and h, and the properties of F and A we transform Equation (1) to

$$X = A(\{0\}, A(\{1\}, f(g_1(X), g_2(X)))) \tag{2}$$

LEMMA 1. *For all U, $U = f(g_1(U), g_2(U))$*

PROOF. By structural induction. The Lemma is obviously true for Λ, and for any sequence of length 1. Assume it is true for V, then:

$$f(g_1(A(\{a\}, A(\{b\}, V))), g_2(A(\{a\}, A(\{b\}, V))))$$

$$= A(\{a\}, A(\{b\}, f(g_1(V), g_2(V)))$$

$$= A(\{a\}, A(\{b\}, V)) \text{ by induction hypothesis.} \quad \square$$

From Equation (2) and this Lemma above we deduce:

$$\mathcal{X} \subseteq X$$

With this Lemma again it is trivial to see that $X \subseteq \mathcal{X}$, which is proves the result. Since the mapping length is continuous, length($\mathcal{X}$) is the minimal solution (in $\overline{\mathbb{N}}$) of

$$\text{length}(\mathcal{X}) = 2 + \text{length}(\mathcal{X})$$

which is obviously ∞. Hence T_1 and T_2 are infinite sequences and so are Y and Z. We have thus answered the first two questions raised in Section 1 about program S. $\square$

The simplicity of the program S and the proof produced should not induce the reader into believing that only very simple minded proofs are feasible. Milner and Weyrauch [9] used the system LCF, based on Scott's induction rule, to check mechanically the complete proof of the correctness of a small compiler, a very large proof indeed. LCF can be readily used for our purposes and very large and trustworthy proofs could be produced on this system.

PROPERTY 2 (SCOTT). *The minimal solution of the Σ_P is a continuous function of the parameters of the system, in particular the values of the input streams, or the operators of the system.*

In more concrete terms, Property 2 means that, in this model of parallel computation:

1. Arbitrary interconnection of systems, as well as processes, is legitimate. Hence, top-down design finds here a mathematical justification since we can postpone the decision to implement a given *function* by a single process or a set of interconnected processes: this decision will not introduce perturbations in the remainder of the system.

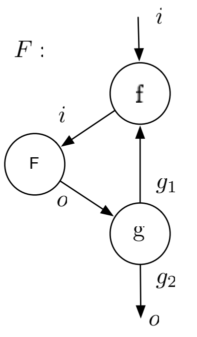


Figure 7: ***A recursive parallel schema.***

2. A parallel program can be safely simulated on a sequential machine, provided the scheduling algorithm is fair enough, *i.e.* it eventually attributes some more computing time to a process which wants it. If this algorithm is not fair however, the only thing that may happen is for the parallel program to produce *less output* than what could be expected. But what is produced is correct.

This remark and a simple argument on lengths answer the last question about program S raised in the first section.

4. RECURSION

The parallel program introduced so far actually exhibit a *bounded* parallelism: only a finite number of processes may compute simultaneously. It is necessary and easy to introduce the recursive *parallel* programs, where an unbounded number of machines may compute in parallel.

A recursive parallel schema is a set $F_1, F_2, \cdots, F_\ell$ of parallel schematas in which some nodes may be labeled $F_1, F_2, \cdots, F_\ell$. If a parallel schema F_j has input lines labeled $i_1, i_2, \cdots, i_p$ and output lines $o_1, o_2, \cdots, o_g$, then in each occurrence of F_j the same labels must occur on its input and output lines. An example is given on Figure 7.

(*N.B.*: this is a way to ensure that the parallel recursive programs are *syntactically* well formed; it is sufficient for our purposes although it may give several labels to an edge). We construct now a set of fixpoint equations that contain variables in two *types*: sequence domains, and continuous mappings between sequence domains.

Example 3. *To the schema on Figure 7 we associate the system* Σ

$$o = F(i) = g_2(F(f(i, X)))$$
$$X = g_1(F(f(i, X)))$$

where X and F are respectively an unknown sequence and an unknown continuous mapping between sequence domains. The continuous mappings from a c.p.o. into a c.p.o. with the ordering:

$$f \subseteq g \text{ iff } \forall x f(x) \subseteq g(x).$$

The existence of a minimal (now functional) solution is still assured and Property 1 and Property 2 hold along with their concrete interpretation. A little bit more care has to be exerted to make sure that the implementation computes the minimal fixpoint. The only problem is to know when to start unfolding a recursive call to a process. The good strategy is *not* to start when *input* is presented *but* when *output* is requested. This rule is basically the delay rule of Vuillemin [8].

5. SCHEMATOLOGY

Structural properties of parallel programs are discovered in studying parallel program schemata. For example we can prove that the schemata on Figure 8 are equivalent, *i.e.* whatever process f and g may be the two resulting programs will be equivalent.

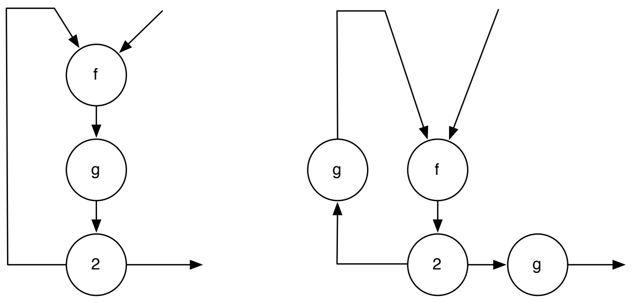


Figure 8: ***Two equivalent schemata.***

(Nota: these schemata are partially interpreted: the node 2 called a 2-plicator sends a copy of each input on each output line. We allow such nodes in schemata because they introduce no new fixpoint equations).
We state the main results (Courcelle, Kahn, Vuillemin [10]):

Theorem 1 : The equivalence of schemata containing uninterpreted processes and n-duplicators is decidable.

Theorem 2 : There exists a unique minimal schema $\hat{S}$ (*i.e.* containing a minimum number of process nodes) equivalent to a given schema S.

Theorem 3 : The system of equations corresponding to S containing the minimum number of equations are obtained by taking minimal cuts of $\hat{S}$.

The results concerning recursive parallel schemata are much harder. Restricting ourselves to recursive processes with *one input and one output* we know (Courcelle-Vuillemin [11]):

Theorem 4 : Equivalence of recursive parallel schemata is decidable.

6. DISCUSSION AND CONCLUSION

The kind of parallel programming we have studied in this paper is severely limited: it can produce only *determinate* programs. We argue however that:

i) large parts of operating systems are written so as to be determinate. The method of monitors advocated by Hoare narrows down the possible locations of nondeterminacy.

ii) the primitives *wait* and *send x on y* that we studied are not too far from reality as exemplified by [1, 3, 4].

iii) we do not think it is impossible to extend the theory to non-determinate parallel programs, although how to satisfactorily do so is far from obvious.

iv) the programming language we have introduced can be extended by adding new *primitive* processes (*i.e.* that cannot be programmed as processes with *wait* and *send*). A typical such process is
WARN (integer in X,Y; logical out Z)
that sends a *true* value on its output line each time some integer is received on *either* of its input lines. The only condition to be verified by the new primitive processes, and verified by WARN, is that the history of the output line be a continuous function of the histories of the input lines.

Looking now at the merits of our approach, we see the essential one as the eradication of the notion of *state of a complex system*. More precisely, in Lauer [12] and Gilbert [13] for example, a system is thought of as having a huge "state vector" and making non-deterministic transitions from state to state. This view leads to proofs growing exponentially with the number of processes (we grow linearly) and is blind to the structure of the system, making the proofs counter-intuitive. Furthermore it cannot deal with an unbounded number of processes, something we get almost "for free". Our proofs can be checked mechanically in LCF [14], another non negligible advantage since they will often be tedious but without great mathematical depth.

Our last conclusion is to recall a principle that has been so often fruitful in Computer Science and that is central in Scott's theory of commutation: a *good* concept is one that is closed

1. under arbitrary composition
2. under recursion.

Acknowledgments

Many thanks to J. Vuillemin and R. Milner who helped me to formulate this theory. J.M. Cadiou and J. Vuillemin provided most useful comments on many draft versions of this paper.

7. REFERENCES

[1] R. M. Balzer. An overview of the ispl computer systems design. *Communication of the ACM*, 16(2):117–122, 1973.
[2] Thomas J. Dingwall. Communication within structured operating systems. Technical Report TR 73-167, Cornell University, 1973.
[3] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communication of the ACM*, 13(4):238–241, 1970.
[4] IBM system 360 Operating system. Control program services, Form C28-6541-O.
[5] Robin Milner. Model of LCF. Technical Report AIM-186/CS-332, Computer science department, Stanford University, 1973.
[6] Jean-Marie Cadiou. *Recursive definitions of partial functions and their computations.* PhD thesis, Stanford University, 1972.
[7] Zohar Manna, Stephen Ness, and Jean Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, 16(8):491–502, 1973.
[8] Jean Vuillemin. *Proof techniques for recursive programs.* PhD thesis, Stanford University, 1973.
[9] Robin Milner and R. Weyrauch. Proving compiler correctness in a mechanized logic. In Edinburgh University Press, editor, *Machine Intelligence 7*, Edinburgh, 1972.
[10] Bruno Courcelle, G. Kahn, and J. Vuillemin. Algorithmes d'équivalence et de réduction à des expressions minimales dans une classe d'équations récursives simples. In *Second Conference on Automata, Languages and Programming*, Saarbrüken, 1974.
[11] B. Courcelle and J. Vuillemin. Semantics and axiomatics of a simple recursive language. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 13–26. ACM, 1974.
[12] Hugh C. Lauer. *Proof techniques for recursive programs.* PhD thesis, Computer science department, Carnegie Mellon University, 1972.
[13] Philip Gilbert and W. J. Chandler. Interference between communicating parallel processes. *Communications of the ACM*, 15(6):427–437, 1972.
[14] Robin Milner. Implementation and applications of scott's logic for computable functions. In *Proceedings of ACM conference on Proving assertions about programs*, 1972.