A Preliminary Theory for Parallel Programs

Gilles Kahn

A PRELIMINARY THEORY FOR PARALLEL PROGRAMS

Gilles KAHN*

Résumé :

Dans sa thèse, D. ADAMS a introduit un modèle de calcul parallèle qui permet de représenter une large classe d'algorithmes parallèles. Depuis, D. SEROR a signalé l'intérêt du modèle de ADAMS pour représenter les tâches d'un système d'exploitation.

Nous montrons dans ce rapport comment certaines idées de D. SCOTT permettent un traitement systèmatique des propriétés de ce modèle de calcul parallèle.

Abstract :

In his thesis, D. ADAMS introduced a model of parallel computation which is convenient to represent a wide class of parallel algorithms. Later, D. SEROR pointed out the interest of ADAMS'model to represent the tasks of an operating system.

We show here how ideas due to D. SCOTT allow a systematic treatment of the properties of this model of parallel computation.

---

* Work reported herein has been carried out under a joint project supported by Commissariat à l'Energie Atomique and IRIA (LABORIA)

Introduction :

In his thesis [1], D. ADAMS introduced a model of parallel computation which is convenient to represent a wide class of parallel algorithms. He exhibited several moderately convincing examples of parallel programs for sorting or matrix computations. Later, D. SEROR [11] pointed out that ADAMS'model was actually better suited to represent the tasks of an operating system. We shall study here the formal properties of this kind of parallel programs.

The paper is intended to be self-contained but previous exposure to SCOTT [9, 10] would clearly be beneficial to the reader.

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 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 stations may communicate.

ii) A communication line transmits information within an impredictable 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 a program elsewhere).

This intuitive description of a parallel computation calls for several 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 one or another of its input lines, but of course it can be waiting on the conjunction of several input lines by sequentially waiting on each one of them. Third, we do not restrict the computing stations to have a finite memory. A good representation of this system

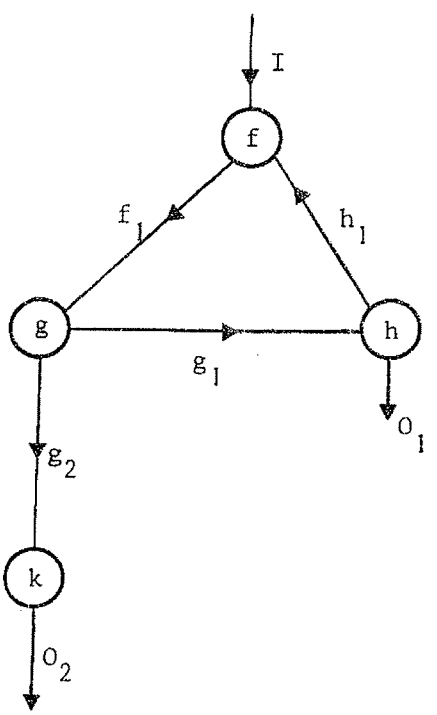
is a set of Turing machines connected via one-way tapes, where each machine can use its own working tape. For the reader acquainted with OS/360 [13] or similar operating systems, one can think of a set of assembly language programs : the computation lines are the variables which can be WAITed on or POSTed. Restriction iii) means that only one program is allowed to POST a given variable.

Parallel program schemata

We formalize now the notion of parallel computation introduced above.

1. Syntax

A parallel program schema is an oriented graph with labeled nodes and edges, together with some supplementary edges (see fig 1) : incoming edges with only end vertices, meant to represent the input lines, and outcoming edges, with only origin vertices, the output lines.



I : input line

f_1, g_1, g_2, h_1 : internal communication lines

O_1, O_2 : output lines

Figure 1.

2. Semantics

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, 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 lines into the histories of its output lines.

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 $\sqsubseteq$ defined by

$$X \sqsubseteq Y \text{ iff } X \text{ is an initial segment of } Y$$

is a partial order on D^ω. We make D^ω into a lattice (fig 2) by adding an artificial element $\top$ such that

$$\forall\, x \in D^\omega : x \sqsubseteq \top$$

The greatest lower bound (g.l.b) of two sequences is their longest common initial segment. If two sequences x and y are incomparable, their least upper bound (l.u.b) $x \cup y$ is $\top$, otherwise it is the longest of x and y.

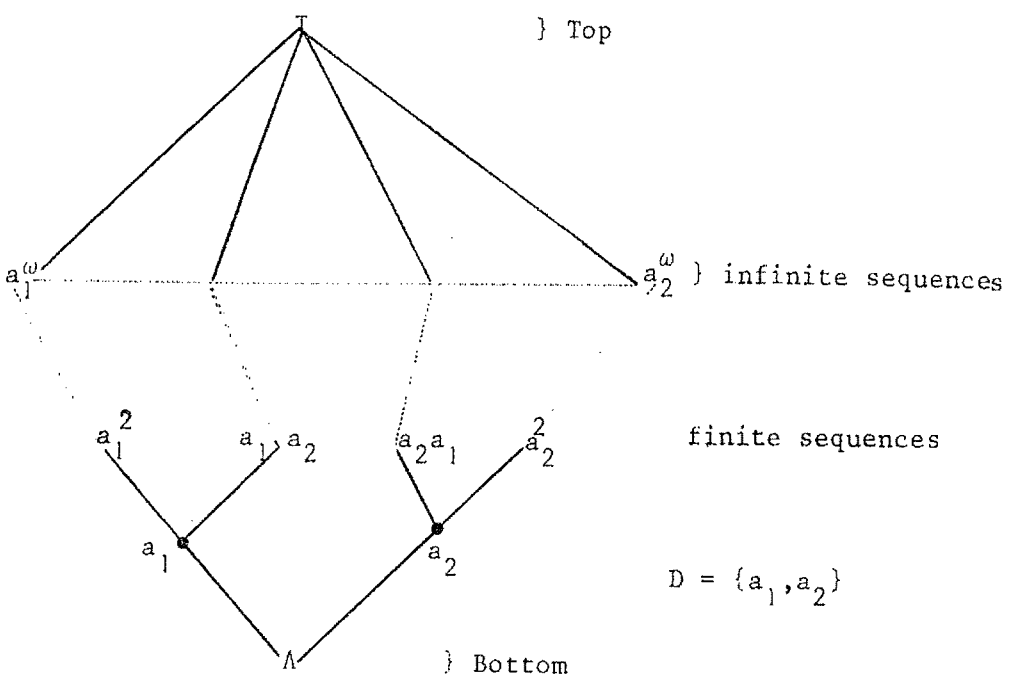


Fig 2: D^ω

We recall the definitions :

- A directed subset A in a lattice is a set such that

$$x \in A,\ y \in A \Rightarrow x \cup y \in A.$$

- A lattice T is complete if every subset of T has a l.u.b.

The reader can check that D^{ω} is a complete lattice. If D is countable, D^{ω} is a data type in the sense of Scott [9].

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}.

2.4 Continuous mappings

A mapping f from a complete lattice A into a complete lattice B is continuous iff, for any directed subset a of A.

$$f(\cup a) = \cup f(a)$$

where the $\cup$ on the left-hand side of the equation concerns the ordering in A and on the right-hand side the ordering in B. Note that a continuous mapping is also monotonic, i.e.

$$x \subseteq y \Rightarrow f(x) \subseteq f(y)$$

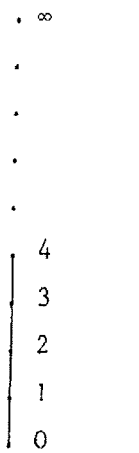
We give several examples of continuous mappings of particular importance in the sequel :

- HD : to any sequence x in D^{ω}, HD associates the unit length sequence constituted of the leftmost element of x.
- TL : to any sequence x in D^{ω}, TL associates the sequence to the right of its leftmost element.
- CONS : takes two arguments L1 and L2 in D^{ω} to produce the sequence : leftmost element of L1 followed by L2.

Note that these definitions imply :

$$TL(\Lambda) = HD(\Lambda) = \Lambda \ ; \ CONS(A,\Lambda) = HD(A) \ ; \ CONS(\Lambda,A) = \Lambda$$

- we shall need also to talk formally about the length of a sequence. Io do so , we can take the integers with their usual order, complete them with a top element ∞ to obtain a complete lattice $\overline{N}$ (fig 3). Addition in this lattice is a continuous function . The mapping from D^{ω} to $\overline{N}$ which maps a sequence into its length is continuous.



The lattice $\overline{N}$

Fig 3.

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, D_2,....D_n and producing data in D'_1, D'_2, ...D'_p we associate p continuous functions from $D_1^\omega \times D_2^\omega \times \ldots D_n^\omega$ into (respectively) $D_1'^\omega$, $D_2'^\omega$, ... $D_p'^\omega$.

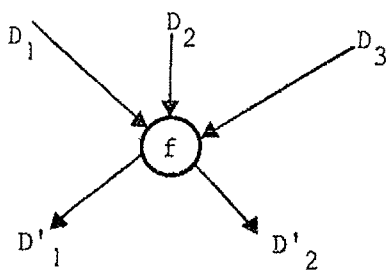


Fig 4.

For example, in Fig 4, we specify two continuous functions f_1 and f_2 in order to interpret node f :

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

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

Examples :

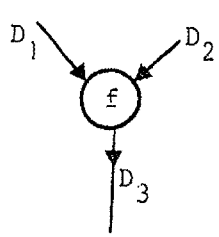


Fig 5.a

a) Fig 5.a :

$D_1 = D_2 = D_3 = N$

$D_1^\omega = D_2^\omega = D_3^\omega = N^\omega$

f maps L_1,L_2 in N^ω into the sequence L_3 in N^ω whose i-th element is the sum of the i-ths elements of L_1 and L_2.

fig 5.b

b) fig 5.b

$L \in D^{\omega}$, $x \in D$, $\{x\} \in D^{\omega}$ (sequence of length 1)

$f(L) = CONS(\{x\},L)$ (Note $f(\Lambda) \neq \Lambda$).

fig 5.c

c) fig 5.c

$L \in N^{\omega}$, $L' = f(L) \in N^{\omega}$, the ith element of L' is the sum of the first i elements of L and $f(\Lambda) = \Lambda$

(Note : f needs unbounded memory).

Other examples can be found in Kahn [3].

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 the possibility for a station to take the decision to send some output only after it has received an infinite amount of input.

Remarks :

- Computable functions over the real numbers (See for example Minsky[14]) are continuous.

- The reader should convince himself that a sequential program written with WAIT and POSI (or READ and WRITE) is a continuous function from its input stream(s) into its output stream(s).*

The fundamental result

A parallel program is an interpreted parallel schema. The key results about parallel programs are contained in the

* The converse is not true. Call serial the continuous functions which can be written as sequential programs with WAIT and POST. The following continuous function φ is not serial :

$$\varphi \in D^{\omega} \times D^{\omega} \to D'^{\omega},\ D' = \{a\},\ \varphi(x,y) = \underbrace{aa.....a}_{length(x)+length(y)}$$

Theorem :

Given a parallel program and some input sequences :

i) A unique minimal history can be associated to each communication line and in particular to the output lines.

ii) The unique minimal history of each communication line is a continuous function of the input sequences.

Proof :

a) To each parallel program P, we associate a set Σ_p of fixpoint 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 lines in the program :

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

ii) If $X_1, X_2, \ldots X_n$ are the variables associated to the input lines, $i_1, \ldots i_k$ the sequences fed as inputs on the lines write

$$X_1 = i_1$$
$$\cdot \quad \cdot \quad \cdot$$
$$\cdot \quad \cdot \quad \cdot$$
$$X_k = i_k$$

iii) For each node f, interpreted with functions $f_1, \ldots f_p$, input variables $X_1, \ldots X_n$, output variables $X'_1, \ldots X'_p$, write p equations

$$\left.\begin{aligned} X'_1 &= f_1(X_1, \ldots X_n) \\ X'_p &= f_p(X_1, \ldots X_p) \end{aligned}\right\}$$

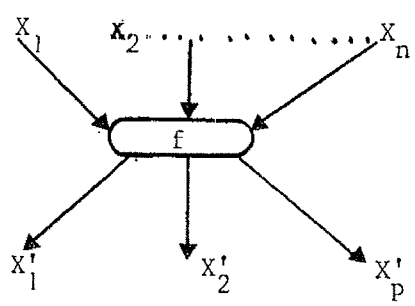
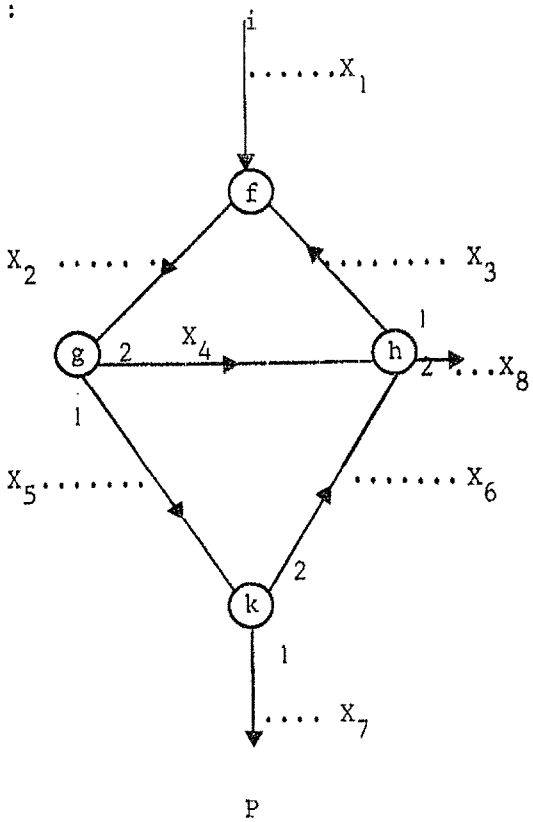


Fig 6

Example :



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

Fig 7

Clearly, the histories of the lines of the program P have to satisfy the system Σ_p.

b) We have taken care in the previous section that :

i) the variables X_i should range over complete lattices.

ii) the functions involved in should be continuous mappings.

It is well known (see for example Kleene's first recursion Theorem [4]) that :

- There exists a unique minimal solution for Σ_p.
- The solution can be approached by successive iterations, starting with $\forall_i$, $X_i = \Lambda$ [Strictly speaking, there exists a Λ for each sequence domain]

And from Scott [10], we know a little more about the solution :

- The solution is a continuous function of the parameters of the system, in this case the input streams to the program (e.g:i on fig 7).

Remark :

In Σ_p we have a ridiculously high number of variables and equations. Actually, we need only one variable per output line when the program contains no cycles. If there are cycles, we find a minimal cut-set of the program and introduce only variables for the lines in this cut-set.On the example on Fig 7, we keep only X_2, X_6, X_7, X_8. Σ_p becomes :

$$\Sigma_p \quad \begin{cases} X_2 = f(i,\ h_1(g_2(X_2), X_6)) \\ X_6 = k_2\ (g_1(X_2)) \\ X_7 = k_1(g_1(X_2)) \\ X_8 = h_2(g_2(X_2), X_6) \end{cases}$$

From now on, we shall only consider the implementations of parallel programs that actually compute minimal histories. For instance the realization via Turing Machines is adequate .

Applications

Thanks to the fundamental result, we study the properties of a system of equations rather than the behaviour of a complex machine.

1. First examples

1.1 In the system of fig 8, let X range over D^ω and f be the identity in D^ω. We associate to the (input-less) system, the equation :

$$X = X$$

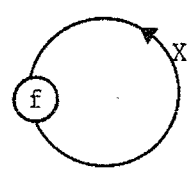


fig 8.

Any sequence in D^ω is a solution, but Λ is the minimal solution. Nothing happens !

1.2. Interpret now f as

$$f(L) = CONS(a, L) \qquad a \in D$$

In other words, the machine f sends a, then everything that comes along. We get now the equation :

$$X = CONS(a, X) \qquad (1)$$

We can approximate the minimal solution U of this equation :

$$U_o = CONS(a,\Lambda) = a$$

$$U_1 = CONS(a,X_o) = aa$$

$$U_n = CONS(a,X_{n-1}) = a^n$$

$$U = \lim_{n\to\infty} X_n = a^\omega$$

We prove now, that the length of U is infinite. The proof is pedantic, but much in the style of what can be done with in the context of an automatic proof checker such as LCF (Milner [6]).

$$length(CONS(a,X)) = 1 + length(X) \quad \text{(def. of length)}.$$

Since length is a continuous mapping from D^ω to $\overline{N}$:

$$(1') \quad length\ (U) = 1 + length\ (U)$$

But (1') is a fixpoint equation in $\overline{N}$ with minimal solution ∞.

In essence, we just gave a formal proof that the systems of fig 8, in the interpretation considered, runs forever.

1.3 In the system of fig 9 interpret

f as $f(L) = CONS(a, L)$

g as $g(L) = TL(L)$

(f sends an a, then anything that comes in, g waits for something and then sends what comes after)

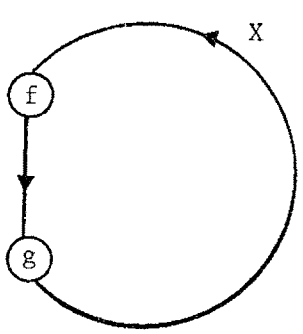


Fig 9.

The equation $X = g\ (f(X))$ reduces to $X = X$ thus X is the empty sequence : this system blocks.

On the other hand if $g(L) = CONS(b,L)$, we obtain

$$X = (b\ a)^{\omega}$$

and the system runs forever.

1.4 On the example of fig 10, parallelism is more actively involved. Functions g and h are interpreted as :

$$g(L) = CONS(a,L) \qquad h(L) = CONS(b,L)$$

and f, f'_1, f'_2 are defined recursively :

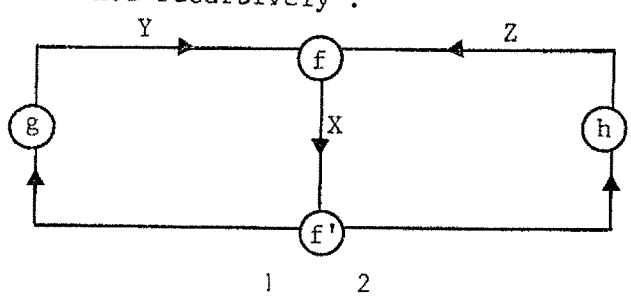


Fig 10.

$$f(L,M) = CONS\ (HD(L),\ CONS(HD(M),\ f(TL(L),\ TL(M))))$$

$$f'_1(L) = CONS\ (HD(L),\ f'_1\ (TL(TL(L))))$$

$$f'_2(L) = CONS\ (HD(TL(L)),\ f'_2\ (TL(TL(L))))$$

By induction it is easy to prove that :

$$X = (a\ b)^{\omega} \quad Y = a^{\omega} \qquad Z = b^{\omega}$$

Note :

We shall often use in the sequel the n-plicator (fig 11) : this machine sends a copy of each input on all of its n output lines.

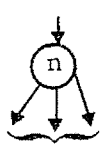


Fig 11.

When an n-plicator occurs in P, it introduces only trivial equations in Σ_p.

2. Proving properties of parallel programs

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. Numerous techniques for dealing with systems of fixpoint equations have been already developped (see for example Manna, Ness, Vuillemin [5] for a review or Vuillemin [12]) and they are readily applicable. Furthermore an entire logical system doted of a powerful induction rule has been built and even implemented (Milner [6]) for that very same purpose.

3. Closure

The main result can be thought of as a closure property : given a certain number of computing stations, we can connect them into a network that is equivalent to a single computing station (See Patil [8]). It also shows that in a parallel program, substitution is allowed : we can replace a program by an equivalent one without risking to modify the initial program. This results legitimates top-down (or hierarchical) design of parallel programs. It also yields easy proofs of some properties of sets of fixpoint equations :

Property 1

The schemata of fig12 are equivalent

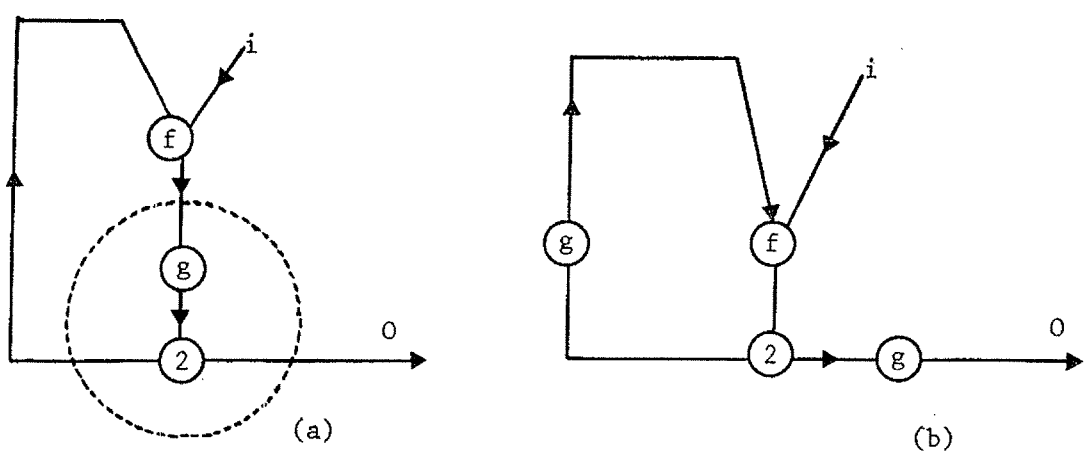
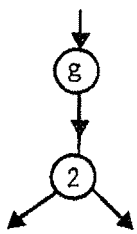


fig 12.

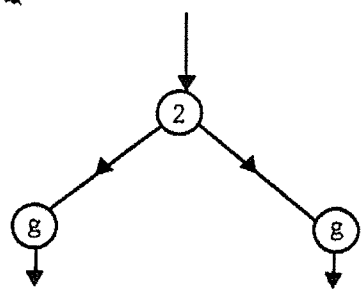
Proof :

The schemata of fig 13 are obviously equivalent and fig 12b is obtained from fig 12a by substitution of 13 a by 13 b. ∎



a

Fig 13.



b

If we denote $\mu x.t$ the minimal solution of $x = t(x)$, we recognize in Property 1 the following lemma of Vuillemin [12]:

$$\mu x.g(f(x,i)) = g(\mu x.f(g(x,i))).$$

Property 2 $\quad \mu x.f(x,x) = \mu z.(\mu y.f(y,z))$.

Proof : From the decomposition indicated on Fig 14 we can write either (I) $X = f(X,X)$ or if $\varphi(z)$ denotes the fixpoint of

$$\left.\begin{aligned} Y &= f(Y,z) \\ Z &= \varphi(Z) \\ X &= Z \end{aligned}\right\} \quad (II)$$

∎

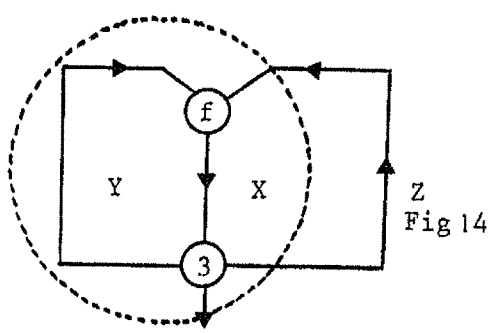


Fig 14

Remark :

This is essentially the method used by Scott, de Bakker [2] or Park [7] to eliminate simultaneous recursions.

Property 3 The schemata on fig 15 are equivalent.

Proof : From Property 1 :

$$\mu x.f^2(x) = f(\mu x.f^2(x))$$

Hence $\quad \mu x.f(x) \leq \mu x.f^2(x) \qquad (1)$

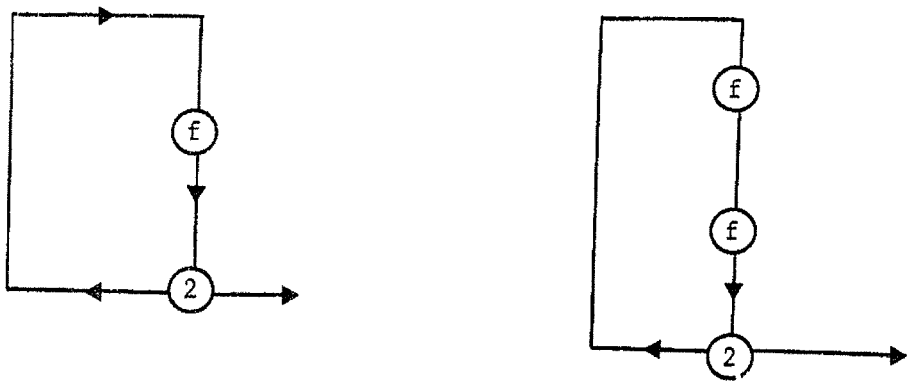
But $\quad \mu x.f(x) = f(\mu x.f(x)) = f^2(\mu x.f(x))$

Hence $\quad \mu x.f^2(x) \leq \mu x.f(x) \qquad (2)$

From (1) and (2) we get

$$\mu x.f(x) = \mu x.f^2(x) \qquad (3)$$

(3) is the algebraic statement of the equivalence of the schemata on fig 15 a and 15 b :



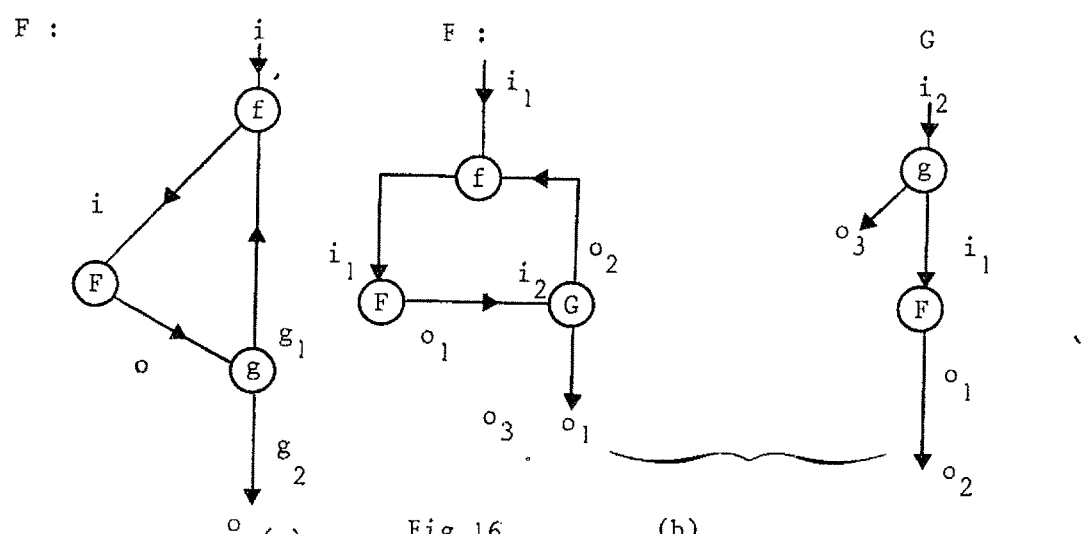
(a) (b)

Fig 15.

Recursion

The parallel programs introduced so far actually exhibit a limited parallelism : only a finite number of machines may work simultaneously. Following our approach, we introduce easily the recursive parallel programs, where an unbounded number of machines may compute in parallel.

A recursive parallel schema is a set F_1, F_1, ...F_2 of parallel schemata in which some nodes may be labeled F_1, F_2, ...F_1. If a parallel schemata F_j has input lines labeled i_1, i_2, ...i_p and output lines o_1, o_2, ...o_g, then in each occurrence of F_j the same labels must occur on its input and output lines. Examples are given on fig 16 :



(a) Fig 16. (b)

(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 one edge).

The fixpoint equations now contain variables in two types : sequence domains, and continuous mappings between sequence domains.

Example

To the schema on fig 16(a) we associate the system of equations Σ :

$$\Sigma \quad \begin{cases} o = F(i) = g_2(F(f(i,X))) \\ X = g_1(F(f(i,X))) \end{cases}$$

where X and F are respectively an unknown sequence and an unknown continuous mapping between sequence domains. The continuous mappings from a complete lattice into a complete lattice constitute also a complete lattice (cf Scott [10]) with the ordering

$$f \sqsubseteq g \text{ iff } \forall x \; f(x) \sqsubseteq g(x)$$

The fundamental result holds without modification.

Here again we shall consider only the implementations of recursive parallel programs that actually compute the fixpoints (sequences and functions). The discussion of valid implementations is beyond the scope of this paper, but it can be readily noted that Adams [1] and Seror [11] fail to give correct computation rules.

Example

Interpret the schemata on fig 17 as follows :

$$f_1(i) = TL(i) \quad f_2(i) = HD(i) \quad g(i_1,i_2) = CONS(i_2, i_1)$$

and h defined recursively as

$$h(i) = CONS(h_1(HD(i)), h(TL(i)))$$

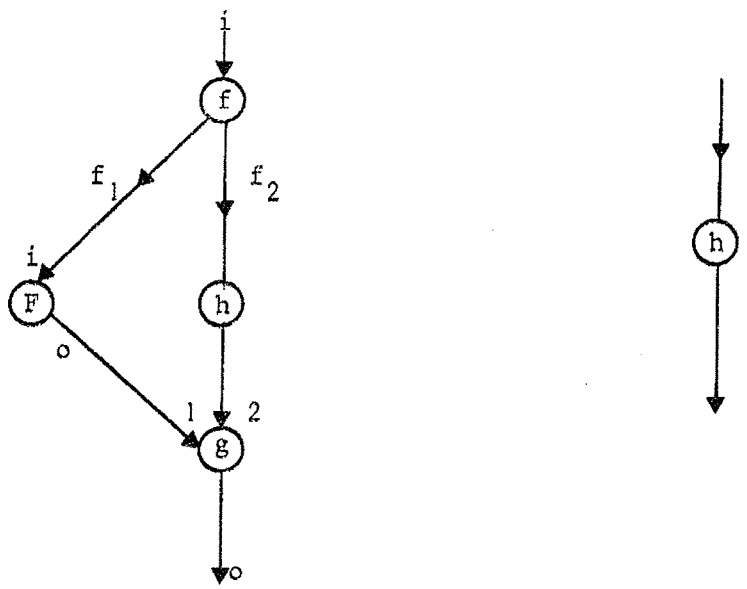
with $i \in D^{\omega}$, $h_1 \in D^{D}$.

From $F(i) = g(F(f_1(i)), hf_2(i))$ we get easily

$$F(i) = CONS(h_1(hd(i)), F(TL(i)))$$

Hence $F = h$ ∎

In this interpretation, schema 17(a) and 17(b) compute the same function. But in 17(a) many instances of h are computing in parallel !



(a) (b)

Fig 17.

Conclusion

The essential result of this paper is that we can consider a certain variety of parallel programs as regular programs, but operating on a different data type : sequence domains. The usual methods for proving properties of programs will work. A proof-checking program in the style of Milner's LCF [6] can be used to check these proofs.

In this paper, we have only developped a theory for determinate parallel programs. Significant portions of operating systems can be modeled by such programs. We hope that Scott's insightful theory can be applied to non determinate parallel programs, the next step towards a satisfactory theory for operating systems.

Acknoledgments :

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 the many draft versions of this paper.

REFERENCES

[1] Adams, D. "A computation model with data flow sequencing". Ph. D. dissertation. Stanford University, Computer Science Department. December 1968.

[2] de Bakker, J.W, Scott, D. "A theory of programs" Unpublished Notes. August 1969.

[3] Kahn, G. "An approach to systems correctness" Proceedings of the ACM 3rd Symposium on Operating Systems Principles. October 1971.

[4] Kleene, S.C. "Introduction to metamathematics". North-Holland. 1952.

[5] Manna, Z., Ness, S., Vuillemin, J. "Introduction methods for proving properties of programs" ACM conference on Proving Assertions about Programs. January 1972.

[6] Milner, R. "Implementation and Applications of Scott's Logic for computable functions" ACM Conference on Proving Assertions about Programs. January 1972.

[7] Park, D., Hitchcock, P., "Induction rules and termination proofs" in Automata, Languages and Programming. North-Holland 1972.

[8] Patil, S., "Closure Properties of interconnection of determinate systems" Project MAC Conference on concurrent systems and parallel Computation. June 1970.

[9] Scott D., "Outline of a mathematical theory of computation" Programming Research Group Technical Monographs PRG-2. Oxford University.

[10] Scott D., "Continuous lattices" Programming Research Group Technical Monographs PRG-7. Oxford University.

[11] Seror D., "D.C.P.L A distributed control programming language". Ph.D. dissertation. Computer Science Department. University of Utah. August 1970.

[12] Vuillemin, J., "Proof techniques for recursive programs". Ph.D. dissertation. Stanford University, Computer Science Department (To appear)

[13] IBM Operating system/360. Control program services. FORM C28-6541-0.

[14] MINSKY, M. "Computation : finite and infinite machines". Prentice Hall. 1967.