Printing Requisition/Graphic Services

Dept. No. 77057

Title or Description: CS-79-15 "Predicate Logic as a Language for Parallel Programming"

Date: Feb. 20, 1980 | Date Required: ASAP please | Account: 126-6353-41

Signature: M.H. van Emden | Signing Authority: [illegible signature]

Department: Computer Science | Room: MC 5100B | Phone: 3402

Delivery: ☐ Mail ☐ Pick-up ☐ Via Stores ☐ Other

1. Please complete unshaded areas on form as applicable. (4 part no carbon required).
2. Distribute copies as follows: White, Canary and Pink—Printing, Arts Library or applicable Copy Centre Goldenrod—**Retain**.
3. On completion of order, pink copy will be returned with printed material. Canary copy will be costed and returned to requisitioner. **Retain as a record of your charges.**
4. Please direct enquiries, quoting requisition number, to Printing/Graphic Services, Extension 3451.

Reproduction Requirements: ☒ Offset ☐ Signs/Repro's ☐ Xerox

Number of Pages: 32 | Number of Copies: 15

Type of Paper Stock: ☒ Bond ☐ Book ☐ Cover ☐ Bristol ☐ Supplied

Paper Size: ☒ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17

Paper Colour: ☒ White ☐ Other

Ink: ☐ Black

Printing: ☐ 1 Side ☒ 2 Sides

Numbering: to

Binding/Finishing Operations: ☒ Collating ☐ Corner Stitching ☐ 3 Ring ☐ Tape ☐ Plastic Ring ☐ Perforating

Folding — Finished Size

Cutting — Finished Size

Special Instructions

Covers supplied and 2 staples.

Cost: Time/Materials		Fun.	Prod.Un. Qt.	Prod.Un. No.	Mins.	Total
Signs/Repro's	1					
Camera	2					
Correcting & Masking Negatives	3					
Platemaking	4					
Printing	5					
Bindery	6					
Sub. Total Time						
Sub. Total Materials						
Prov. Tax						
Total						

Film — Qty | Size

Plates — Qty | Size & Type

Paper — Qty | Size

Plastic Rings — Qty | Size

Outside Services

Graphic Services May 68 482-2

Printing Requisition/Graphic Services

Dept. No. 53001

1. Please complete unshaded areas on form as applicable. (4 part no carbon required).
2. Distribute copies as follows: White, Canary and Pink—Printing, Arts Library or applicable Copy Centre. Goldenrod—Retain.
3. On completion of order, pink copy will be returned with printed material. Canary copy will be costed and returned to requisitioner. Retain as a record of your charges.
4. Please direct enquiries, quoting requisition number, to Printing/Graphic Services, Extension 3451.

Title or Description: CS-79-15 Predicate Logic as a Language for Parallel Programming

Date: Feb 13/84

Date Required:

Account: 126-4431-02

Signature:

Signing Authority:

Department: Computer Science

Room: 6057

Phone:

Delivery: ☐ Mail ☐ Pick-up ☐ Via Stores ☐ Other

Reproduction Requirements: ☐ Offset ☐ Signs/Repro's ☒ Xerox

Number of Pages: 37

Number of Copies: 1

Type of Paper Stock: ☒ Bond ☐ Book ☐ Cover ☐ Bristol ☐ Supplied

Paper Size: ☒ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17

Paper Colour: ☒ White ☐ Other

Ink: ☐ Black

Printing: ☒ 1 Side ☐ 2 Sides

Numbering: to

Binding/Finishing Operations: ☒ Collating ☒ Corner Stitching ☐ 3 Ring ☐ Tape ☐ Plastic Ring ☐ Perforating

Folding: Finished Size

Cutting: Finished Size

Special Instructions:

Cost: Time/Materials	Func.	Prod Un.	Prod. Op. Cl.	No.	Mins.	Total
Signs/Repro's	1					
Camera	2					
Correcting & Masking Negatives	3					
Platemaking	4					
Printing	5					
Bindery	6					
Sub. Total Time						
Sub. Total Materials						
Prov. Tax						
Total						

Film: Qty Size

Plates: Qty Size & Type

Paper: Qty Size

Plastic Rings: Qty Size

Outside Services

GRAPHIC SERVICES SEPT 80 482-2

Printing Requisition / Graphic Services

Dept. No. 46010

Title or Description

Date | Date Required | Account

Signature | Signing Authority

Department | Room | Phone | Delivery ☐ Mail ☐ Via Stores ☐ Pick-up ☐ Other

1. Please complete unshaded areas on form as applicable. (4 part no carbon required).
2. Distribute copies as follows: White, Canary and Pink—Printing, Arts Library or applicable Copy Centre Goldenrod—**Retain.**
3. On completion of order, pink copy will be returned with printed material. Canary copy will be costed and returned to requisitioner. **Retain as a record of your charges.**
4. Please direct enquiries, quoting requisition number, to Printing/Graphic Services. Extension 3451.

Reproduction Requirements ☐ Offset ☐ Signs/Repro's ☐ Xerox

Number of Pages

Number of Copies

Type of Paper Stock ☐ Bond ☐ Book ☐ Cover ☐ Bristol ☐ Supplied

Paper Size ☐ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17

Paper Colour ☐ White ☐ Other

Ink ☐ Black

Printing ☐ 1 Side ☐ 2 Sides

Numbering to

Binding/Finishing Operations ☐ Collating ☐ Corner Stitching ☐ 3 Ring ☐ Tape ☐ Plastic Ring ☐ Perforating

Folding Finished Size

Cutting Finished Size

Special Instructions

Cost: Time/Materials	Fun.	Prod.Un.	Prod. Cl.	Opr. No.	Mins.	Total
Signs/Repro's	1					
Camera	2					
Correcting & Masking Negatives	3					
Platemaking	4					
Printing	5					
Bindery	6					
Sub. Total Time						
Sub. Total Materials						
Prov. Tax						
Total						

Film Qty	Size	**Plates** Qty	Size & Type
Paper Qty	Size	**Plastic Rings** Qty	Size

Outside Services

Graphic Services May 68 482-2

PREDICATE LOGIC AS A LANGUAGE FOR PARALLEL PROGRAMMING

by

M.H. van Emden, G.J. de Lucena*
& H. de M. Silva

Department of Computer Science
Department of Systems Design*

University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Revised Version November 1980

CS-79-15

*Present address: Departamento de Sistemas e Computacao
Universidade Federal da Paraiba
Campina Grande, Paraiba
Brasil

ABSTRACT

We describe the formulation, execution, semanticization, and verification within first-order predicate logic of programs in Kahn's model of computation. The relations computed by process activations are defined in logic. The state of a network of communicating parallel processes is specified in a single statement of logic which is a concise textual representation of such a network. The state is understood to comprise the configuration of the network of process activations, the contents of the channels, as well as the state of each sequential computation within a process activation.

It is possible to derive within logic results from the process definitions and from the state specification in such a way that each stage of the derivation can again be interpreted as a state of a parallel computation and that the transitions between stages is also directly meaningful in terms of Kahn's model of computation.

We show that dataflow programs in Lucid are closely related to our representation of these programs in logic. We give an example of partial verification of a terminating program. Finally, we sketch the application of recent results on greatest fixpoints and infinitary Herbrand universes to verification of nonterminating programs.

1. INTRODUCTION

Kahn has proposed [LPP] an attractive model of computation, together with a mathematical semantics for it. In a subsequent paper [NPP] with McQueen an implementation of the model was described and illustrated by examples which show that the model is conducive to elegant and easy-to-verify solutions to interesting programming problems.

We introduce a description of Kahn's model of computation by a simple programming problem. The problem is to perform 'balanced addition' on a sequence of reals. Usually numbers are added as in

$$((((((a_1 + a_2) + a_3) + a_4) + a_5) + a_6) + a_7) + a_8$$

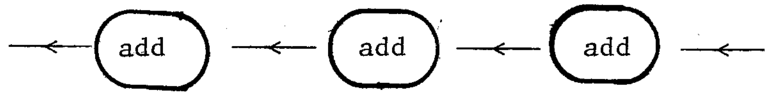
With respect to rounding errors it is preferable to add them as in

$$((a_1 + a_2) + (a_3 + a_4)) + ((a_5 + a_6) + (a_7 + a_8))$$

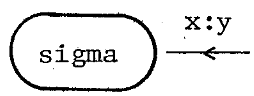
which is an example of balanced addition. The programming problem requires this to be done in a single pass over a sequence of reals which has to be sequentially accessed. The length of the sequence is not known in advance.

A Parallel *) program consists of a network of processes connected by channels which transmit data. In order to perform balanced addition on the eight numbers of our example we use a network of three processes which all perform the same computation (called 'add') of getting two successive numbers out of their input channel and putting the sum into their output channel.

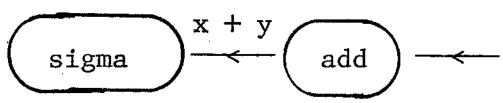
*) We use the capitalized 'Parallel' to denote that a feature is specific to Kahn's model of computation.



The above network is of course not a satisfactory solution. The number of add-processes in the network should depend on how many reals have to be added. So instead of the above *static* network, which does not change its configuration, we need a *dynamic* network, which does. We define a process called 'sigma' with an input channel only.



As soon as it has read two numbers x and y, it changes the network to



Sigma is an example of a dynamic process. The effect is to generate exactly as many activations of the add-process as are necessary to perform balanced addition on a sequence of reals of which the length is initially unknown. As soon as sigma reads eof, it prints the number previously read, if present, otherwise it prints 0.

In this paper we describe the formulation, execution, semanticization, and verification within first-order predicate logic of programs in Kahn's model of computation. We define the relations computed by process activations. We specify a state of a Parallel computation in a clear and concise way as a single statement of logic. The state is understood to comprise the configuration of the network of process activations, the contents of the channels, as well as the state of each sequential computation within a process activation. We also show how to transform a Lucid, data-flow program step by step into an equivalent logic program.

The basic discovery reported in this paper is that it is possible to derive within logic results from the process definitions and from the state specification in such a way that each stage of the derivation can again be interpreted as a state of a Parallel computation and that the transition between stages in the derivation is also directly meaningful in terms of Kahn's model of computation.

We envisage the following advantages of our approach to Parallel computation: because the programming language and specification language are the same, correctness proofs are easy to formalize and remain understandable even after formalization. We include a proof that sigma indeed produces the sum; the proof method is due to Clark and Tärnlund [FOT].

Kahn's model of computation is remarkably convenient for a wide range of applications [NPP], but it does not cover the entire domain where parallelism is useful. This paper shows that logic can be restricted to coincide with Kahn's model. But one should keep in mind that, as a programming language, it is not restricted to this model: examples are very high level formulations of sorting [PLPL] and of the eight-queens problem [CRLP], where parallelism is essential.

There are two different methods for implementation of the logic approach to parallelism, both based on the Prolog language [GDM]. Clark and McCabe have made a new version of Prolog [ICP], with coroutining control built into it. For the examples in this paper we have used the existing Prolog language, taking advantage of its flexible data structures to write a Parallel interpreter in Prolog for logic definitions. The sequential computations are passed directly on to Prolog, while the interpreter only negotiates the scheduling among the different process activations of the network.

2. A LANGUAGE FOR PARALLEL PROGRAMMING

In the paper by Kahn [LPP] Parallel programs are expressed in an Algol-like language with some additional constructs and primitives for dealing with parallel computation. We follow Kahn's method by adhering as closely as possible to an Algol-like language, in our case, Pascal.

A process is analogous to a procedure in that both execute a computation which is defined in a declaration, see lines (2) and (3) of Box 2.1. A call referring to a process declaration creates a process activation, such as in lines (5) and (6). Whenever a process is created it obtains channels as actual parameters, which are created by declarations, such as in lines (1) and (4). Typically several processes are created in unspecified order to be executed in parallel, such as ADD and SIGMA in line (5). The *par* operation of line (5) is the parallel counterpart of the sequential ";".

```
(1) program var u:  channelof real;
(2)   process ADD  (inchannel u:  real; outchannel v:  real);
      {adds successive pairs in input stream and outputs their sums}
         var x,y:  real;
      begin while not eof(u) do
            begin get(x,u)
            ;     if eof(u)
                  then begin put(x,v); put(eof,v); stop end
                  else begin get(y,u); put(x+y,v) end
            end
      ;     put(eof,v); stop
      end;
(3)   process SIGMA (inchannel u:  real);
      {writes the sum of all numbers contained in u}
(4)      var x,y:  real; ul:  channelof real;
      begin if eof(u)
            then begin write(0); stop end
            else begin get(x,u)
                 ;     if eof(u)
                       then begin write(x); stop end
                       else begin get(y,u); put(x+y,ul)
(5)                         ;     ADD(u,ul) par SIGMA(ul)
      end        end          end;  {read numbers into  u}
(6) begin SIGMA(u) end.
```

Box 2.1:   A Parallel Program.

The way a process operates on channels is specified (by *inchannel* or *outchannel*) in the code of the process declaration which refers to formal parameters which stand for channels. When processes are created this must happen in an environment where channels have been created by suitable declarations, such as in lines (5) and (6). Those created channels occur as actual parameters in the statements which create processes.

A process which has a channel as actual parameter replacing an inchannel (outchannel) formal parameter, is the *consumer* (*producer*) of that channel. Processes and channels must be created in such a way that no channel has more than one producer and also not more than one consumer.

The primitives specific to Parallel computation are 'get' and 'put' which have as first argument a value of type t and as second argument a value of type *channelof* t; and 'eof' which has one argument of type channel.

get(x,u) removes the first element of u and assigns it to x; if no element is present in u, then the call remains blocked until the time when an element becomes present.

eof(u) returns true if the first element in the channel u is the end-of-file marker eof and false if the first element of u is not eof. While u is empty execution is blocked, as with 'get'.

put(x,y) inserts element x into channel u.

Note that none of these commands allows a terminating test for emptiness of a channel. The 'get' and 'put' are adapted from Kahn's work, which only gave examples of infinite histories. In this example we do not want to specify what happens when a process reads past 'eof', hence the explicit 'stop', which halts forever the activation of a process and causes it to vanish from the network.

One should distinguish 'static' from 'dynamic' process definitions. A process with a static definition does not change the configuration of processes and channels created. It contains only sequential code. It typically executes a cyclic computation. A process with a dynamic definition causes the configuration to change. It typically contains a *par* statement creating new process activations and starting their Parallel execution; the definition also creates new channels to connect them. ADD is an example of a static process definition; SIGMA is an example of a dynamic one.

3. LOGIC SPECIFICATION OF RELATIONS COMPUTED BY PROCESSES

The distinguishing feature of networks of process activations is that control of the sequencing of the activations of the processes is of no concern to the programmer; it is implicit in the way processes are connected by channels in the network. The primitive operations on the channels have been chosen in such a way that the programmer can regard each process as computing a relation between the histories of the channels to which the process is connected. We use here history in Kahn's [LPP] sense: the set of all data items that have existed in the channel at any time during the computation. This set is ordered as follows.

Case I: x and y have been simultaneously present in the channel. In this case, x before y in the history if x was in front of y in the channel.

Case II: x and y were never simultaneously present in the channel. In this case, x before y in the history if x was in the channel at an earlier moment than y was.

Because the relation between histories as computed by each process separately is central to our understanding of the network of processes as a whole, it is natural to express each such relation separately in a formal definition. As formal system we choose the clausal form of first-order predicate logic.

We represent histories by terms. As variables we use u, v, w, x, y, z, possibly with subscripts. In our examples the constants are numbers or the symbol 'eof', which stands for a special kind of history. The only thing we need to assume about eof is that it contains no data to be processed. More typically, a history is a term of the form x:y, where x is a number and y is a history and is that part of x:y that comes after x.

The relation computed by the ADD process of Box 2.1 is defined as the least model of the following clausal sentence:

```
           { add(eof, eof)
(3.1)...   , add(x:eof, x:eof)
           , add(x12:y, x1:x2:x) ← sum(x1,x2,x12) & add (y,x)
           }
```

where sum is a 'built-in' relation: the sentence is considered to contain the clause 'sum(a,b,c)' for all numbers a, b, and c such that a + b = c.

4. DERIVATIONS AND COMPUTATIONS

We have given a syntax for expressing definitions of relations. It is now time to see how to use such definitions; for example, to be able to use (3.1) for showing that

add(9:5:1:eof, 5:4:3:2:1:eof)

is an instance of the relation defined in (3.1). Such instances are defined by means of *derivations*.

Suppose that we are given the input history 5:4:3:2:1:eof and that we want to use (3.1) to obtain the corresponding, as yet unknown, output history w. We write the *goal statement*

← add(w, 5:4:3:2:1:eof)

We note that the third clause is applicable, which says that the above goal statement is solvable if we can solve

← sum(5,4,x12) & add(w1, 3:2:1:eof)

where w = x12:w1 . We assume that sum(5,4,x12) is solvable immediately with x12 = 9 . The remainder of the derivation is the following sequence of goal statements:

← add(w1, 3:2:1:eof)

← sum(3,2,x12) & add(w2, 1:eof)

(using the third clause and having set w1 = x12:w2)

← add(w2, 1:eof)

(having set x12 = 5)

□

which is the empty goal statement obtained by using the second clause of (3.1) and having set w2 = 1:eof.

The empty goal statement ends the derivation with success. The net result of the substitutions w = 9:w1, w1 = 5:w2, w2 = 1:eof is w = 9:5:1:eof which is according to (3.1) the output history corresponding to the input history 5:4:3:2:1:eof.

For us the most important property of derivations is the following. Let P be a set of definite clauses and let there be a derivation from $\leftarrow$ A to $\Box$ and let θ be the accumulated product of the successive substitutions in the derivation. Then [APT] each variable-free instance of Aθ is logically implied by P. It is in this sense that we can say that *results of derivations are logical implications of definitions.* We discuss next how derivations may be interpreted as either sequential computations of procedure-oriented programs or as Parallel computations of networks of process activations. In both cases the interpretation guarantees that *results of computations are logical implications of procedure or process declarations.* In this way the model theory of first-order predicate logic provides a denotational semantics for sequential programs, which was pointed out in [SPL] where the relationships with the fixpoint approach were discussed. The process interpretation of logic, which is explained in this paper, shows that the results of [SPL] also apply to the denotational semantics of Parallel programs.

The 'procedural interpretation' [PLPL, LPS] of logic shows that derivations are similar to computations, and that definite clauses are similar to procedure definitions. The details of the latter similarity are as follows.

The conclusion of a clause is the procedure heading. The predicate symbol in the conclusion is the identifier of the procedure being defined; its arguments are the formal parameters of the procedure definition. The premiss of the clause is the body of the procedure. Each atomic formula of the premiss is analogous to a procedure call.

Before discussing the similarity between derivations and computations, we review what are, in our view, computations in the execution of a procedure-oriented program. Such a computation is a sequence of states of a stack of procedure calls. The transition from one state to the next is obtained by procedure invocation: the replacement of the call at the top of the stack by the body of a matching declaration. Part of the matching process is the replacement of the formal parameters in the procedure heading by the corresponding actual parameters in the procedure call. The computation terminates when the stack is empty.

The set of possible successors of a given goal statement in a derivation depends on the selected atom of that goal statement. In the procedural interpretation we regard goal statements and premisses as ordered sets, in which the leftmost goal is always the selected atom. When we identify goals with procedure calls, it is clear that the successive goal statements of a derivation can be identified with the successive states of the stack during a computation of a procedure-oriented program.

5. THE PROCESS INTERPRETATION

A procedure has a definition which is distinct from its zero or more activations, each of which can be identified in the stack as the remains of a body. So also a process has a definition which is distinct from its activations in a network. We have already shown how to express in logic the definition of a process. It remains to complete what we call the *process interpretation of logic* by showing how to express a network of activations which execute according to Kahn's model of computation.

Our starting point is the procedural interpretation, which models states of a *sequential* computation by a *single* stack. In Kahn's model states of a *parallel* computation are *networks* of process activations, each of which carries out a sequential computation. We adopt from the procedural interpretation the representation of the state of a computation by a goal statement. The difference in the process interpretation is that the goal statement represents not a single stack, but a network of process activations. Because each of the process activations executes a sequential computation, it is represented by a stack. As a result, *in the process interpretation, a goal statement is interpreted as a network of stacks connected by channels with contents as given by the state of the computation being represented.*

We will give rules for reading off from the goal statement which activations are connected by a channel, what its direction is, and what its contents are. We first show an example of a logic derivation representing the successive states of a network of processes following the definitions of Box 2.1 as they perform balanced addition on the sequence of numbers 5, 4, 3, 2, 1. The relation computed by the processes are

{ add(eof, eof), add(x:eof, x:eof)

, add(x12:y, x1:x2:x) ← $\boxed{\text{sum(x1,x2,x12) \& add(y,x)}}$

(5.1)... , sigma(0:eof,eof), sigma(x:eof,x:eof)

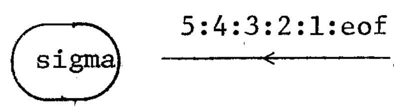
, sigma(z,x1:x2:x) ← $\boxed{\text{sum(x1,x2,x12) \& sigma(z, x12:y)}}$

& $\boxed{\text{add(y,x)}}$

}

The first goal statement of the derivation is:

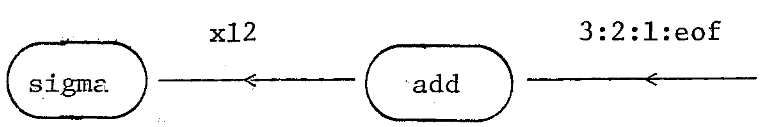
← $\boxed{\text{sigma(z, 5:4:3:2:1:eof)}}$

The corresponding network is: 

. We now continue to list goal statements of the derivations with comments explaining their process interpretation. Matching with the last clause for sigma gives:

← $\boxed{\text{sum(5,4,x12) \& sigma(z,x12:y)}}$ & $\boxed{\text{add(y,3:2:1:eof)}}$

There are now two process activations, connected in a network as follows

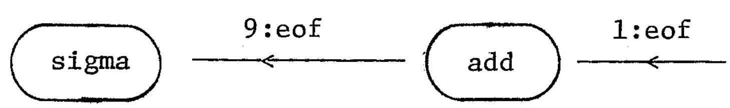


The fact that there are two stacks of goals to be executed in parallel, is copied from the premiss of the third clause for sigma. The connection between the two follows from the fact that the input history of sigma is x12 (which is going to be 5+4) followed by the output history y of add. By the definition of history of a channel as the sequence of all data items that are ever present in the channel, it follows that *there is a channel directed from add to sigma containing 5+4 in the present state.*

The goals sum and add can now be replaced in either order or simultaneously, giving

$$\leftarrow \boxed{\texttt{sigma(z,9:x12,y)}} \ \& \ \boxed{\texttt{sum(3,2,x12) \& add(y,1:eof)}}$$

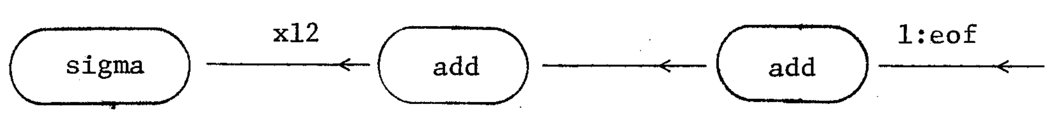
This is interpreted as the network



Both processes now have sufficient input to execute. We also execute the goal sum(3,2,x12) which belongs to the sequential code of add. After executing in any order sum and sigma, we obtain

$$\leftarrow \boxed{\texttt{sum(9,5,x12) \& sigma(z,x12:y1)}} \ \& \ \boxed{\texttt{add(y1,y)}} \ \& \ \boxed{\texttt{add(y,1:eof)}}$$

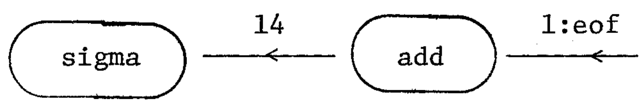
This is interpreted as the network:



Only the rightmost process has enough input. Hence

← sigma(z,14:y1) & add(y1,1:eof)

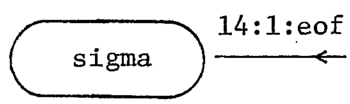
with network



Notice that in our formulation a stopped process vanishes. Again only the rightmost process has enough input:

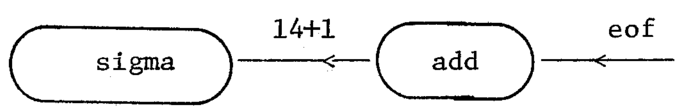
← sigma(z,14:1:eof

with network



← sum(14,1,x12) & sigma(z,x12:y) & add(y,eof)

has network



← sigma(z,15:eof)

Now the second clause for sigma derives the empty goal statement and hence finishes the derivation/computation. The resulting substitution for z in this goal statement, and also in the initial goal statement, is 15:eof.

After having seen examples of all its features, it is now time to give explicitly the *process interpretation of logic*.

a) Cyclical processes are defined as relations among histories, which need not be finite. The definition is inductive where the induction step refers only to finite subsequences of the histories involved. The induction step in the definition corresponds to one cycle in the execution of the cyclical process. If the histories are finite, then the inductive definition has a basis.

b) For the purpose of the process interpretation, the premisses of the clauses are partitioned into stacks. Each stack corresponds to the state of a sequential computation. Hence, in the definition of a static process, where the body is a single sequential computation, there is only one stack. In the definition of a dynamic process, when the body specifies parallel execution of process activations, there is more than one stack: one for each process activation.

c) The goals of a goal statement consist of a number of stacks, one for each process activation in the corresponding network. If two stacks share a variable, then the corresponding activations share a channel. In one of the stacks the term containing the shared variable consists of that variable only, say u. This stack is the activation of the producer process. In the other stack the term containing the shared variable has the form $t_1:\ldots:t_n:u$. This stack is the activation of the consumer process. The terms

$t_1,\ldots,t_n$ are the contents of the channel; t_1 is received first, t_2 next, and so on. In case $n = 0$ the channel is empty and there is no way to tell in which direction the data flow.

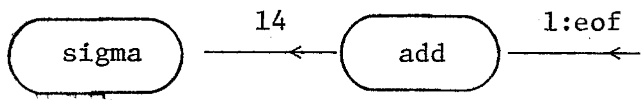
d) In Parallel computation, any activation is eligible for execution except those which are blocked in a get or eof operation on an empty channel. In logic, any goal of a goal statement may be selected when performing a derivation step. For only certain selections can such a derivation step be interpreted as a Parallel computation step: the goal must be in an activation which is eligible for execution. Once the activation has been determined, the selected goal is also determined as the leftmost. Because in logic there are no explicit 'get' operations, the rule which determines whether a process activation is ready for execution varies from case to case. For example, unless the next item is eof, always two items must present before the cycle of an 'add' or 'sigma' process activation can be initiated.

We have seen that every computation step of a Parallel program is a derivation step, but it is not so the other way around: the process interpretation disallows in general the selection of most of the goals of a goal statement. However, from the logical point of view the same result is obtained whatever goal is selected at each particular derivation step. Some selections, although disallowed by the process interpretation, are instructive variants on Kahn's model of computation.

For instance, take in the above example the goal statement

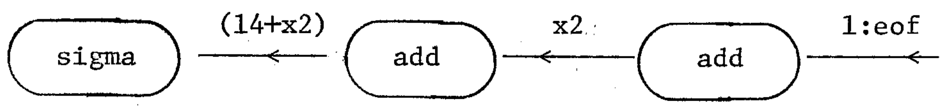
← sigma(z, 14:y1) & add(y1, 1:eof)

with network



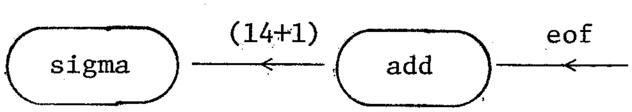
Sigma is not eligible for execution as it requires two items in the input channel. Suppose it would nevertheless be selected. Then the next goal statement would be

← sum(14,x2,x12) & sigma(z, x12:y) & add(y,x) & add(x2:x, 1:eof)



With selections admissable under the process interpretation, the second argument of add is always input and the first is always output. However, now (according to rule (c) above) the situation has been reversed in the channel between the two activations of add: x2 has been sent from left to right. x2 is a variable, not a data item, which also occurs elsewhere, for example in the input channel of sigma. Next time the rightmost activation of add sends an item it is not communicated in the usual way: the variable x2 will be instantiated with the item wherever the variable occurs. We see this by now executing the rightmost add:

← sum(14,1,x12) & sigma(z, x12:y) & add(y,eof)



The resulting state is now one which also occurs in the previous example. Apparently, process activations can be allowed (in logic) to run ahead of their input. The missing items appear as variables in the internal computations and are also sent as variables to where they should have come from. When the missing items are eventually produced, the variables are instantiated with the items and everything ends up in the situation as would also be obtained according to the rules of Parallel computation.

6. OTHER FORMALIZATIONS OF KAHN'S MODEL OF COMPUTATION

Logic is not the only non-imperative programming language in which Kahn's model of computation can be expressed: other examples include Lisp [FP] and Lucid [LLPL]. In this section we briefly explain the main idea of Lucid and compare data-flow programs in Lucid with those in logic.

Lucid is a language developed by E. Ashcroft and W. Wadge, and formally described in [LLPL]. Lucid has in common with logic that:

a) it is a single formal system which can be used both for writing programs and for reasoning about programs;

b) it is assertional: each statement is an axiom, hence can be understood without reference to an execution mechanism.

Here we are interested just in ULU, a subset of Lucid which can be regarded as a data-flow language. We will briefly and informally introduce ULU by using examples.

In ULU, expressions denote infinite sequences of data objects; functions transform sequences into sequences.

Let us assume that

$$\langle a_0, a_1, a_2, \ldots \rangle$$

is an infinite sequence where a_0 is the first element, a_1 is the second element, etc. .

x = <1, 2, 3, 4, ...>

three = <3, 3, 3, 3, ...>

T = <true, true, true, true, ...>

F = <false, false, false, false, ...>

P = <false, false, true, true, ...>

Here are some examples of Lucid functions:

first x = <1, 1, 1, 1, ...>

(note that first three = three)

next x = <2, 3, 4, 5, ...>

three fby* x = <3, 1, 2, 3, ...>

x + three = <4, 5, 6, 7, ...>

(this is the pointwise extension of addition)

x eq three = <false, false, true, false, ...>

x asa** P = <3, 3, 3, 3, ...>

(x asa P is a constant sequence corresponding to x at the smallest index where P is true).

if P then x else zero = <0, 0, 3, 4, ...>

---

* fby is pronounced "followed by".

** asa is pronounced "as soon as".

Consider now a simple ULU program to compute $[\sqrt{N}]$:

```
first x = one
first y = one
next  x = x + one
next  y = y + two * x + one
result  = (x-one) asa (y gt n);
```

if n = <20, 20, 20, ...>

then a solution for the above equation is:

```
       x  =  <1, 2, 3, 4, 5, ...>
       y  =  <1, 4, 9, 16, 25, ...>
  y gt n  =  <false, false, false, false, true, ...>
 x - one  =  <0, 1, 2, 3, 4, ...>
  result  =  <4, 4, 4, 4, 4, ...>
```

A solution for the balanced-addition problem can be formalized in ULU as follows:

```
(6.1)...     add (x) = if first x eq eof
                       then eof
                       else if first (next x) eq eof
                          then first x fby eof
                          else (first x + first(next x))fby add(next(next x)

(6.2)...   sigma(x) = if first x eq eof
                      then zero
                      else if first (next x) eq eof
                              then first x
                              else sigma (add (x))
```

As an illustration, if $x = <1, 2, 3, 4, 5, \text{eof} >$, then the equation defining sigma implies:

$$
\begin{aligned}
\text{sigma}\ (x) &= \text{sigma}\ (\text{add}(<1, 2, 3, 4, 5, \text{eof}\ >)) \\
&= \text{sigma}\ (<3, \text{add}(\ 3, 4, 5, \text{eof}\ >)\ >) \\
&= \text{sigma}\ (<3, 7, \text{add}(<5, \text{eof}\ >)\ >) \\
&\quad \cdots \\
&= \text{sigma}\ (<3, 7, 5, \text{eof}\ >) \\
&= \text{sigma}\ (\text{add}(<10, 5, \text{eof}\ >)) \\
&= \text{sigma}\ (<15,\ \text{eof}\ >\ ) \\
&= \ 15, \text{eof}\ >
\end{aligned}
$$

We will illustrate how to transform an ULU program step by step into an equivalent logic program. Consider the ULU definition of sigma (6.2).

Step 1: We rewrite (6.2) as conditional equations,

$$
\begin{aligned}
&(6.3)\ldots \quad && \text{sigma}\ (x) = \text{zero} && \leftarrow \ \text{first}\ x\ \text{eq}\ \text{eof} \\
&(6.4)\ldots \quad && \text{sigma}\ (x) = \text{first}\ x && \leftarrow \neg\ (\text{first}\ x\ \text{eq}\ \text{eof}) \\
& && && \ \ \&\ \neg\ \text{first}\ (\text{next}\ x)\ \text{eq}\ \text{eof} \\
&(6.5)\ldots \quad && \text{sigma}\ (x) = \text{sigma}(\text{add}(x)) && \leftarrow \neg\ (\text{first}\ x\ \text{eq}\ \text{eof}) \\
& && && \ \ \&\ \neg\ (\text{first}\ (\text{next}\ x)\ \text{eq}\ \text{eof})
\end{aligned}
$$

Step 2: With every non-boolean function $f(x_1, \ldots, x_n)$ we associate a relation $F(\text{result}, x_1, \ldots x_n)$, where $\text{result} = f(x_1, \ldots, x_n)$; also, with every boolean function $b(x_1, \ldots, x_n)$, we associate a relation $B(x_1,\ldots,x_n)$. In our example, we associate the predicate symbols SIGMA, ADD, FIRST, NEXT and EQ, with the function symbols sigma, add, first, next and eq, respectively.

<u>Step 3</u>: We rewrite the conditional equations (6.3) - (6.5) using the predicates above:

(6.6)... SIGMA(zero, x) $\leftarrow$ FIRST(eof, x).

(6.7)... SIGMA(x_0, x) $\leftarrow$ FIRST(x_0, x) & $\neg$ EQ(eof, x_0)
& NEXT(x_t, x) & FIRST(eof, x_t).

(6.8)... SIGMA(z, x) $\leftarrow$ FIRST(x_0, x) & $\neg$ EQ(eof, x_0)
& NEXT(x_t, x) & FIRST(x_1, x_t) & $\neg$ EQ(eof, x_1)
& ADD(x', x) & SIGMA(z, x').

The above program is *representation-independent* because no commitment has been made concerning data representation.

<u>Step 4</u>: Let us choose to represent by "eof" any sequence we are not going to process, and let us use the right-to-left-binding operator ":" to represent sequences according to the definition

<sequence> :: = eof | number : <sequence>

The following clauses provide an *interface between the representation-independent program* (6.6) - (6.8) *and our chosen data representation:*

(6.9) ... FIRST(eof, eof).

(6.10) ... FIRST(x_0:eof, x_0:x_t)

(6.11) ... NEXT(x_t, x_0:x_t) .

<u>Step 5</u>: Using (6.10) to resolve the first occurence of FIRST in (6.7) we obtain:

(6.12) $\mathrm{SIGMA}(x_0{}':\mathrm{eof},\ x_0{}':x_t{}') \leftarrow \neg\, \mathrm{EQ}(\mathrm{eof},\ x_0{}':\mathrm{eof})\ \&\ \mathrm{NEXT}(x_t, x_0{}':x_t{}')$
$\&\ \mathrm{FIRST}(\mathrm{eof},\ x_t)$.

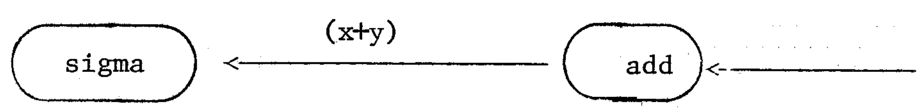
Resolving all occurrences of FIRST and NEXT in (6.6) - (6.8) by using (6.9) - (6.11), we obtain a *representation-dependent logic program:*

(6.13) SIGMA(zero, eof)

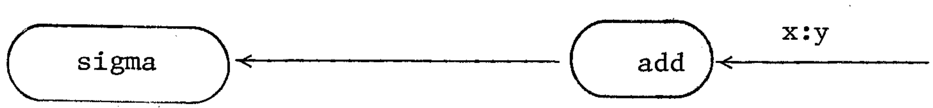
(6.14) $\mathrm{SIGMA}(x_0:\mathrm{eof},\ x_0:\mathrm{eof}) \leftarrow \neg\, \mathrm{EQ}(\mathrm{eof},\ x_0:\mathrm{eof})$

(6.15) $\mathrm{SIGMA}(z,\ x_0:x_1:x_t) \leftarrow \neg\, \mathrm{EQ}(\mathrm{eof},\ x_0:\mathrm{eof})\ \&\ \neg\, \mathrm{EQ}(\mathrm{eof},\ x_1:\mathrm{eof})$
$\&\ \mathrm{ADD}(x',\ x_0:x_1:x_t)\ \&\ \mathrm{SIGMA}(z,\ x')$.

Assuming that the clauses will be considered in textual order, and considering that our program does not involve backtracking, we may drop the predicates EQ from (6.13) - (6.15), obtaining a program which is almost identical to (5.1), obtained directly from the informal description of the problem. The differences arise from the fact that now we used "zero" where previously we had used 0:eof, and also from the fact that in (5.1) we defined sigma in such a way that, as soon as it reads in two numbers x and y, it changes itself to



while in the ULU program (and consequently in (6.13) - (6.16)) it changes itself to



7. CORRECTNESS FOR TERMINATING PARALLEL COMPUTATIONS

For verification of Parallel programs expressed in logic we use the method proposed and demonstrated by Clark and Tärnlund [FOT]. According to their method the definitions used for computation are proved as theorems from specifications in first-order predicate logic. Unlike the definitions, the specifications are not necessarily in clausal form. The type of verification obtained is partial correctness: if the computation terminates, the instance of the relation derived by the computation is also an instance of the relation defined by the specification.

We illustrate the method by a verification of example (5.1), our logic version of the Parallel program in Box 2.1. As specification for sigma we use:

(6.1)... sigma(0:eof,eof) $\wedge$

(6.2)... $\forall$x,y,z. sigma(z,x:y) $\leftrightarrow$ $\exists$z1.sigma(z1,y) & sum(x,z1,z) $\wedge$

(6.3)... $\forall$x,y. add(x,y) $\rightarrow$ $\exists$s.sigma(s,x) & sigma(s,y)

According to the specification, the sum in sigma is obtained in the normal way. We can prove each of the clauses for sigma in (5.1) as a theorem from the specification with the help of some general arithmetical knowledge, such as the properties of 'sum'. We prove in effect that, for associative addition of reals (it is this property that rounding errors typically invalidate), balanced addition is equivalent to naive addition.

<u>Proof</u> of sigma(z,x1:x2:x) ← sum(x1,x2,x12) & sigma(z,x12:y) & add(y,x):

sum(x1,x2,x12) & sigma(z,x12:y) & add(y,x) → (6.2)

∃s'.sum(x1,x2,x12) & sigma(s',y) & sum(s',x12,z) & add(y,x) → (6.3)

∃s'.sum(x1,x2,x12) & sum(s',x12,z) & sigma(s',y) & sigma(s',x) → (property of sum)

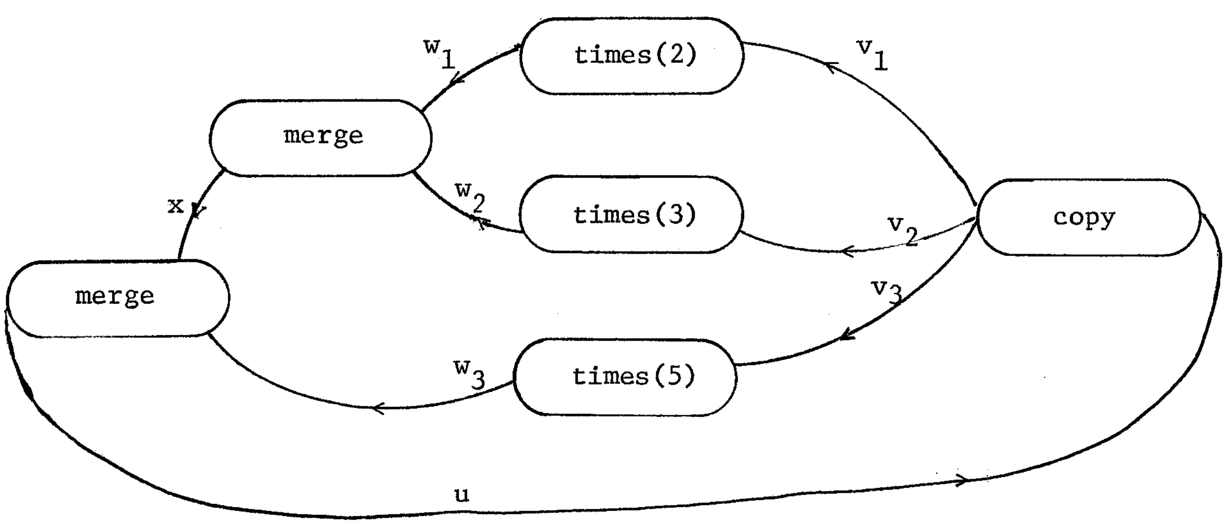
∃s.s'.sum(s',x2,s) & sum(s,x1,z) & sigma(s',x) → (6.2)

∃s.sum(s,x1,z) & sigma(s,x2:x) → (6.2)

sigma(z,x1:x2:x).

8. CORRECTNESS FOR NONTERMINATING PARALLEL COMPUTATION

We introduce this section with an example involving infinite histories. The example is from Kahn and McQueen [NPP]. It is a program to solve Hamming's problem: to make a computer print in increasing order all positive integers having only 2, 3, and 5 as prime factors. All processes are static, so the network remains unchanged during the entire computation. The network is shown below. All channels are initially empty except for the integer 1 in u.



times(i): output history is, element by element, i times the input history;

merge: output history is the result of merging the input histories, where duplicates are suppressed;

copy: each output history equals the input history; as a side effect its input is printed.

These relations are specified as follows:

```
H = {times(w1:w,v1:v,u) ← prod(v1,u,w1) & times(w,v,u)
    ,merge(x:u,x:w1,x:w2) ← merge(u,w1,w2)
    ,merge(x1:u,x1:w1,x2:w2) ← x1 < x2 & merge(u,w1,w2)
    ,merge(x2:u,x1:w1,x2:w2) ← x1 > x2 & merge(u,w1,w2)
    ,copy(x:v1,x:v2,x:v3,x:u) ← copy(v1,v2,v3,u)
    }
```

The network can be read off from the following goal statement:

```
← copy(v1,v2,v3,1:u)
& times(w1,v1,2) & times(w2,v2,3) & times(w3,v3,5)
& merge(x,w1,w2) & merge(u,x,w3)
```

The set of sequences that can be printed out by the program is characterized by the predicate 'result' as defined in the clause below and supported by the clause in H.

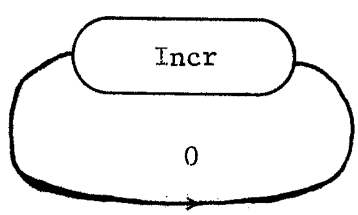
```
Result(1:u) ← copy(v1,v2,v3,1:u)
     & times(w1,v1,2) & times(w2,v2,3) & times(w3,v3,5)
     & merge(x,w1,w2) & merge(u,x,w3)
```

Let us briefly review the main features of the fixpoint semantics for logic programs as developed in [SPL]. With a set P of definite clauses there is associated a monotone mapping T from interpretations to interpretations such that I is a Herbrand model of P iff $I \supseteq T(I)$. Hence the least fixpoint of T is the set of all variable-free atomic formulas which are true in all Herbrand models of P and it is also the set of all possible results of derivations (and hence finite computations) from P. The least fixpoint semantics of [SPL] is adequate for terminating computation.

However, for the Hamming program the denotation of Result in the least fixpoint of T is empty. And indeed, there is no finite computation with ←Result(x) as first goal statement. But surely there must be a meaningful relationship between the Hamming program and the infinite sequence substituted for x by the infinite derivation starting with ←Result(x). It is reasonable to require of a semantics of logic programming to establish such a relationship, and therefore the least fixpoint semantics of [SPL] requires at least an extension.

That this can be done for nonterminating computations in general has been shown in [ITS]. Without going into any details, we will here illustrate the main idea with the simplest possible example.

Consider the following network consisting of a single process Incr which continues reading a number, writing it, incrementing it by one, and placing the result on its output channel, which is also the input channel. This channel contains initially a single number 0.



The definition of the relation computed by Incr is:

$$\text{Incr}(x1{:}y,\ x{:}z) \leftarrow \text{sum}(x,1,x1)\ \&\ \text{Incr}(y,z)$$

The network is specified by the goal statement $\leftarrow \text{Incr}\ (x,0{:}x)$. This goal statement initiates an infinite computation, which substitutes for x successively $1{:}\ldots{:}n{:}x_n$ for $n = 1,2,\ldots$.

Let us see what the sentence

$$\begin{aligned} P = \{&\text{Incr}(x1{:}y,x{:}z) \leftarrow \text{sum}(x,1,x1)\ \&\ \text{Incr}(y,z) \\ &,\text{Omega}(0{:}x) \leftarrow \text{Incr}(x,0{:}x) \\ &\} \end{aligned}$$

says about the result $0{:}1{:}2{:}\ldots$ of the computation starting with $\leftarrow$ Omega(x). The result certainly is not in the denotation of Omega in the least fixpoint of T, the transformation associated with P, which is empty. One reason why we cannot expect otherwise is that the underlying domain, the Herbrand universe, contains only finite terms. In fact, with this domain, the denotation of Omega in any fixpoint of T is empty. Thus, if we are to give a semantics for infinite computations we must consider infinitary Herbrand universes containing all terms of the usual Herbrand universe plus the infinite terms that can be regarded as limits of monotone sequences of finite terms.

Now the denotation of Omega in the least fixpoint of T, when taken in the infinitary Herbrand universe, is also empty. This time, however, the denotation of Omega in the greatest fixpoint of T is exactly what we want, namely the sequence 0:1:2:... of all natural numbers.

The results of [ITS] can be used to verify P. In the first place, if we can show that no derivation from P exists with ← Omega(x) as first goal statement and x some variable-free term not equal to the omega sequence, it would follow [NAF, APT] that any such Omega(x) is false in all models of, and hence in the greatest model of

```
P' = {Incr(x1:y,x:z) → Sum(x,1,x1) & Incr(y,z)
     ,Omega(0:x) → Incr(x,0:x)
     }
```

which is the converse of P. If we can show that all derivations from P starting from ← Omega(0:1:2:...) are infinite then it would follow that Omega(0:1:2:...) is true in the greatest model of P', provided that the infinitary Herbrand universe is the underlying domain. In [APT] the greatest model of P' is related to the greatest fixpoint semantics of P. This example suggests that greatest fixpoints characterize infinite computations in a way that is similar to the way least fixpoints characterize finite computations.

9. RELATED WORK

Kowalski [PLPL, LPS] introduced the procedural interpretation and discussed the possibility of coroutining among goals. Bruynooghe and Clark [CRLP] have pursued this coroutining much further. They arrived at a model of computation of great generality, having among others as special cases both Kahn's model and lazy evaluation. Moreover, Clark and McCabe have implemented this in a system called IC-Prolog.

This paper has in common with the work of Bruynooghe and Clark that coroutining computations are obtained by a suitable choice of selected atom. Another approach is taken by Pereira and Monteiro [PCLP] who assume that, for efficiency reasons, the leftmost goal is always the selected atom. They obtain the equivalent of parallel execution by a systematic and very elegantly conceived transformation of the logic definition.

10. REFERENCES

[LPP] G. Kahn "The semantics of a simple language for parallel programming" Proc. IFIP 74.

[NPP] G. Kahn and D.B. McQueen "Coroutines and networks of parallel processes" Proc. IFIP 77.

[FOT] K.L. Clark and S.A. Tärnlund "First-order theory of data and programs" Proc. IFIP 77.

[PLPL] R.A. Kowalski "Predicate logic as programming language" Proc. IFIP 74.

[CRLP] M. Bruynooghe and K. Clark "A control regime for logic programming" (in preparation).

[GDM] A. Colmerauer "Grammaires de metamorphose" in L. Bolc (ed). "Natural language communication with computers" Springer LNCS.

[ICP] K.L. Clark and F.G. McCabe "IC-Prolog manual" Department of Computing and Control, Imperial College.

[MOL] J.A. Robinson "A machine-oriented logic based on the resolution principle" J.ACM 12 (1968).

[ATP] D.W. Loveland "Automated Theorem-Proving" North Holland 1978.

[APT] K. Apt and M.H. van Emden "Contributions to the theory of logic programming" Research Report CS-80-12, Dept. of Computer Science, University of Waterloo.

[LPS] R.A. Kowalski "Logic for Problem-Solving" North Holland-Elsevier, 1979.

[SPL] M.H. van Emden and R.A. Kowalski "The semantics of predicate logic as programming language: J.ACM 23 (1976).

[ITS] H. Andreka, M. van Emden, I. Nemeti and J. Tiuryn "Infinite term semantics for logic programs" (in preparation).

[PCLP] L.M. Pereira and L.F. Monteiro "The semantics of parallelism and co-routining in logic programming" Colloquium on Mathematical Logic in Programming, Salgótarján, Hungary, Sept. 1978.

[FP] P. Henderson "Functional Programming" Prentice-Hall, 1980.

[LLPL] E.A. Ashcroft and W.W. Wadge "Lucid - A Logical Programming Language" Academic Press (to be published).