AN OR-PARALLEL TOKEN MACHINE

Seif Haridi and Andrzej Ciepielewski
Department of Telecommunication and Computer Systems
Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

achine model consisting of a limited number of processors, a token pool and storage is defined. A token represents the state of a process, which in turn ecutes a branch in the search tree of a logic program. Tokens in the token l correspond to processes which are ready for execution but not allocated a cessor. Processors execute processes as presecribed by the tokens and ate new tokens. A processor executes a compiled form of the programs. We ine the translation of programs into sequences of abstract machine instruc-ns and define the interpretation cycle of a processor.

Introduction

n clause programs can be executed in different modes without changing their ning up to termination. The most common mode is Prolog's left-to-right ection of subgoals and depth-first traversal of the search tree using back-cking. Instead of a sequential exploration of alterantive solutions, the rch tree can be traversed in parallel. This mode has been lately called parallelism. It can be implemented on a single processor [RoSi], but comes st to its right when a large number of processors is used.

goal of our research is a multiprocessor architecture for efficient execu-n of Or-parallelism. We share this goal with a growing number of research-: [CoKi], [EKM], [Po], [UmTa] and [FNM]. We have already defined an inter-ter for Or-parallelism and investigated the feasibility of using structure ring in a distributed implementation [CiHa83A,CiHa83B]. In this inter-ter, we have studied, in detail, the problem of managing simultaneously eral binding environments. The interpreter evaluates programs in their tract source form and creates a computation process for each alternative deterministic branch.
this paper we define an abstract machine model consisting of a limited ber of processors, a pool of tokens and a storage. The unlimited number of cesses in our interpreter is now mapped onto the finite number of proces-s. A processor executes a compiled form of programs. Subgoals are ected in a specific order as defined by the sequence of instructions. We cribe the translation of logic programs into sequences of the abstract hine instructions and define the semantics of the instructions. The truction set we define here is similar to that of the sequential machine cribed in [HaSa]. Finally, we discuss methods for controlling the amount parallelism and compare our machine with other proposals.

Abstract machine model

ogic program consists of an initial call and a set of relations. A rela-n consists of a number of clauses, where each clause is either an assertion an implication. An implication has a head and a body. The body is a literal a conjunction of literals.

Ex 1: The following is a program for list-permutation; it consists of two relations: p(ermute) and d(elete):

```
1.  p([],[]).
2.  p(xs,[y|ys]) ←— d(xs,y,zs) & p(zs,ys).

1.  d([x|xs],x,xs).
2.  d([x|xs],y,[x|ys]) ←— d(xs,y,ys).
```

and a possible initial call:

```
p([1,2],ys)
```

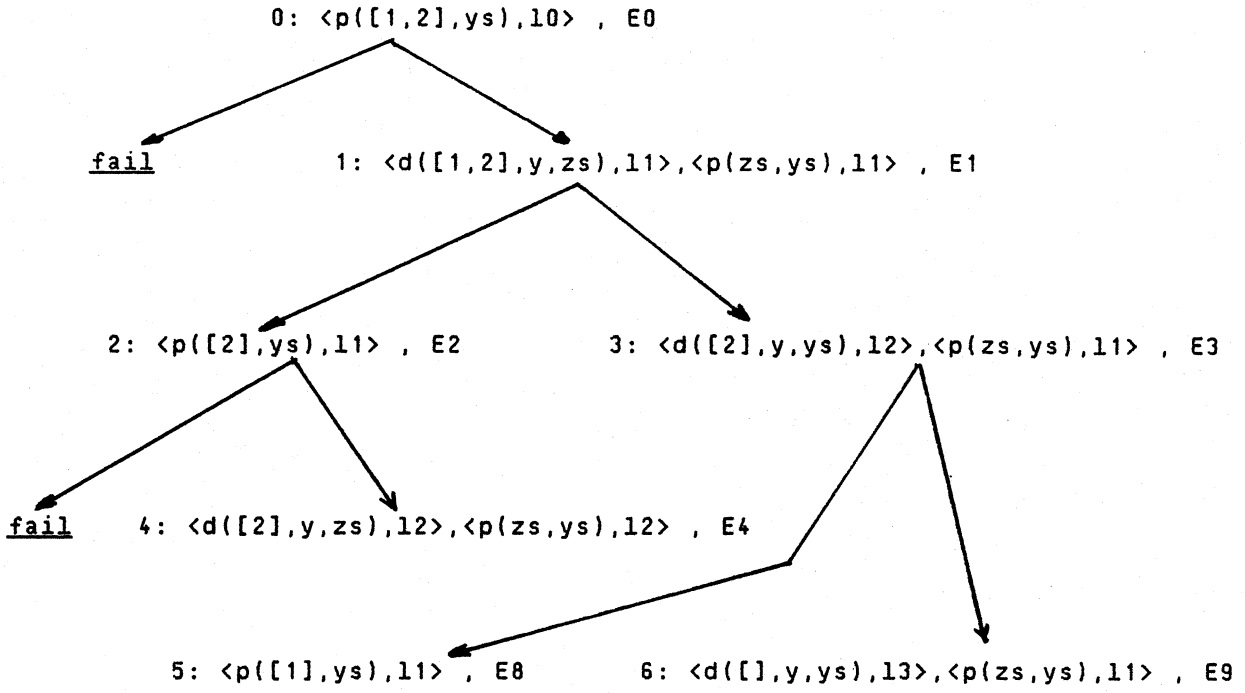
Let us denote the i'th clause of a relation r by r.i, then p.1 and d.1 are assertions, and p.2 and d.2 are implications.

Execution of a program can be described by a search tree. A node in such a tree represents the state of a subcomputation: a sequence of goals and a binding environment:

$$\langle Goal1\rangle,\langle Goal2\rangle,\ldots,\langle Goaln\rangle \ , \ E_i$$

A binding environment consists of contexts containing the values of the variables in the invoked clauses, one context for each clause invocation. Children of a node represent the states reached after executing a goal in the given state.

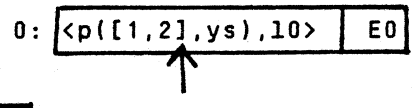
Ex 2: The following figure illustrates the initial four levels of the search tree corresponding to the program in Ex 1. Notice that each goal consists of a literal and a context name l_i which identifies the context containing the values of the variables occuring in the literal. In the figure, the environments E_i are not shown, instead literal substitution of values for variables is used when possible.

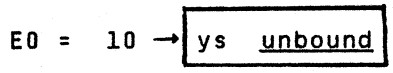


In the interpreter described in [CiHa82A,CiHa83B], a process is created for the root of the search tree. It starts a child process for each clause of the relation chosen by the current goal and then terminates. A newly created process executes unification and if it fails, it terminates, otherwise it creates children processes and then terminates. A branch of computation terminates successfully when there are no more goals to solve. A solution can be extracted from the binding environment.

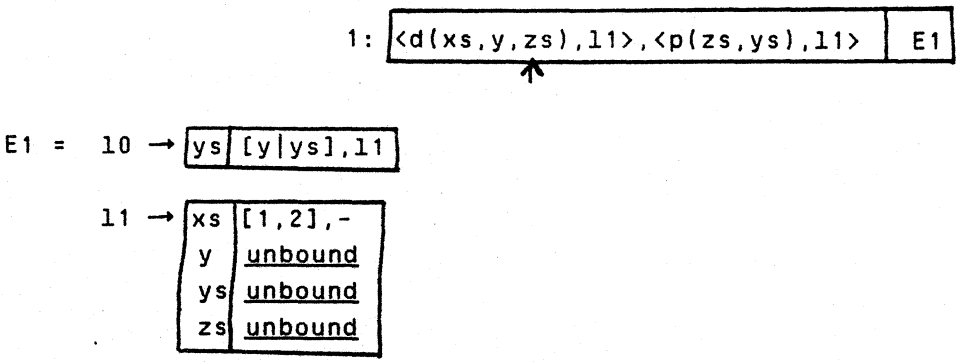
Ex 3: Four snapshots of possible generations of processes for the search tree in Ex 2, where the state of each process is shown. Processes are about to perform a unification step. Current goals are indicated by upward arrows. Environments are also shown in detail where the value of a variable is either unbound or a pair: (Source Term,Context Name).

Snapshot 1: the process corresponding to node 0:

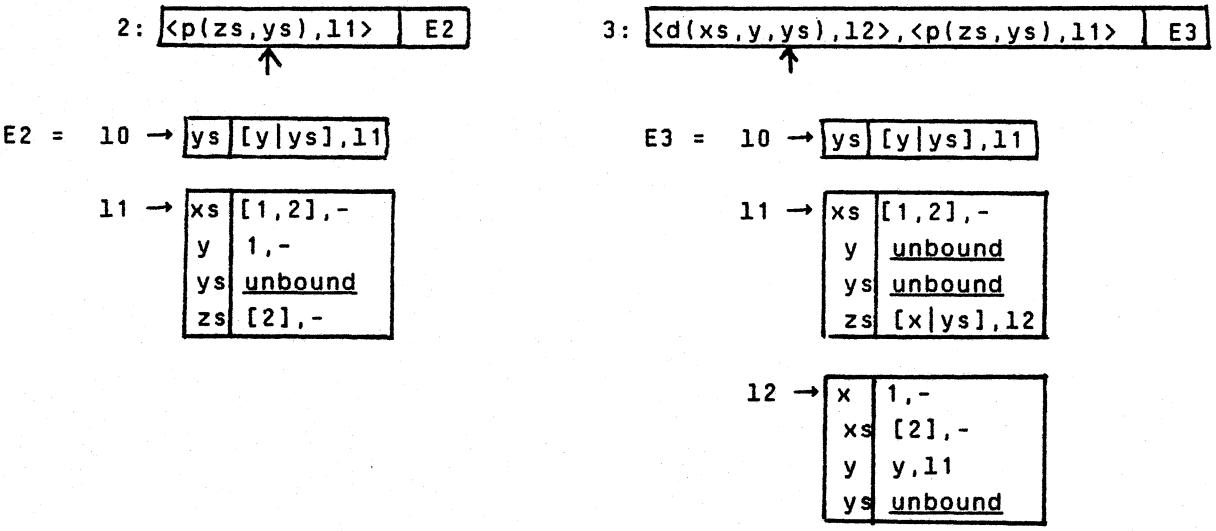




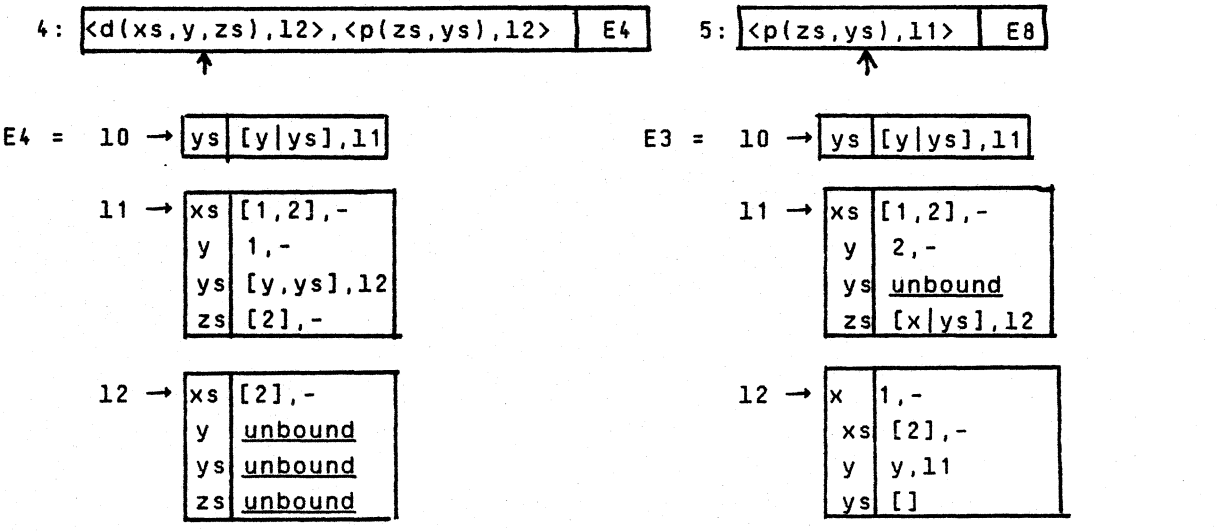
Snapshot 2: the process corresponding to node 1:



Snapshot 3: the processes corresponding to nodes 2 and 3:



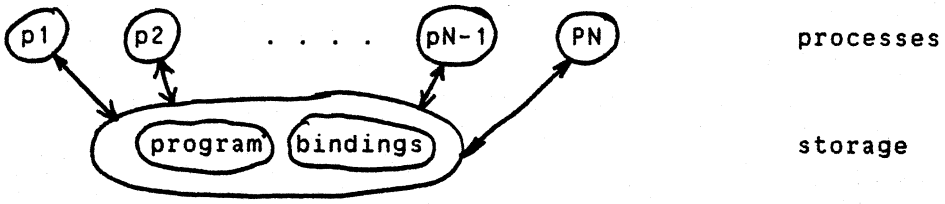
Snapshot 4: the processes corresponding to nodes 4 and 5, we assume that the process corresponding to node 6 and its children have already terminated.



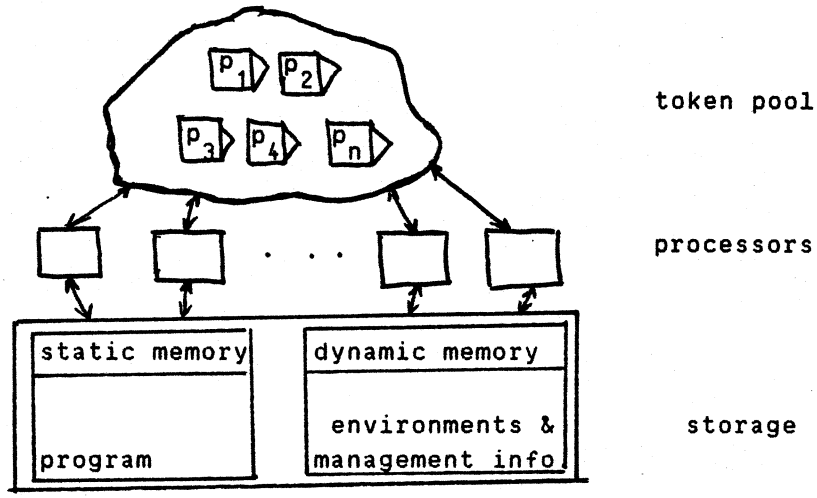
Efficient methods for maintaining a separate address space for each binding environment are described in [CiHa83A,CiHa83B]. For the rest of this paper it

is enough to realise that a variable can be accessed or updated through the unique name: <Environment name,Context name,Variable name>.

A computation described by our earlier interpreter may be visualised as an unlimited number of processes and a storage for binding environments and programs.



In the machine model we present here, the unlimited number of processes is mapped onto a finite number of processors. On this conceptual level we can picture the machine as consisting of a token pool, a set of processors and a storage. Storage is divided into a static memory for programs and dynamic memory for the binding environments and other management information. Tokens in the pool represent processes which are ready for execution but are not allocated a processor. Processors execute processes as prescribed by the tokens and create new tokens. Processors communicate with the storage to access program and data.



The above abstraction is similar to the one presented by Darlington and Reeve in the description of ALICE [DaRe]. It is very useful for handling problems of parallel computations. Furthermore, it can be a starting point for many different architectures.

As mentioned above, the state of a process consists of a list of goals and a binding environment. Such a state will be represented in our machine by a token residing in the token-pool or in one of the processors, and by a possibly empty list of continuation frames residing in the dynamic memory.
A token consists of the following fields:

1. Literal reference (L),
2. Context name (C),
3. Environment name (E),
4. Continuation-Frame reference (CF) and
5. other information to be described later.

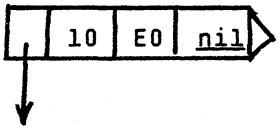
A continuation frame has the following fields:

1. Literal reference (L),
2. Context name (C) and
3. Continuation-Frame reference (CF).

In the next section, when the machine instructions are specified, the L-field in tokens and continuation frames will be a reference to an instruction.

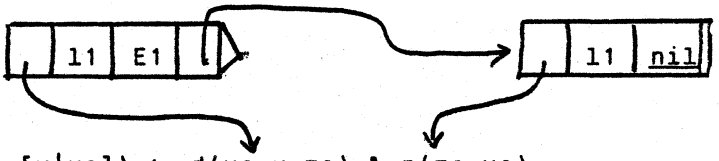
Literals of a clause are selected form left to right. This implies that the head of the goal-list is always the current goal and the tail are the remaining goals. The L and C fields of a token represent the current goal, whereas its continuation frames represent the remaining goals.
These ideas are illustrated by the following snapshots which correspond to those of Ex 3.

Snapshot 1: there is one initial token having the current goal <p([1,2],ys),l0>, and no continuation frame, i.e. field CF is nil.



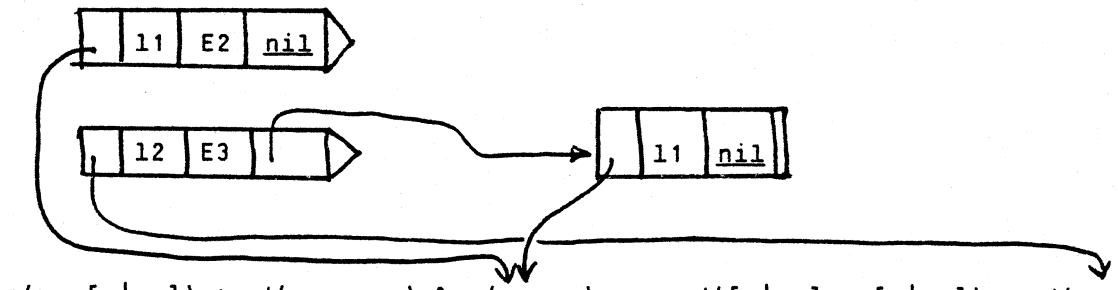
p([1,2],ys)

Snapshot2: one token with the current goal <d(xs,y,zs),l1>, the remaining goal <p(zs,ys),l1> is represented by a continuation frame.



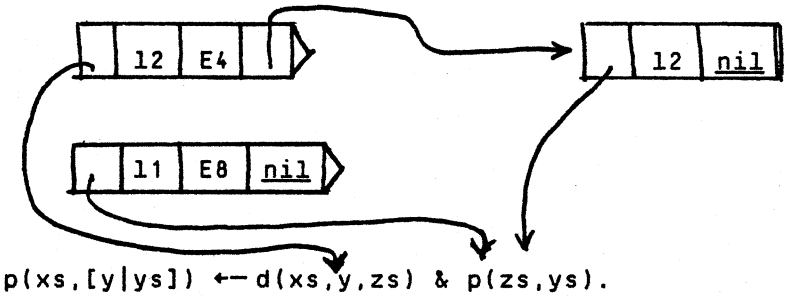
p(xs,[y|ys]) ← d(xs,y,zs) & p(zs,ys).

Snapshot 3: two tokens, the one corresponding to node 3 has a continuation frame.



p(xs,[y|ys]) ←d(xs,y,zs) & p(zs,ys). d([x|xs],y,[x|ys]) ←d(xs,y,ys).

Snapshot 4: evident.



p(xs,[y|ys]) ←d(xs,y,zs) & p(zs,ys).

Continuation frames are read-only data objects. This allows several tokens to share continuation frames.

3. Translation of programs into machine code

In this section, we describe the translation of Horn clauses into machine code. We give also a short summary of the instructions and their effect. The exact specification of the interpretaion cycle of a processor is given in Section 5. Notice that the machine code representation of terms is not given.

We use the following metavariables, which may be indexed, to range over basic syntactic entities:

Terms:	t,q,r,s.
Relation names:	R,S.

By ↑t and ↑R we mean a reference to the representation of the term t and to the relation (clause) R respectively.

An assertion having m variables:

$$R(t_1,\ldots,t_n)$$

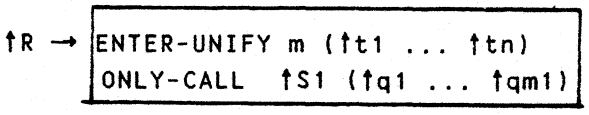
is translated into:

```
↑R → ENTER-UNIFY m (↑t1 ↑t2 ... ↑tn)
     RETURN
```

An implication with only one literal in its body and having m variables:

$R(t_1,\ldots,t_n) \leftarrow$
$\quad S_1(q_1,\ldots,q_{m1})$.

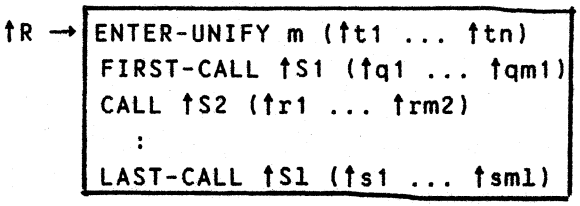
is translated into:



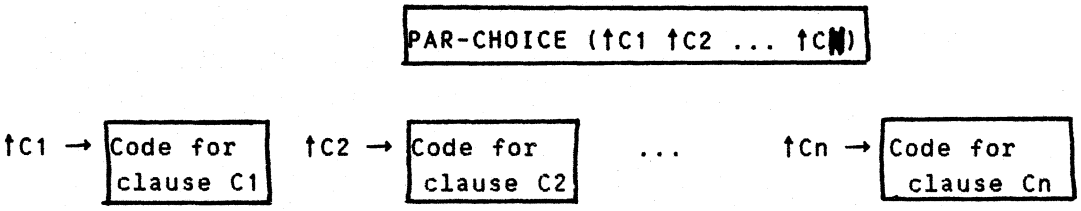
An implication with two or more literals in its body and having m variables:

$R(t_1,\ldots,t_n) \leftarrow$
$\quad S_1(q_1,\ldots,q_{m1})$ &
$\quad S_2(r_1,\ldots,r_{m2})$ &
$\quad\quad\vdots$
$\quad S_l(s_1,\ldots,s_{ml})$

is translated into:



A Relation R consisting of several clauses, C1 C2 ... Cn, is translated into:



A processor fetches a token and executes the instruction it refers to. After the instruction is performed the processor may create none, one or more tokens according to the interpreted instruction. As mentioned earlier every token has a list of continuation frames. The description that follows, of the instructions, will be relative to the token being interpreted, so "Remove first continuation frame" actually means to remove the first continuation frames from the list associated with the interpreted token.

(1) ENTER-UNIFY m (↑t1 ↑t2 ... ↑tn) :
Create a variable-context for m variables in current environment. Execute a unification step; the callers parameters are referred to in the interpreted token.

(3) RETURN :
Return control to the caller. The next instruction to be executed is stored in the first continuation frame.

(4) ONLY-CALL ↑S (↑t1 ... ↑tn) :
Transfer control and parameters to S; this instruction is used where there is exactly one literal in the body of a clause and therefore no continuation frames are created.

(5) FIRST-CALL ↑S (↑t1 ... ↑tn) :
Create a continuation frame; save next instruction in it and link it first in the continuation frame list; transfer control and parameters to S.

(6) CALL ↑S (↑t1 ... ↑tn) :
Remove the first continuation frame; link first a continuation frame referring to next instruction; transfer control to S.

(7) LAST-CALL ↑S (↑t1 ... ↑tn) :
Remove the first continuation frame; transfer control to S.

(8) PAR-CHOICE (↑C1 ↑C2 ... ↑Cn) :
Create n tokens each having its own environment; i.e. n parallel activities are initiated. The created tokens share the continuation frame list of the interpreted token.

4. Notational Conventions

The next section specifies the interpretation cycle of a processor. We present here the essential characteristics of the specification language used there. The language used may be considered as an imperative fraction of Meta-IV [BjJo].

4.1. Types of Objects

The elementary type NAT is the class of all natural numbers 0, 1, ... and the type BOOL is the class of truth values true and false. Elementary types like unbound is meant to be the singleton set with the element unbound.

Lists

The type of lists of objects each having the type A is denoted by
A*
The list of the objects e1, e2, ... ,en in this order is formed using
<e1,e2, ...,en>
An empty list is denoted by nil. The following operations apply to a list l where l = <e1,e2, ...,en>:
l[i] == ei (yields the i'th element of a list),
len l == n (the length of the list l)

Reference types

Let A be a type then

↑A

is the type of references (addresses) of objects of type A. If i has the type ↑A, then the operation i↑ returns the object a of type A referred by i otherwise it returns nil.

Cartesian products

The type of heterogenous n-tuples for which the first object is of type A1, the second object is of type A2, etc. is denoted by

A1 A2 ... An

An object of this type is treated as a list of length n.

Abstract Types

Abstract types of compound objects may be specified by means of the following rules:

(1) A = B1 | B2 | ... | Bn

This rule defines the abstract type named A (a type identifier) to be the union of the (disjoint) types defined by B1, B2, ..., Bn, where Bi are type identifiers or type expressions as defined above.

(2) A :: B1 B2 ... Bn

This rule defines the type A to be the type of A-tagged n-tuples of the type (B1 B2 ... Bn). An A-tagged n-tuple object is formed with the expression

mk-A(e1,e2,...,en)

where 'mk-' is the so called make constructor. The above expression generates the tuple <e1,e2,...,en> equipped with the tag 'A'.

(3) A = B1 B2 ... Bn

The same as rule (2) however tuples are not tagged.

4.2. Statements

A crucial statement used in the specification below is

```
(def x: e;
 S
)
```

where x is an identifier, e is an expression possibly having some side effect (e.g. a procedure returning a value), and S is a statement. The expression e is evaluated first, then all occurrences of x in S are replaced by the value returned by e, finally S is evaluated in this context. More generally in the construct

```
(def mk-A(x1,x2,...xn) = e;
 S
)
```

e is evaluated to yield an A-tagged n-tuple, the immediate components of which are then denoted by x1, x2, ..., xn in the evaluation of the statement S.

The other forms of statements are familiar from other imperative languages (Pascal etc.). For example sequential statement composition has the form:

(S1;S2; ...;Sn),

cases have the form:

cases e0 : (e1 → S1, e2 → S2, ... , en → Sn)

and the indexed iteration:

for i = m to n do S(i)

A definition of a procedure F returning a value in our specification language is assigned a type of the form:

F: B1 B2 ... Bn => B (n ≥ 0)

telling that F has n arguments that are of the types B1, B2, ..., Bn and returns a value of the type B.

If F does not return a value, i.e. is applied for its side effect only, F will get a type of the form

F: B1 B2 ... Bn =>

5. Specification of the processor cycle

The instructions introduced in the previous sections are executed by each processor. Here, we define the basic execution cycle of the processor and the exact meaning of the instructions.

5.1. Instruction set

A program consists of the initial call and a sequence of instructions.

```
Program = INIT-CALL Code
Code = Instruction*
```

There are following instructions:

```
Instruction = INIT-CALL | FIRST-CALL | CALL | LAST-CALL | ONLY-CALL |
              PAR-CHOICE | ENTER-UNIFY | RETURN
```

With the following syntax:

```
INIT-CALL :: ↑Instruction Nat (↑Parameter)*
FIRST-CALL :: ↑Instruction (↑Parameter)*
CALL :: ↑Instruction (↑Parameter)*
LAST-CALL :: ↑Instruction (↑Parameter)*
ONLY-CALL :: ↑Instruction (↑Parameter)*
PAR-CHOICE :: (↑Instruction)*
ENTER-UNIFY :: Nat (↑Parameter)*
RETURN :: nil
```

Both Parameters and instructions are stored in the static memory.

5.2. Tokens and continuation frames

The state of a process is represented by a token and a list of read-only continuation frames stored in the dynamic memory. Here follows the definition of a token (Token) and a continuation frame (Cont-Frame) which were schematically introduced in Section 2.

```
Token :: ↑Instruction Context-Name ↑Environment  ↑Cont-Frame (↑Parameter)*
Cont-Frame :: ↑Instruction Context-Name ↑Cont-Fram
```

↑Environment refers to the process's environment directory, and Context-Name is used to lookup the designated context in this directory. (↑Parameter)* is the list of parameters in a call instruction.

5.3. Execution cycle

In each cycle a processor fetches a token from the token pool, fetches the referred intruction from the static storage, and finally decodes and executes the instruction. A result of an instruction is none, one or more tokens. No more tokens means that this branch of the search tree has terminated, either with success or with failure. One token means that the current branch is continued. More tokens means that a nondeterministic point has been encountered and a fork into new branches has occurred.

The interpretation cycle of a processor is shown below. A number of auxiliary functions, or procedures, are used there. These functions are divided mainly into two groups: (1) functions operating on the token pool, and (2) functions operating on the dynamic storage for managing environments and variable-contexts.

Token management

```
FetchToken : => Token
    Delivers a token form the token pool to the calling processor.
SendToken : Token =>
    Sends a token to the token pool.
```

Binding environment management

The following operations are described in detail in [CiHa83A,CiHa83B]:

DuplicateEnv : ↑Environment => ↑Environment

Creates a logical copy of the input environment and returns a reference to the newly created environment.

ReleaseEnv : ↑Environment =>

Reclaims the storage of the input environment and all its contexts that are no longer accessible.

SendSolution : ↑Environment =>

SendSolution(e) extracts the bindings of the variables in the first context in e and then performs a ReleaseEnv operation.

NewContext : ↑Environment Nat => Context-Name

NewContext(e,n) creates a new context of n variables in e and returns its name.

Unify: (↑parameter)* Context-Name (↑parameter)* Context-Name ↑Environment => BOOL

Unify executes of the unification algorithm, it accesses and assigns values of variables.

One more auxiliary function is

NextInstr : ↑Instruction => ↑Instruction

NextInstr(i) returns a reference to the instruction following i↑.

Here follows the processor cycle.

```
Instruction-Processor processor() ≜
    (cycle
        (def mk-Token(i,c,e,cf,ps): FetchToken();
         cases i↑ :

            mk-FIRST-CALL(i1,ps1) →
                (def cf1 : New(mk-Cont-Frame(NextInstr(i),c,cf));
                 SendToken(mk-Token(i1,c,e,cf1,ps1))
                ),
            mk-CALL(i1,ps1) →
                (def mk-Cont-Frame( , ,cf1) : cf↑;
                 def cf2 : New(mk-Cont-Frame(NextInstr(i),c,cf1));
                 SendToken(mk-Token(i1,c,e,cf2,ps1))
                ),
            mk-LAST-CALL(i1,ps1) →
                (def mk-ContFrame( , ,cf1) : cf↑;
                 SendToken(mk-Token(i1,c,e,cf1,ps1))
                ),
            mk-ONLY-CALL(i1,ps1) →
                SendToken(mk-Token(i1,c,e,cf,ps1)),
            mk-PAR-CHOICE(is) →
                (for i=1 to len is do
                   SendToken(mk-Token(is[i],c,DuplicateEnv(e),cf,ps));
                 ReleaseEnv(e)
                ),
            mk-ENTER-UNIFY(n,ps1) →
                (def c1: NewContext(e,n);
                 if Unify(ps,c,ps1,c1,e) then
                    SendToken(mk-Token(NextInstr(i),c1,e,cf,nil))
                 else    !Failure
                    ReleaseEnv(e)
                ),
            mk-RETURN() →
                (if cf=nil then     !Success
                    SendSolution(e)
                 else
                    (def mk-Cont-Frame(i1,c1,cf1) : cf↑;
                     SendToken(mk-Token(i1,c1,e,cf,nil)))
                    )
                )
        )
    )
```

6. Discussion

During an Or-parallel execution the number of processes, as prescribed by their tokens, usually exceeds the number of available processors. The problem of storing the state information during the traversal of a search tree is not special for our parallel machine. In a sequential machine, information about not yet executed alternatives must be saved. Breadth-first traversal of the search tree usually leads to a combinatorial explosion of the space requirement. Therefore, practical logic programming systems control the traversal of the search tree usually by using a depth-first traversal strategy combined with a mechanism for pruning some branches of the search tree. Such a mechanism takes the form of a rudimenary Cut (Slash) operator as in Prolog, or intelligent backtracking or both.
Similarly, any feasible parallel machine should incorporate mechanisms for
(1) controlling the traversal of the search tree, and
(2) pruning some branches of the search tree.

The first issue can be reduced to that of adopting a proper policy for scheduling the tokens on the available processors. For instance, if the token pool has the form of a LIFO queue, and each processor keeps always one of the tokens it produces and sends the other tokens to the queue, our parallel machine would then work as a 'broad' depth-first machine, investigating in parallel a number of branches that is equal to the number of processors. That is to say, having n processors we get approximately n Prolog machines working in papallel. The centralised access of such a token pool would presumably create a bottle-neck in the system and have to be approximated by partitioning the token pool on the processors. This issue will be treated in a forthcoming paper.

The second issue requires either an extension to the source language, or a seperate control language. Very often one would like to get exactly one solution for a subgoal. This happens, for example, when the relation defined is in fact a function for certain patterns of arguments, or when it is a test predicate with all arguments instantiated, or for several other reasons. Once a solution for a goal is found, the other branches in the search tree having the same goal can be pruned. Translating this into our machine means that a mechanism for aborting certain tokens should be available. Such a mechanism does not require any process hierachy nor message passing between processes. The extended machine incorporating a mechanism solving this, and other problems, is described in another paper [CiHa83C].

Our abstract machine can, be classified as an unconventional control-driven machine [Tr]. The execution sequence is decided by the flow of control in tokens. It is interesting to compare it with a data-driven abstract machine proposed by Umeyama and Tamura[UmTa]. In their proposal a program is represented by a dataflow graph. Tokens carry instantiated goals, subtitution sets or both. Tokens are dynamically tagged to distinguish different invocations of the same clause. Tokens with the same tag must be matched during the execution. We consider the dataflow principle is an unnecessary complication

in an Or-parallel machine, because of the overheads in both creating unique tags and matching tokens with the same tag. Dataflow is not needed because the control flow of the programs can be determined at compile time regardless of the arrival of data. The dataflow principle might be interesting when some form of and-parallelism is considered.

References

[BjJo] D Bjørner, C B Jones (eds), The VDM: The Meta Language, Lecture Notes in Computer Science 61, Springer-Verlag 1980

[CiHa83A] A Ciepielewski, S Haridi, Storage Models for Or-parallel execution of Logic Programs, TRITA-CS-8301, Royal Institute of Technology, Stockholm 83

[CiHa83B] A Ciepielewski, S Haridi, A Formal Model for Or-parallel execution of Logic Programs, to appear in IFIP 83, North Holland P. C., Mason (ed)

[CiHa83C] A Ciepielewski, S Haridi, Control of Activities in an Or-parallel Token Machine, in this proceedings.

[CoKi] J S Conery, D F Kibler, Parallel Interpretation of Logic Programs, ACM Symposium on Functional Programming Languages and Computer Architecture, October 1981

[DaRe] J Darlington, M Reeve, ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages. Proceeding of ACM Conference on Functional Programming Languages and Computer Architecture, 1981.

[EKM] N Eisinger, B Kasit, J Minker, Logic Programming a Parallel Approach, First Logic Programming Conf., Marseille 1982

[FNM] K Furukawa, K Nitta, Y Matsumoto, Prolog Interpreter Based on Concurrent Programming, in proceedings of the First International Logic Programming Conference, Marseille 82

[HaSa] S Haridi, D Sahlin, An Abstract Machine for LPL0 TRITA-CS-8302, Royal Institute of Technology, Stockholm 83

[Po] G H Pollard, Parallel Execution of Horn Clause Programs, PhD Thesis, Imperial College of Science and Technology, University of London, 1981

[Tr] P C Treleaven, D R Brownbridge, R P Hopkins, Data-Driven and Demand-Driven Computer Architecture, ACM Computing Surveys, vol 14, No 1, March 1982

[RoSi] J A Robinson, ε E Siebert, LOGLISP: Motivation, Design and Implementation, in Logic Programming edited by K L Clark and S-Å Tärnlund, Academic Press 82

[UmTa] S Umeyama, K Tamura, Parallel Execution of Logic Programs, in proceedings of the 10 Symp. on Computer Architecture, Stockholm 83