PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

Execution of Bagof on the Or-parallel Token Machine

Andrzej Ciepielewski and Seif Haridi
Dept. of Telecommunication and Computer Systems
Royal Institute of Technology
10044 Stockholm, Sweden

ABSTRACT

In order to achieve efficient parallel execution of logic programs new computer architectures and new ways of controlling execution of programs must be deviced. In this paper we discuss one aspect of Or-parallel execution. We describe mechanisms necessary for parallel execution of the bagof construction and show how their implementation is supported on a parallel token machine consisting of a limited number of processors, a token pool and a storage. In the context of Or-parallel execution the bagof construction is not only a way to collect alternative solutions of a relation in a list (bag). It is also a way to control parallelism, because the search tree of a program becomes smaller when the bagof is used. Besides, the bagof has been proposed as an interface between pure Or-parallelism and a form of And-parallelism. The main problems encountered during distributed implementation of the bagof are: control of termination of subcomputations looking for the alternative solutions to a program, and merging of the solutions into the environment of the computation waiting for the results. The decentralised mechanisms we propose here are also of interest outside the domain of pure Or-parallelism.

1. Introduction

In order to achieve efficient parallel execution of logic programs new computer architectures and new ways for controlling execution of programs must be deviced. In our research we have concentrated on problems of Or-parallel execution of logic programs, partly because they are easier than the problems of combined And-Or-parallelism, and partly because their solutions are part of the solutions to the more general problem.
We have defined a process model for Or-parallel execution and a storage model for managing multiple, simultaneous bindings produced during the execution [1,2]. We have also defined a mechanism for aborting unnecessary processes [2,3]. Finally we have designed a parallel token machine supporting Or-parallel execution, abortion mechanism, and the storage defined by the model [2,3,4].
In this paper we describe mechanisms necessary for parallel execution of the bagof construction, and show extensions to the token machine supporting implementations of the mechanisms.
The bagof is a construction for collecting solutions to a relations in a list. The construction can be invoked anywhere in a program. In the context of parallel execution the bagof has an important side-effect of controlling parallelism, because the search tree of a program becomes much smaller when bagof is used.
The bagof construction we use is analog to the ones described in [5,6,7].
The rest of this paper is organised as follows. First, to make the paper self contained, we give an overview of the token machine with the model of storage for maintaining multiple, simultaneous bindings of variables, and introduce shortly the semantics of the bagof. Afterwards, we describe the problems of a parallel implementation of bagof, and present the mechanisms necessary for solving them. Finally we show the extensions that must be done to the token machine in order to support implementation of the mechanisms, and show a small example.

2. Overview of the Or-Parallel Token Machine model

An Or-parallel computation can be visualised as an unlimited number of independent processes, one for each alternative nondeterministic branch in the search tree of a program, sharing a storage for binding environments and programs (see Figure 1).

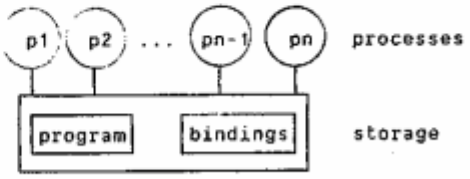


Figure 1. An or-parallel computation model.

In the Or-Parallel token machine model, the unlimited number of processes is mapped onto a finite number

of processors. On this conceptual level the machine, as depicted in Figure 2, consists of a token pool, a set of processors and a storage. The storage is divided into a static memory for programs and a 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.

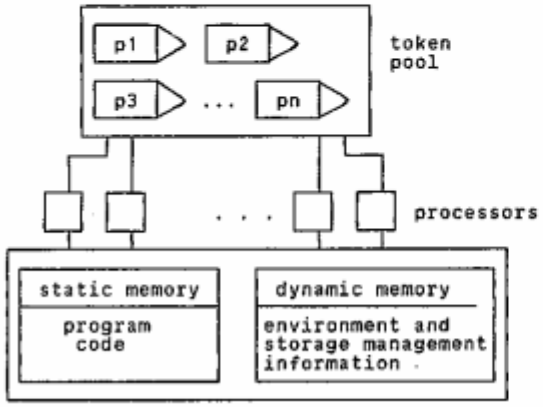


Figure 2. The Or-Parallel Token Machine model.

The state of a process consists of a list of goals and a binding environment. Such a state is represented in our machine by a token residing in the token-pool or in one of the processors, a binding environment residing in the dynamic memory, and by a, possibly empty, list of continuation frames also residing in the dynamic memory.

A token consists of the following fields:

1. Literal reference (L),
2. Context name (C),
3. Environment reference (E),
4. Continuation-Frame reference (CF)
5. Term list reference

A binding environment of a process consists of contexts for storing values of variables in literals. A new context is created each time a literal is invoked. A context name refers to a context in a given environment. During Or-parallel execution each variable may be bound to several values, still each process must have access to just one value, the one in its environment.

There are several methods for maintaining a separate address space for each binding environment. In this paper we present a simplified version of the storage model described in [1,2]. Other models are described in [8,9,10].

The storage for binding environments consists of two types of storage: directory storage and context storage. A directory, stored in the directory storage, consists of references to contexts stored in the context storage. Each process has a private directory. The binding environment of a process consists of all the contexts referred from its directory. Variables in the environment of a process are accessed and updated through the unique name, a triple: <environment reference, context name, variable name>, where environment reference is the address to the directory of a process, context name is the offset of an entry in the directory, variable name is the offset in the context addressed by the entry.

When a process creates two or more offsprings each gets a private directory. The new directories are created from the old one in the following way. Each context referred from the old directory is investigated. If it does not contain unbound variables - we say it is committed, then the reference to it is placed in all the new directories at the same offset as in the old one. If the context contains unbound variables - we say it is uncommitted, then a copy of the context is made for every new directory, and the reference to one copy is placed in each directory at the same offset as in the old one. By making copies of uncommitted contexts we ensure that alternative values will be given to variables in separate contexts belonging to different environments. At the same time we utilise the single assignment property of logic variables by allowing sharing of committed contexts.

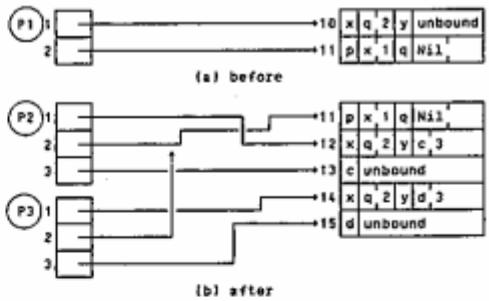


Figure 3. Snapshot of the storage before and after creation of two processes and duplication of the environment. At (b) context 11 is shared between the new environments, context 10 has been copied into contexts 12 and 14, where y has got different values.

An improved model in which investigation of all entries in the directory of a process is avoided, when new directories are created, is described in [1,2].

A continuation frame has the following fields:

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

Continuation frames are read-only data objects that are usually shared among several tokens.

The pair <L,C>, either in a token or a continuation frame, represents a goal; the L-field identifies a static literal and the C-field identifies the context, in a given environment E, containing the values of the

variables occurring in the literal L. 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 consists of the remaining goals. The pair <L,C> of a token represents the current goal, whereas its continuation frames represent the remaining goals. Below, when the machine instructions are outlined, the L-field in tokens and continuation frames will be a reference to an instruction.

A processor execution cycle proceeds as follows. The processor fetches a token from the token pool, fetches the referred instruction from the static storage, and finally decodes and executes the instruction. A result of an instruction is zero, 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.

Horn clauses are translated into an abstract machine code. Below we show only sequences of generated instructions; the machine representation of terms is given in [11].

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

Terms: t,q,r,s.
Literals: R,S.

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

A program consist of an initial call and relations.

An initial call having n terms containing m distinct variables as parameters:

$R(t_1,...,t_n)$

is translated into:

```
INIT-CALL m #R (#t1 #t2 ...#tn)
DISPLAY
```

Execution of the first instruction initialises a computation, The initial token transfers the control and the parameters to the relation R. The initial continuation frame points to the following DISPLAY instruction. This instruction will be executed by all computations which will complete R successfully.

An assertion having n terms containing m distinct variables as parameters:

R(t1,...,tn)

is translated into:

```
#R -> ENTER-UNIFY m (#t1 #t2 ... #tn)
RETURN
```

A token invoking a relation (assertion or implication) carries the context name and the parameters of the caller. Before the unification is executed ENTER-UNIFY creates a new context. When the code of an assertion is executed no new goals are created, after the unification has finished successfully the control is transferred to the caller. If an assertion has been called by an initial call the next instruction to be executed after RETURN is DISPLAY.

An implication having n terms containing m distinct variables as parameters:

```
R(t1,...,tn) <-
    S1(q1,...,qm1) &
    S2(r1,...,rm2) &
        :
    Sl(s1,...,sml)
```

is translated into:

```
#R -> ENTER-UNIFY m (#t1 ... #tn)
    FIRST-CALL #S1 (#q1 ... #qm1)
    CALL #S2 (#r1 ... #rm2)
        :
    LAST-CALL #Sl (#s1 ... #sml)
```

The instructions of an invoked implication are always executed in the same context. When a relation, invoked by a FIRST-CALL instruction or a CALL instruction, terminates, the next instruction to be executed is identified by the continuation frame created during that call. On the other hand a LAST-CALL instruction does not create any continuation frame, and therefore when a relation initiated by a LAST-CALL terminates, control is returned to the caller of the implication. This leads to tail recursion optimisation.

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

```
#R -> PAR-CHOICE (#C1 #C2 ... #CN)
#C1 -> DUPLICATE      #CN -> DUPLICATE
   Code for  ...      Code for
   clause C1          clause Cn
```

PAR-CHOICE results in a number of new subcomputations. Each will start by creating a copy of the current environment.

The following description of the instructions, is relative to the token being interpreted, so "Remove first continuation frame" actually means to remove the first continuation frame from the list associated with the interpreted token.

Another abbreviation we use is "transfer control and parameters to X", it means: create a token referring to X and its parameters and send it to the token pool.

(1) INIT-CALL m #R (#t1 #t2 ... #tn) :
Create the initial environment; create, in the new environment, a context for m variables; create a continuation frame, save address to the next instruction in it and link it first in the continuation frame list; transfer control and parameters to R;

(2) DISPLAY
Display values of the variables in the context corresponding to the initial call.

(3) ENTER-UNIFY m (#t1 #t2 ... #tn) :
Create a context for m variables in the current environment. Execute a unification step; the callers parameters are referred to by a secondary field in the interpreted token.

(4) RETURN :
Return control to the caller. A reference to the next instruction to be executed is stored in the first continuation frame.
(5) FIRST-CALL #S (#t1 ... #tn) :
Create a continuation frame referring to the next instruction 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; create a new continuation frame referring to next instruction and link it first in the continuation frame list; transfer control and parameters to S.
(7) LAST-CALL #S (#t1 ... #tn) :
Remove the first continuation frame; transfer control and parameters to S.
(4) PAR-CHOICE (#C1 #C2 ... #Cn) :
Create n tokens sharing environment of the interpreted token; The created tokens share also the continuation frame list of the interpreted token.
(9) DUPLICATE
Duplicate the environment of the interpreted token; Transfer control to the next instruction.
This instruction, as well as the other storage operations, is described in detail in [1,2].
The complete specification of the basic machine is presented in [2,4].

3. Semantics of Bagof

Invocation of a predicate with the help of Bagof will have the following format: bagof(b-variable, term, predicate). There are two constraints: b-variable must not occur in the term or the predicate following it, and all variables occurring in the term must also occur in the predicate following it. The above constraints are introduced to enforce the intended use of Bagof (see below).
Let us show an example of an implication containing a bagof call.

```
primefactors(lpf,x) <-
bagof(lpf,u,primedivisor(u,x)).
```

The initial call primefactors(lpf,10) will produce, assuming the function primedivisor is properly defined, the list lpf=<2,5>.
Bagof(s,t,P) is logically equivalent to:

$s=<u \mid \exists\, y_1,\ldots,y_k\ (u=t \wedge P)>$

where $y_1,\ldots,y_k$ are variables occurring only in P and t (local to bagof). The bagof can only be used to produce a binding for s, it cannot be used to test if some list satisfies a condition or to generate values for any non-local variables. To satisfy the last assumption, and still be able to have an efficient implementation, we require all non-local variables to be ground.
Bagof(s,t,P) invokes a relation with the same name as the name of P, and produces a list s, where each element is an instance of t with variables bound by an alternative solution of P. The invoked relation is executed in Or-parallel mode, as described earlier. The list of terms is produced only if execution of the invoked relation terminates.
Let us show two more examples.
A program

```
initial call: motherofchildren(Eve,ch).
relation: motherofchildren(m,ch) <- female(m) &
bagof(ch,q,childof(q,m)).
```

with properly defined relations female and childof, will produce the list,ch, of all children of Eve.
And a program

```
initial call: bagof(list-of-pairs,Pair(s,l),p(s,l)).
relation: p(s,l) <- student(s) & bagof(l,u,takes-
course(s,u)).
```

with properly defined relations student and takes-course, will produce a list of pairs: a student, all courses taken by the student.
The examples above are taken from [5].
Alternative definitions of bagof are possible [6], but in the context of Or-parallel execution the one above seems to be the only reasonable one, since invocation of a relation finds all solutions to it (all sets of bindings making it true).

4. Implementation Problems

When a relation with several statements is invoked a set of independent computations is started, each working on its own binding environment. Each computation branches into several subcomputations etc, until the successful leaves (empty goal list) of the search tree are reached. A successful leave of a tree contains a final binding environment. To be used, each result (binding of a set of variables) must be extracted from a final environment.
When a relation p is invoked by bagof(s,t,p) in an environment E0, an ordinary Or-parallel computation is started for p, except that its initial environment is a copy of E0, i.e. contains all bindings created prior to the invocation. When the execution of the relation p is ready, the results from the final environments must be transferred to the environment E0, and the list s constructed, consisting of the instances of the term t with different bindings. We will illustrate the problem by a schematic example.
In the schematic examples below, we will use a notation similar to the one used in the preceeding sections but extended with notation for the bagof construct and bagof goals.

```
Variables: x,y,z
Terms: t,q,r,s
Literals: P,Q,R,S
Bagof construct: bagof(x,t,P)
Ordinary goals: <P>,<R>,<S>
Bagof goal: <bagof(P)><&>
```

Notice that the goal corresponding to the bagof construct consists of two goals: a proper goal for the literal invoked by bagof, and a "&" (Collect-) goal, which will be explained shortly.

Consider the following schematic program P. The program consists of the initial call P and the following relations:

(1) P <- Q & bagof(x,t,R) & S
(2) Q <- Q1
(3) Q <- Q2
(4) R <- R1
(5) R <- R2

S is an assertion

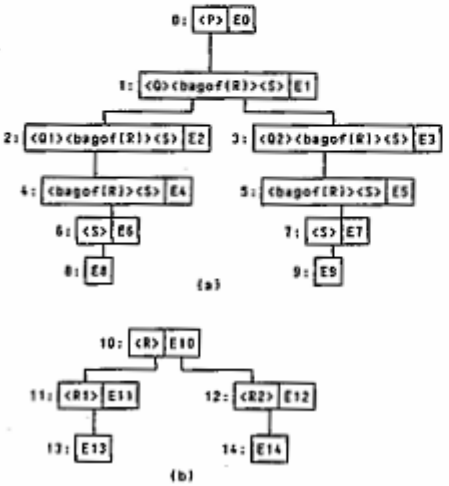


Figure 4. Search trees for the program P. (a) shows the search tree for P without the search subtree for an invocation of bagof(R) ("main" tree), and (b) the search tree for an invocation of bagof(R) ("auxiliary" tree). A node in a search tree consists of a list of goals and an environment reference. Variables in bagof are not shown. Collect goals are ignored in this figure.

Consider Figures 4a and 4b. When the goal <bagof(R)> in e.g. node 4 is executed, the traversal of the branch in the main tree containing this node is suspended, until the auxiliary search tree for the goal <R> is traversed. Traversal of the auxiliary search tree is started. The initial binding environment in the root of the search tree for R (node 10), is a copy of the environment E4. When all the leaves of the auxiliary tree are reached, construction of the suspended branch can continue. The environment E6 is constructed from E4, and the results extracted from E13 and E14. The goals in the branch beginning with node 6 can use all the results created during the execution of bagof(<R>) and gathered in the list x.

There are two problems to be solved: synchronisation of the final subcomputations of a bagof goal, and merging of the results from the environments of the final subcomputations into the environment of the suspended computation.

Let us first consider the synchronisation problem.

5. Synchronisation Mechanism

When a leaf in an auxiliary search tree is reached, it must be known if all other leaves in the tree have been reached (termination problem), and which goal is to be executed next.

To solve the termination problem, we introduce a collect goal and a counter. The collect goal is part of a bagof goal, and the counter is associated with the suspended node of the main tree. When a leaf in a auxiliary tree is reached, the next goal to be invoked is the collect goal following the bagof proper goal, which had invoked traversal of this auxiliary tree. Conceptually, for every subcomputation in the auxiliary tree, the list of remaining goals consists of the goal list in the current node and the goal list in the suspended node in the main tree. When a successful leaf in an auxiliary tree is reached, then the next goal to be solved, is the first goal on the list in the suspended node. When a goal in a node of an auxiliary tree fails, all the remaining goals in this node must be skipped, and the first goal in the suspended node should be invoked. To achieve this, we introduce references from nodes in an auxiliary tree to the goal to be taken in the suspended node. The problem arises because even failures must be counted in order to know when the execution of a bagof goal terminates.

The synchronisation mechanism works as follows. The counter is initialised to 1 when the proper goal of a bagof is invoked, and is increased by n-1 when n descendants of a node in an auxiliary tree are created. The counter is decreased each time when the collect goal of the bagof goal is invoked. The collect goal is executed every time a final leaf in the auxiliary tree is reached, even in the case of a failure. The construction of the main tree continues (the goal following the collect goal is invoked) when the counter reaches zero.

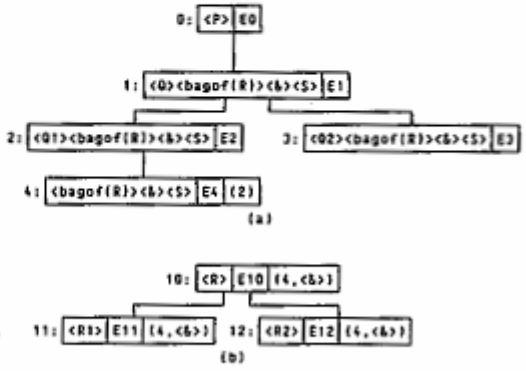


Figure 5. Initial levels of the trees from Figure 4 augmented with the control information. (4,<&>) is a reference to the collect goal in the node 4, (2) is the value of the counter.

Consider Figure 5. The bagof goal in node 4 was executed and the traversal of the auxiliary tree was started. The counter associated with node 4 is 2 and there are two leaves in the auxiliary tree. The continuation point after the construction of the auxiliary tree is the goal <&> in node 4.
In parallel systems it is not good to have centralised resources, like the counter we have just defined, accessed by many computations, because access to such resources creates potential bottle-neck. Having a common counter would lead to memory contention.
Instead of having a one counter we propose a tree of counters. The shape of a counter tree corresponds to the shape of the associated auxiliary search tree, that is, each node of a counter tree corresponds to a node with more than one descendant in the associated auxiliary tree. A node in the auxiliary tree has a reference to the corresponding node in the counter tree.
A counter tree is managed as follows.
- The root of a counter tree is created when the first node in the auxiliary tree having more then one descendent is reached. The counter in the root is initialised to the number of descendants of the node in the auxiliary tree. All descendant nodes will refer to the created counter tree node.
- A descendant to a node in the counter tree is created when a node in the auxiliary tree, referring to this counter node and having more than one descendant, is reached. The counter in the new node is initialised to the number of descendants of the associated node in the auxiliary tree. All descendant nodes of the node in the auxiliary tree will refer to the created counter tree node. If a node in an auxiliary tree has just one descendant, the descendant inherits parent's reference to a node in the counter tree.
- The counter in a counter tree node is decreased when one of the final leaves referring to it is reached, and the following collect goal is invoked. When the counter in a node reaches zero, the counter in its ancestor node is decreased recursively. When the counter in the root reaches zero, i.e. when all the leaves of an auxiliary tree are reached, the construction of the suspended branch in a main tree can continue.
Consider Figure 5 again. The counter tree for this example consists of just one node, and it is thus reduced to a single counter.
Let us show a more complex example.

initial call: T
relations: T <- bagof(x,t,P) & Q
P <- R & S
R <- R1
R <- R2
S <- S1
S <- S2
S <- S3
Q,R1,R2,S1,S2,S3 are assertions.

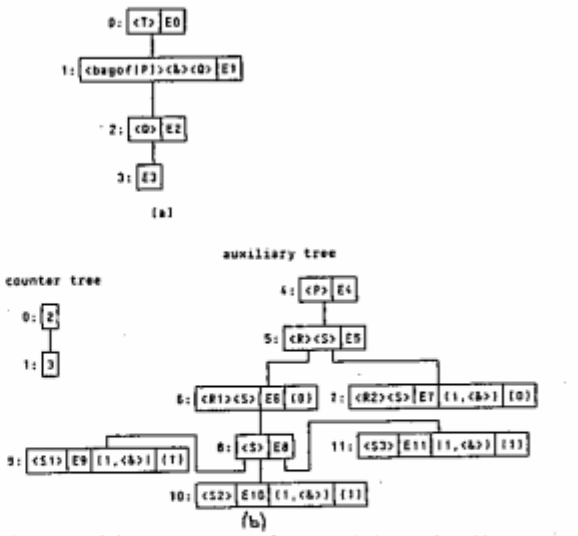


Figure 6. (a) shows the main search tree for the program T. (b) shows part of the auxiliary tree and the associated counter tree. (1,<&>) is a reference to the collect goal in node 1 of the main tree, (n) is a reference to node n in the counter tree. The continuation information is shown only in the leaves of the auxiliary tree.

Consider Figure 6. Node 0 in the counter tree was created when the goal <R> in node 5 was invoked. Node 1 in the counter tree was created when the goal <S> in node 8 was invoked. Say, <R2> in node 7 is invoked and fails. Then the counter in node 0 is decreased, and when goals <S1>, <S2>, and <S3> get ready in any order, the counter in node 1 is decreased and when it reaches zero the counter in node 0 is decreased, and also becomes zero.
Let us now consider the problem of merging the results from the environments of final subcomputations into the environment of a suspended computation.

6. Merge Mechanism

When a goal of the form <bagof(s,t,P)> is executed, a list s is constructed, consisting of instances of term t with the variables bindings provided by the alternative solutions to the relation P.
Consider Figure 7. s is bound to a data constructor List with the variable bindings in the context with name 2 and address 10. Components of List are t(y,z,w), and the auxiliary variable x. The bindings of y and z are the solutions to P, and the binding of x links two solutions to P. Context 10 contains values of y and z provided by the first solution, the value of s, and the value of x referring to the second solution. Context 40 contains values of y and z provided by the second solution, and the value of x, being in this case Nil. Contexts 30 and 50 contain the rest of the binding trees of x and y provided by solutions 1 and 2 respectively.

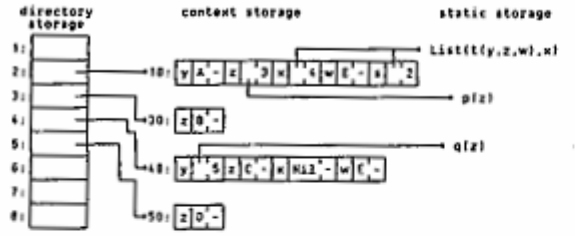


Figure 7. The storage representation of a possible solution to bagof(s,t(y,z,w),P(y,z)): s=List(t(A,p(B),E),List(t(q(D),C,E),Nil)). We have assumed that P(y,z) have two solutions, and that w is a non-local variable. The offsets in the directory and the addresses in the context storage are chosen arbitrarily.

When a bagof computation is started it gets a private copy of the current binding environment, and the reference to the current environment is saved. Each Or-parallel subcomputation executes in its own environment. When a subcomputation successfully terminates, the results provided by it, must be copied from its environment to the saved one. A result is the binding for variables in the term t, in a bagof call.

The binding of a variable consists of a tree of values (provided there are no circular bindings). Each value is a pair <Term, ContextName>, where ContextName is a reference to an entry in an environment directory. When a value is copied from a context in one environment to a context in another environment, the Term field remains unchanged, but the ContextName field gets a new value, because the corresponding entry will get a different offset in the new directory, than in the old one.

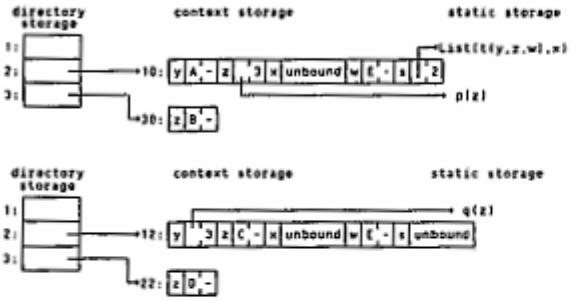


Figure 8. Content of the storage after both solutions shown in Figure 7 are ready, but only the first solution has been copied. (a) shows the saved environment with the incomplete list of solutions. (b) shows the environment of the second solution.

Consider Figure 8. Bindings provided by the second solution, see (b), will be copied to the saved environment and linked to the first solution, see (a) . During the copying, the dynamic part (ContextName) of the value of y in context 22 is changed, the result is shown in Figure 7. Notice that only the relevant contexts in a result environment are copied to the saved environment, and that the variables have the same offsets before and after copying.

Notice also, that the requirement about the non-local variables being ground has been done to make efficient copying of values between environments possible.

Copying of solutions between the environments is done independently for each subcomputation, the cooperation is needed only when a solution is linked into the list of solutions. The cooperation is achieved by associating with the suspended computation a reference to the currently last element on the list of solutions. When a solution is copied the saved reference is replaced by the reference to the new solution, and the new solution is linked last in the list.

Consider Figure 7. When the first solution is ready, the context name 2 (referring context 10) is associated with the suspended computation. When the second solution is ready and copied, it is linked in the list by binding x in context 10 to the second solution. Was it not the last solution then the context name 5 (referring context 40) would be associated with the suspended computation.

7. Implementation of the mechanisms on the token machine

In order to support the bagof construction the basic machine must be extended to handle the information associated with the described mechanisms. Two new types of frames, collection frames and counter frames, and some new instructions are introduced, and two instructions are modified. Tokens are extended with three fields, Collection-Frame reference, Counter-Frame reference, and Type, and consist of the following fields:

1. Literal reference (L),
2. Context name (C),
3. Environment reference (E),
4. Continuation-Frame reference (CF)
5. Collection-Frame reference (CL)
6. Counter-Frame reference (CN)
7. Type
8. Term list reference (T)

Collection frame reference identifies a list of collection frames associated with the token. The head of this list corresponds to the most recently invoked bagof goal, since bagof calls can be arbitrary nested. Counter frame reference identifies a list of counter frames associated with the token. The head of this list corresponds to the most recent node of the auxiliary search tree having more than one descendant node. The type field indicates if a token represents a failed or a successful computation. Continuation frames are unchanged.

Collection frames have the following fields:

1. Literal reference (L)
2. Destination context name (DC)
3. Result context name (RC)
4. Context name (C),

5. Environment reference (E),
6. Continuation-Frame reference (CF)
7. Collection-Frame reference (CL)
8. Counter-Frame reference (CN)

A collection frame is created for each invocation of a bagof goal proper. The fields of a collection frame are used in the following way. Literal reference and Continuation-Frame reference are used to find the continuation point when a computation terminates. On a successful computation the same information may be found in the first continuation frame pointed by the token. Destination context name, Result context name, Context name and Environment reference are used to transfer bindings between environments, and to link the alternative solutions. Besides, Environment reference and Context name are used to build a new token when all computations invoked by the corresponding bagof call proper have terminated. A result context name identifies the context containing variables of the term t in a bagof goal, this name is the same for all solutions. The destination context name identifies the solution most recently appended to the list of solutions. The fields Collection-frame reference and Counter-Frame reference are necessary for nested invocations of the bagof construct. The nested bagof calls are implemented by the same technique as the nested calls in general, i.e. by linking frames, in this case collection frames.

Counter frames have the following fields:

1. Counter
2. Counter-Frame reference

A counter frame is created each time a goal in a node with more than one descendant is executed. The Counter field records the number of non terminated branches.

Both collection and counter frames are shared data objects which must not be duplicated.

The execution cycle of processors is unchanged. Horn clauses without the Bagof construct are translated and executed as before. An implication containing m unique variables and having a bagof call in its body, but not in the first or last call position, with other calls of any type:

```
R(t1,...,tn) <-
        :
    bagof(s,t,P(q1,...,qk)) &
        :
```

is translated into:

```
#R -> ENTER-UNIFY m (#t1 ...#tn)
        :
    BCALL #Pi(#q1 ... #qk)
    COLLECT m-1 #s #t #x
        :
```

where m is the number of distinct variables in the clause plus link variable x, where x is an auxiliary variable necessary for constructing a list of solutions.

A similar implication with a bagof call first in the body, and other calls of any type, is translated into:

```
#R -> ENTER-UNIFY m (#t1 ...#tn)
    FIRST-BCALL #Pi(#q1 ... #qk)
    COLLECT m-1 #s #t #x
        :
```

And finally, a similar implication with a bagof call last in the body, is translated into:

```
#R -> ENTER-UNIFY m (#t1 ...#tn)
        :
    BCALL #Pi(#q1 ... #qk)
    COLLECT m-1 #s #t #x
    IMPL-RETURN
```

The synchronisation and merge mechanisms are implemented as follows. When a bagof goal proper is invoked, the information about the suspended node in the main tree is saved in a collection frame, which contains also a reference to the instruction to be taken when a subcomputation fails and the information needed by the merge mechanism. The counter trees are built of counter frames. A collection frame is created when BCALL, FIRST-BCALL, or INIT-BCALL is executed. Its Destination context name field is updated when a COLLECT instruction is executed. A collection frame is released when the associated counter tree becomes empty. A counter frame is created, and its counter initialised, when a PAR-CHOICE instruction is executed. The counter is decreased when a COLLECT instruction is executed, and the frame released when the counter reaches zero.

Two instructions of the basic machine, ENTER-UNIFY and PAR-CHOICE, are modified. The new and the modified instructions are described below. As before, the description of an instruction is relative to the token being interpreted.

(3') ENTER-UNIFY m (#t1 #t2 ... #tn) :
Create a context for m variables in the current environment; execute a unification step; the callers context name and parameters are referred to in the interpreted token; if the unification succeeds, transfer control to the next instruction; if the unification fails and it is a computation working on a solution to a bagof goal, i.e. the token refers to a collection frame, transfer control to the instruction saved in the first collection frame.

(9') PAR-CHOICE (#C1 #C2 ... #Cn)
If it is a computation working on a solution to a bagof goal, create a counter frame with the counter equal to n and link it first in the list of counter frames; create n tokens sharing the binding environments and all the frame lists of the interpreted token.

The above named instructions BCALL, INIT-BCALL, and FIRST-BCALL are very similar, for this reason and for the lack of space we show just BCALL.

(10) BCALL #S (#t1 ... #tn) :
Remove the first continuation frame; create a continuation frame referring to the next instruction and link it first in the continuation frame list; duplicate the current environment; create a

collection frame and save in it: the current context name and the environment reference, the address of the following COLLECT instruction and the address of the current continuation frame, name of the context for variables in S, the reference to a node in the current counter tree; link the created collection frame first in the collection frame list; S will execute in the new environment.

(11) COLLECT I #d #t #x

If the type of the current token is not a failure: add a solution to the bag s, i.e. allocate in the environment saved in the first collection frame, a new context for all variables in the clause except for s, save its name in the first collection frame, copy the bindings of variables in the term t from the current environment to the saved environment, and link the new solution last in the list of solutions (the list is empty in case all subcomputations failed); independently of the type of the current token, decrement the counters recursively, starting from the first counter frame; if the resulting counter tree is empty: close the list of solutions, and restore the part of the token saved in the first collection frame (context name, environment reference, continuation frame reference, collection frame reference, counter frame reference); transfer control to the next instruction; the next instruction will execute in the restored environment.

(12) IMPL-RET:

Remove the first continuation frame; return control to the caller. A reference to the next instruction to be executed is stored in the continuation frame which has become first.

The complete specification of the extended machine is presented in [2].

8. An Example

Before discussing the results presented in this paper, we will show search trees and the code for one of the programs mentioned in Section 3.

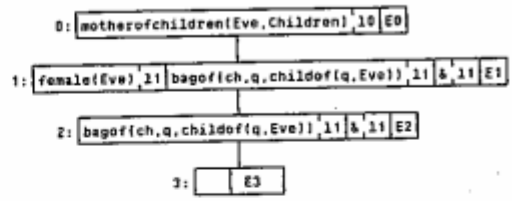
The program:

initial call: mother-of-children(Eve,children).
relations: mother-of-children(m,ch) <-
female(m) & bagof(ch,q,childof(q,m)).
female(Eve). female(Marie).
childof(Jack,Eve), childof(Jane,Eve),
childof(Daniel,Eve), childof(Mark,Marie)

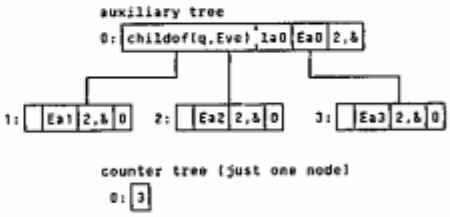
In the trees below, we replace the variables textually, where possible, by the terms to which the variables become bound, instead of showing the contents of environments.

In contrast to the schematic examples presented earlier, we show explicitly that a goal consists of a literal and a context name.

The main search tree:



The auxiliary search tree for the bagof goal in node 2 of the main tree, and the counter tree corresponding to a situation when nodes 1, 2 and 3 of the auxiliary tree has just been reached:



The code:

0: INIT-CALL 1 #motherofchildren #Eve #children
1: DISPLAY
#motherofchildren -> 2: ENTER-UNIFY 4 m ch
3: FIRST-CALL #female #m
4: BCALL #childof #q #m
5: COLLECT 3 #ch #q #x
6: IMPL-RETURN
#female -> 7: PAR-CHOICE 8 11
8: DUPLICATE
9: ENTER-UNIFY 0 #Eve
10: RET
11: DUPLICATE
12: ENTER-UNIFY 0 #Marie
13: RET
#childof -> 14: PAR- CHOICE 15 18 21 24
15: DUPLICATE
16: ENTER UNIFY 0 #Jack #Eve
17: RET
18: DUPLICATE
19: ENTER UNIFY 0 #Jane #Eve
20: RET
21: DUPLICATE
22: ENTER UNIFY 0 #Daniel #Eve
23: RET
24: DUPLICATE
25: ENTER UNIFY 0 #Mark #Marie
26: RET

With the information shown above the reader should be able to simulate execution of the program, in order to discover that the final result of the execution is children = <Jack,Jane,Daniel>. We restrain from showing execution snapshots, because of the lack of space.

9. Discussion

We expect that the results presented in this paper will be useful also outside the domain of pure Or-parallelism. It was proposed in [12], that the bagof construction is used as an interface between pure Or-parallel execution, and a form of And-parallel execution. Besides, any truly parallel implementation of And-parallelism must deal with non determinism with the help of some form of Or-parallelism, so our results will be relevant also there.

References

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

[2] A.Ciepielewski, Towards a Computer Architecture for Or-parallel Execution of Logic Programs, PhD Thesis, Department of Computer Systems, Royal Institute of Technology, TRITA- CS-8401, May 1984

[3] A.Ciepielewski, S.Haridi, Control of Activities in an Or-parallel Token Machine, in proceedings of IEEE Symposium on Logic Programming, Atlantic City, Feb 1984

[4] S.Haridi,A.Ciepielewski, An Or-parallel Token Machine, Logic Programming Workshop 83, Algarve/Portugal, also TRITA-CS-8303, Royal Institute of Technology, Stockholm 1983

[5] K.L.Clark, F.G.McCabe, S.Gregory, IC-Prolog Language Features, in Logic Programming, Clark and Tarnlund (eds), Academic Press, 1982

[6] L.M.Pereira, F.Pereira, L.Byrd, D.Warren, Users Guide to DECSYSTEM-10 Prolog

[7] K.Kahn, A Primitive for Control of Logic Programs, in IEEE Symposium on Logic Programming, Atlantic City, February 1984

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

[9] D.S.Warren, Efficient Prolog Memory Management for Flexible Control Strategies, in IEEE Symposium on Logic Programming, Atlantic City, 1984

[10] G.Lindstrom, Or-parallelism on Applicative Architectures, in proceedings of the 2nd International Logic Programming Conference, Uppsala, July 1984

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

[12] K.L.Clark, S.Gregory, PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, Imperial College, London, Department of Computing