Closed Environments: Partitioned Memory Representation for Parallel Logic Programs

John S. Conery

CIS-TR-86-02a
February, 1986
Revised May, 1987

Abstract

A method known as *closed environments* can be used to represent variable bindings for OR-parallel logic programs without relying on a shared memory or common address space. The representation allows structures to be shared when environments are in the same memory, and avoids the problem of common unbound ancestor variables. Two systems based on closed environments are briefly described.

To appear in
Proceedings of the 1987 Symposium on Logic Programming
September 1–4, 1987, San Francisco, CA

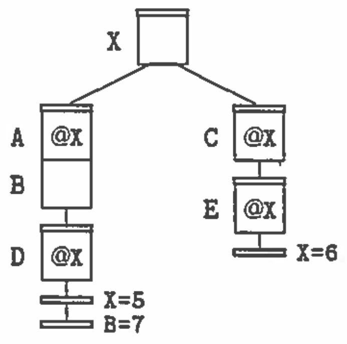
Department of Computer and Information Science
University of Oregon

1 Introduction

Implementation techniques for Prolog have advanced tremendously since the first systems. It is natural to explore the possibility of using the same basic techniques in parallel systems, in order to take advantage of years of experience. There is one major problem that must be solved, however, when Prolog style binding environments are used in OR-parallel systems. In OR-parallel execution, a process spawns parallel descendant processes when it reaches a point where there is more than one clause that can be used to solve a goal. At the implementation level, it is desirable to have new sibling processes share the binding environments created by their common parent process. The problem, well documented by now, is that this can lead to conflicting bindings if sibling processes are allowed to bind variables created by their parent. Each sibling needs a virtual copy of the parent's environment; it is not necessary to make an actual copy, but at a minimum every process needs its own copy of the unbound ancestor variables.

A simple example that shows the source of conflicting bindings is in Figure 1, which has a tree of stack frames. An empty slot in a frame represents an unbound variable. When two unbound variables are unified, one is bound to a reference to the other; the notation `@X` in the figure means the slot has been bound to a reference to the variable `X`. There are two parallel processes in this example, one for each leaf of the tree; the binding environment for a process consists of the set of stack frames from the root of the tree to the leaf. The frames with no variables are the frames for the assertions, which in this program have no variables. The labels next to these frames show which variables are bound by the corresponding unifications. In each case, an ancestor variable is bound, and in two cases the binding is made to a shared ancestor variable.

Ciepielewski and Haridi were the first to address the shared variable problem, devising a scheme that allows parallel processes to share stack frames as long as there are no unbound variables in the frames [2]. A recent paper by Ciepielewski and Hausman describes different techniques for implementing this method and gives some performance results [3]. Competing representations have been proposed by Borgwardt [1], Lindstrom [9], D. S. Warren [16], Yasuhara and Nitadori [18], and others. Crammond implemented a simple OR-parallel virtual machine to measure the relative performance of three of these techniques on a set of benchmarks [6]. What all of these schemes have in common is that at any point in the computation, the binding environment seen by any process is basically the same as that



```
Goal: ← p(X).
p(A) ← q(A) ∧ u(B).
p(C) ← r(C).
q(D) ← s(D).
r(E) ← t(E).
s(5).
t(6).
u(7).
```

Figure 1: Environment Stack in OR-Parallel System

seen by a Prolog program: variable bindings are kept in an array of local stack frames, unification of a goal and the head of a called clause often requires access to frames arbitrarily far back in the stack, and the unification of two unbound variables is represented by binding one variable to a reference to the other. The differences between the methods are that in order to find the value of an ancestor variable, different types of auxiliary structures (*e.g.* binding arrays or hash windows) are used to give each process its own copy of the variable.

The overhead in accessing the auxiliary structures is one of the drawbacks of adapting the three-stack model for OR-parallel systems. Another drawback is the requirement for a single memory address space. Since the binding environment is one large data structure, and a process must be able to access any variable in its portion of the structure, the underlying system must have either a single shared memory, or maintain a single address space in physically disjoint memory modules.

Accesses to shared ancestor variables will lead to problems for two common processor-memory configurations. In a system where each processor has a local memory module, such as Cm* [15] or the Hypercube [13], references to nonlocal addresses are slower. When a variable is bound to a reference to a cell farther back on the stack, or one of the frames used in a unification is located far back in the stack, the address of the ancestor variable will often be a nonlocal reference. An alternative configuration for processors and memories is to connect every processor to every memory through a large switch, as in the NYU Ultracomputer [8]. There are potential problems here, also, since when parallel processes check the binding of their common ancestor, the address of the ancestor may become a "hot spot" of the type investigated by Pfister and Norton [11]. When one memory address is accessed very often, the entire system can be adversely affected.[1]

Both types of time penalties for references to ancestor variables – the implementation level penalty from accessing auxiliary structures, and the machine level penalty from nonlocal or hot-spot references – derive from the fact that a process must be able to access slots anywhere in its stack. There are two situations where unification must have access to the value of an ancestor variable, both illustrated in Figure 1. The first is in the solution of `s(D)`; since `D` is bound to a reference to `X`, the unification algorithm examines `X`, and when it finds it is unbound, it binds it to the constant 5. The second situation is in the solution of `u(B)`. In this case, one of the frames used in unification is an ancestor frame; in order to find the current value of `B`, we have to look further back in the stack.

Empirical evidence for a pattern of memory accesses that shows a nontrivial number of references to ancestor variables can be seen in plots produced by Ross and Ramamohanarao [12]. They plotted addresses of memory references *vs.* time in a Prolog interpreter, in order to measure locality of reference. They found a cluster of locality at the base of the stack, where the shared ancestor variables would be located in an OR-parallel system. The frequency of read accesses to parent variables will be higher in OR-parallel systems than in the sequential Prolog interpreter measured by Ross and Ramamohanarao, since most of the techniques mentioned previously require references to many levels of intervening frames to ascertain the binding status of a variable.

The *closed environment* representation introduced in this paper is not an adaptation of the three-stack model. The binding environment seen by a process at any instant in time is restricted to one or two frames. Bindings are organized so that all the information needed for unification is present in the two input frames. The problems associated with ancestor variables are finessed, since it is not necessary to refer to a frame other than one of the input frames to find the value of a variable, and a variable in another frame cannot be bound during unification. The two situations where ancestor variables are accessed in three-stack models are handled by using a different representation for variable-variable unifications, and a protocol for updating the frames in an active process after a subgoal is solved.

[1]Most of the references will be reads, however, so a switch that combines read accesses may handle this problem.

One source of overhead in the closed environment model is that frames are copied extensively; at the start of the solution of each sibling subgoal, the parent's frame is copied. However, frames can be readily garbage collected, and the number of active frames per solution path will be smaller than for a three-stack implementation. Early simulation results from one of the implementations indicate the amount of memory used in the increased number of frames is comparable to the amount of space used for stack frames plus auxiliary structures of the generalized three-stack models. The advantages of the closed environment approach are that accesses to variable bindings during unification are uniformly fast, since there is no need to search for a binding in intermediate frames between a local frame and an ancestor frame. Also, the environments may be allocated in independent memory spaces, allowing implementation on a non-shared memory multiprocessor.

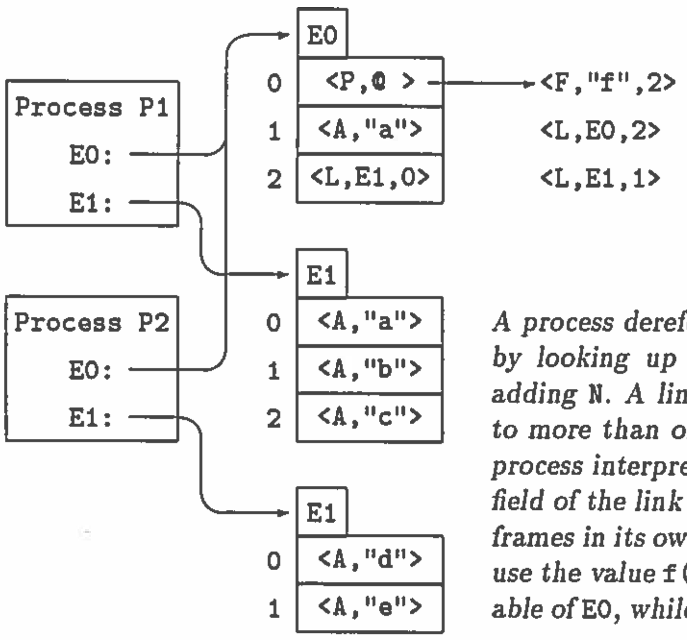
Closed environments were developed for OPAL (*O*regon *PA*rallel *L*ogic), an implementation of the AND/OR Process Model [10]. More recently, they have been incorporated into a parallel virtual machine, named OM [5], that executes compiled logic programs according to the AND/OR Process Model. A variant called loosely closed environments has been used for a pure OR-parallel system [14]. The use of closed environments in OPAL and OM is discussed below, after the description of term representation and unification using closed environments.

2 Definitions

In the closed environment representation, terms are represented as tagged *cells*. Atoms, variables, and pointers to other terms can be stored in single cells. n-ary complex terms are stored in prefix form in consecutive cells, where the first cell is the functor of the term, and the remainder are the n arguments.

A stack frame consists of an array of cells, containing a descriptor cell for the frame ID and size, plus one cell for every variable of the frame. When the system invokes a clause, a new frame is allocated for the variables of the clause. When a variable is bound to an atom, the atom is stored in the variable's slot in the frame. When a variable is bound to a complex term, an instance of the term is built in a global heap, and a pointer to the term is stored in the variable's slot.

The first difference between the three-stack model and closed environments is that when two unbound variables are unified, we use a relative



A process dereferences a link `<L,E,N>` *by looking up the address of* `E` *and adding* `N`. *A link can be dereferenced to more than one variable, since each process interprets the environment ID field of the link according to the set of frames in its own environment.* `P1` *will use the value* `f(a,b)` *for the first variable of* `E0`, *while* `P2` *would use* `f(d,e)`.

Figure 2: Dereferencing a Link

pointer known as a *link* to represent the binding. A link has two fields: a frame ID and an offset. The definition of a frame ID depends on the application using closed environments. IDs are not necessarily unique, which means a link may be dereferenced to many different variables. Dereferencing a link uses three pieces of information: the ID field from the link, the address of the frame with this ID (maintained by a process that owns the frame and does the dereferencing), and an offset from the start of the frame (Figure 2).

Another difference between the closed environment scheme and the three-stack model is that no variables are ever placed in the heap. Instead of putting a variable on the heap, we put a link that dereferences to the variable, and leave the variable in the stack. This is similar to the linking together of unbound variables in a structure copying representation, except the chain of references starts in the heap and not the stack. Note that this means an instance term is sharable. In the example in Figure 2, processes P1 and P2 share the same structure for the instance term, dereferencing the variables of the term according to their own environments.

Pointers to instance terms can be represented by the address of the term on the heap. This is done with the understanding that the frame and the heap are in the same address space. When the unification algorithm builds an instance term in a non-shared memory system, it will put it on the heap of the processor that currently owns the frame. Later, if and when a frame is relocated, its instance terms will have to be copied along with it if the new home for the frame is in a different address space.

We define a *closed environment* to be a set of frames E such that no pointers or links originating in E dereference to slots in frames that are not members of E. This implies that all slots in the frames of E are either unbound variables, atomic terms, links that dereference to other slots in E, or pointers to instance terms in which links, if any, dereference to slots in E. A closed environment containing just one frame is called a *closed frame*.

3 Environment Closing

The following is an algorithm that transforms a closed environment of two interrelated frames until one frame is, by itself, a closed environment.

PROCEDURE Close(RE,CE):

CE is the frame to transform into closed form; RE is the reference frame.

1. *Reverse the direction of all links pointing from* CE *to* RE:

   For each slot X of CE that contains a link to RE, dereference the link to a term T. If T is a nonvariable term, bind slot X to T; if T is an unbound variable of CE, bind slot X to a link to T; if T is an unbound variable of RE, set slot X to an unbound variable and bind T to a link to X.

2. *Extend* CE *with a new unbound variable for every unbound variable of* RE *in an instance term of* CE:

   For each instance term on the heap pointed to from a slot in CE, if an argument A of the term is a link to RE, dereference the link to a term T. If T is a nonvariable term, replace A with T and repeat this step on the arguments of T; if T is an unbound variable of CE, replace A with a link to T; if T is an unbound variable X of RE, create a new

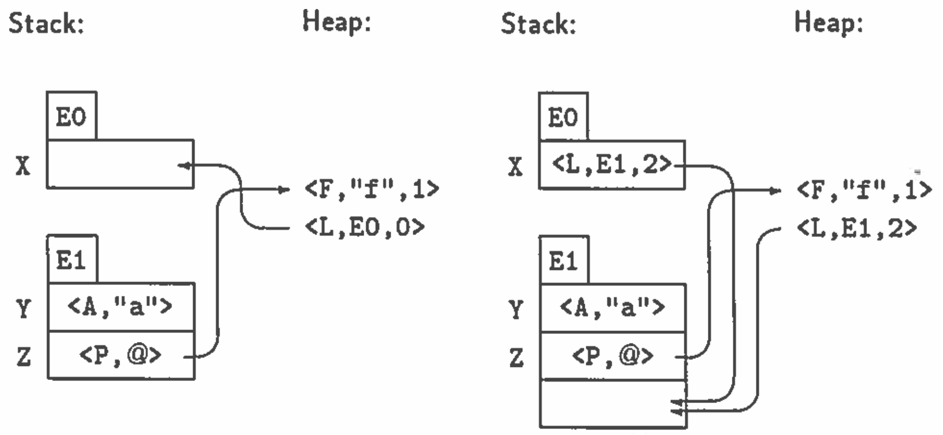


Figure 3: Example of Unification and Closing

variable *Z* in *CE*, bind slot *X* of *RE* to a link to *Z*, and replace *A* with a link to *Z*.

On input, the two arguments *CE* and *RE* are single frames, and together form a closed environment. After the algorithm is applied, all links between the frames will originate in *RE*, and *CE* will be a closed frame. We say this action *closes CE* with respect to the reference frame *RE*. Figure 3 shows an example of unification when both input frames initially contain only unbound variables, and are therefore both closed frames. After the unification, the two frames form a closed environment, but there is a link from *E1* to *E0*, so *E1* is not closed. The result of closing *E1* with respect to *E0* is shown on the right.

The significance of using closed environments is that if the pair of frames used in each unification make up a closed environment, the unification cannot bind slots in frames outside the environment. If, after each unification, the frame that will be used in the next unification is closed with respect to the other frame, the pair of frames used in the next round of unifications can also be a closed environment. As a result, computations on the frontier of the tree of parallel processes are based on closed environments, and variables in the interior (shared) frames are never modified. Furthermore, since all the necessary information is contained in the two frames, nonlocal memory references are avoided when the two frames are in the same memory. A new process and its initial environment can be relocated to a different processor, with its own local memory, and unifications in the new process will not have to refer back to the environment of the parent process.

It is possible to guarantee two closed frames for each unification by observing the following rules:

- Each unification involves two frames: one from the environment of the goal, called the *top* frame, and one for the clause being tried, called the *bottom* frame. If the two frames are each closed before unification, we can always transform them with the environment closing operation after unification so that one will be closed. (The terms "top" and "bottom" come from Ferguson diagrams, which will be explained in Section 4.1 and Figure 4.)
- The bottom frame is always a newly allocated frame, containing only unbound variables, so it is always closed. The environment of the initial goal statement contains only unbound variables, and is thus in closed form, so the first top frame is closed.
- After unifying a goal with the head of a unit clause, close the top frame with respect to the bottom frame. The closed form of the top frame can now be used to solve siblings of the goal.
- After each unification of a goal with the head of a clause that has one or more subgoals in its body, close the bottom frame with respect to the top frame. The bottom frame now becomes the closed top frame for calls to the procedures in the body of the clause.
- After solving the last goal in the body of a clause, close the calling goal's frame (a top frame) with respect to the bottom frame made for the clause. The closed form of the top frame can now be used to solve siblings of the original call.

Closing a frame serves two purposes. Closing the bottom frame with respect to the top after a unification ensures no subsequent unifications using the bottom or its descendants will bind slots in the top or any of its ancestors. Closing a top frame with respect to the bottom frame after a goal is successfully solved imports bindings from the environment of the solution back into the environment of the call, and also prepares the top frame for solution of siblings of the original call. This step replaces the back unification used in the AND/OR Process Model to update an environment with values computed by an independent process [4].[2]

An explanation of what the environment closing operation does, and why it is logically sound, is best made in terms of the procedural semantics of logic programming. A procedure call in a logic program is an inference based on the resolution rule [7]. The set of goals to be solved and the binding environment of the call represent one of the input clauses, and the called clause and its new stack frame represent the other input clause. The set of bindings made to slots of the input environments make up the substitution generated when unifying the goal with the head of the called clause. When the input environments are both closed frames, the substitution is limited to the two input frames; in other words, the variables on the left hand sides of the assignments in the substitution can be found in one or the other of the input frames.

The environment closing operation is a syntactic transformation on the unifying substitution. Each step of the operation may bind a variable, rename a variable, or introduce a new variable, in such a way that the meaning of the substitution is unchanged. Closing one of the frames does not alter the structure of the resolvent, as determined by the goals in the body or call; it simply transforms the binding stack, without altering the meaning of the bindings, until one of the frames is closed.

As an example of how substitutions are manipulated until one frame is closed, consider the following goal statement and clause:

```
← p(a,f(X)) ∧ q(X).
p(Y,Z) ← r(Y) ∧ s(Z).
```

The resolvent is

```
← r(a) ∧ s(f(X)) ∧ q(X).
```

where the unifying substitution is {`Y=a,Z=f(X)`}. Note that `Z`, a variable in the bottom frame, is bound to a term that contains a variable of the top frame, so the bottom frame is not closed after unification. The bottom

[2]The term "back unification" comes from Epilog, which performs a similar operation [17].

frame can be closed by modifying the substitution by adding a new unbound variable V, and binding X to V, giving: {X=V,Y=a,Z=f(V)}. In terms of the new substitution, the resolvent is:

```
← r(a) ∧ s(f(V)) ∧ q(V).
```

The new resolvent is identical to the original resolvent, except for the name of the variable, and, if V is added to the bottom frame, it will now be closed (this example was illustrated in Figure 3).

When implementing the environment closing algorithm, a decision must be made as to whether the input frames are modified in place or new arrays of cells are allocated to represent the modified frames. In the version implemented in OPAL, the closed form of *CE* is written into a newly allocated frame, so the original *CE* is not modified, but *RE* is modified in place. In this system, we often need to make a copy of *CE* before closing it, and the decision to write the closed form of *CE* into a new block of cells combines the copying and closing operations. In the OM implementation, unification and closing are done in environment registers, and a new frame is allocated for the modified *CE* only when it is extended or a copy of *CE* is required.

The algorithm presented earlier makes two passes over *CE*. A one-pass algorithm is possible, but in general the closed form of *CE* will have more slots if the one pass algorithm is used, since each time a link is reversed in pass one, we save an extension to *CE* in pass two. Another optimization is to have the unification algorithm bind the bottom frame in such a way that it is always closed, bypassing the need for a later close operation. We could incorporate a step from Lindstrom's algorithm, where the bottom frame is extended to include slots for each unbound variable of the top frame at the start of unification [9]. The unification instructions of the OM processor extend the bottom frame on demand, in cases where a later close operation would extend the frame. Even with this improvement, however, there are situations where a separate two-pass closing operation would give a more compact frame.

It is interesting to note that after closing the bottom frame with respect to the top frame, variable to variable references are pointing in the opposite direction than in most Prolog implementations based on the three-stack representation. In the latter, the convention is to bind new variables to pointers to old variables and to point stack variables at the heap. When pointers go in the opposite direction there is a danger of dangling references when the referent of the pointer is in a deallocated stack frame. However, in

OR-parallel systems, alternatives are not generated through backtracking, and the environment of a process grows monotonically, so the dangling reference problem will not arise. An advantage of being able to point from the ancestor to the descendant, in such a way that the pointer is dereferenced to different slots by different processes, is that a process does not have to locate the ancestor frame in order to find the value of a variable; the value is held locally, in the descendant frame. A disadvantage is that when there is more than one candidate clause to solve a goal, the top frame must be copied, to allow each unification to bind it in its own way. After this first level of unifications, however, the top frame is shared by all further descendants in that branch of the tree.

In summary, where the three-stack representation would show n references from local frames back to a common ancestor, the tree of closed environments would show a downward pointing link in the ancestor variable that could be dereferenced to n different slots. What makes this representation useful is that this form of dereferencing is never needed during unifications; all the information required for a unification step is present in the two frames involved in the unification. A downward pointing link does not have to be dereferenced until later, when the frame containing it is closed with respect to a descendant frame.

4 Applications

Three systems have been implemented using closed environments. One is a pure OR-parallel interpreter, based on the virtual machine described by Crammond [6]. The programs used by Crammond to compare hash windows, directory trees, and variable importation were executed by this interpreter. The closed environment model used less space than every other technique except Borgwardt's hash windows with small windows. Execution speed on the benchmarks is encouraging, but meaningful comparisons are not possible because of differences in the implementations and the host systems. This interpreter will be described in more detail in a companion paper [14]. The other two systems based on closed environments are outlined below.

4.1 OPAL

OPAL is an interpreter for the AND/OR Process Model, written in C and running under Unix 4.3. It is essentially the same interpreter as the OR-parallel, AND-sequential interpreter that was described in [4], but written in a lower level language in order to test implementation techniques. The interpreter is being ported to a small shared memory multiprocessor, with the goal of comparing the overhead of passing frames and instance variables in two implementations. In one case, we will use shared memory addressing modes to allow direct access to heap structures built by other processors. In the other case, we will force structures to be copied to the local memory of the destination processor when a frame containing a pointer to the structure is relocated.

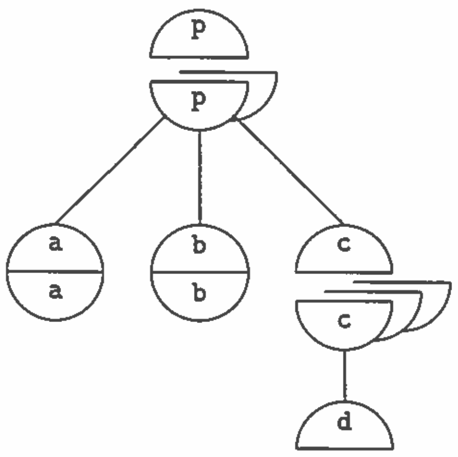
In OPAL, AND processes are used to solve goal statements, either the user's initial goal or the right side of a called clause. The state of an AND process consists of a frame to hold the bindings for the variables of the goal statement, plus some control information. When an AND process needs to solve a literal from its goal, it creates an OR process to manage the alternative solutions for the literal. When an OR process unifies the goal with the head of a nonunit clause, it starts an AND process for the body of the clause.

The environments are managed so that operations in an OR process do not modify its parent's environment. The variables of the AND process should be modified only when the AND process accepts a success message from the OR process. When there are many possible solutions, only one should be in effect at any time. When the OR process starts collecting solutions, it will act as a switch between AND processes, so that only one set of bindings from a descendant is used to determine the current state of the parent AND process.

The state of the system of processes can be represented graphically by a three-dimensional Ferguson diagram (Figure 4). An AND process corresponds to a lower half circle and the attached upper half circles, which represent the head and goal literals of a clause, respectively. An OR process tries to connect an upper half circle representing a procedure call with one of the lower half circles representing candidate solutions for the call. When an OR process has a result for its parent, the two half circles are joined. The snapshot in the top plane of the figure shows the progress of one of the candidates for solving p; this candidate has solutions for its first two goals and is working on the solution for the third.

Downward pointing links in the parent are effectively dereferenced to the current bottom frame presented by the descendant OR process. Other solutions to the OR process's goal are queued, awaiting a redo message from the parent before they can be selected as the current frame.

Frames are passed between processes by start and success messages. The



```
?- p.
p :- a, b, c.
p :- q, r.
a.
b.
c :- d.
c :- e.
c :- f.
```

Figure 4: Processes in OPAL

argument of the start message from an AND process to an OR process is a pointer to the current frame in the AND process. For each candidate clause, the OR process makes a copy of its parent frame to use as the top frame in the unification with the head of the candidate, and allocates a new frame for the variables of the candidate to use as the bottom frame.

For each successful unification with the head of a unit clause, the top frame is closed, and the becomes a result that will be passed back as an argument in a success message from the OR process to its parent AND process. For each successful unification with the head of a nonunit clause, the bottom frame is closed, and is sent in a start message to a new AND process for the body of the clause. After all unifications are done, the OR process is left with a list of top frames, one for each active descendant. When one of the descendants succeeds, the OR process needs to pass the bindings from that success back to its own parent. The success message from a descendant contains an updated copy of the bottom frame initially passed to it. The OR process returns values to its parent by closing the stored top frame with respect to this bottom frame, and sending the newly closed top frame back as the argument of the success message. When the parent is a sequential AND process, this frame now becomes its current environment, and is used to start the next OR process. Recall that in OPAL, closing the top frame automatically makes a copy of it and does not change the old top frame. The old top frame is saved and used with for the next result from the same descendant. The bottom frame from the descendant is discarded. Further details can be found in More's M.S. thesis [10].

4.2 OM

OM (for *O*pal *M*achine) is a virtual machine in the style of the Warren Abstract Machine, except control and unification instructions support the AND/OR Process Model. Instead of instructions to call procedures, build choice points, and maintain a trail, OM has instructions that start descendant processes and send messages between processes.

Like the WAM, OM performs unifications through a series of `get` and `put` instructions compiled specifically for each call and clause head. The `get` instructions of OM build closed bottom frames, so the code compiled for clause heads does not explicitly close the bottom frame after unification succeeds. The bottom frame is extended when a structure is being written to the heap, and part of the structure is a reference to an unbound top variable `V`. Instead of adding the link to `V` to the instance term, OM extends the bottom frame, binds `V` to a link to the new variable, and puts a link to the new variable in the instance term. In addition, there are instructions to close slots of a top frame as a means for incorporating results in success messages back into parent environments.

5 Other Methods for OR-Parallel Binding Environments

5.1 Directory Trees

Ciepielewski and Haridi were the first to tackle the problem of efficient runtime representations for parallel logic programs. Associated with each process is a *directory* containing pointers to stack frames that make up the binding environment for the process. When a new process is started, it initializes its directory by copying its parent's directory and adding a new frame for the called clause [2]. The directory entries in the new process point at the same frames its parent uses as long as those frames contain no unbound variables. If a frame has unbound slots, the new process makes its own copy of that frame and places a pointer to the copy in its directory. The scheme is made more efficient by copying on demand; that is, when a directory is initialized, the frame pointers are set to null, and a frame is not copied until a process needs to bind a slot in it. Finding the value of a variable is a matter of finding the frame for the variable in the directory. If the directory entry is a null pointer, the ancestor directories are searched until a directory is found where the frame pointer is not null. In the copy-on-read strategy, the directories in the search path are updated by giving them copies of the ancestor frame.

5.2 Hash Windows

In Borgwardt's *hash window* technique, an ancestor environment is not copied, but left as is. In situations where a unification would bind a variable in an ancestor frame, a new cell for the variable is allocated in a small local hash table associated with the current frame, and the local copy is bound [1]. Thus instances of shared variables are located in the local hash tables of the processes. Determining the value of an ancestor variable is a matter of checking the hash tables associated with the current frame and every frame in the tree on the path back to the variable's frame, since an intermediate ancestor may have bound the variable. This technique performed the best in Crammond's survey when small (four entry) hash tables were used.

5.3 Binding Arrays

A technique described by D. S. Warren is derived from the use of the trail stack to store backtracking information in sequential systems. Instead of pushing the address of an ancestor variable on the trail stack when the variable is bound, Warren's method will add a pointer to the variable and its binding to a *forward list* in the current frame [16]. The binding in the forward list corresponds to the process's copy of the ancestor variable, and is similar to extending the frame in the closed environment model. Finding the value of an ancestor variable is a matter of checking the forward list of every frame from the current frame back to the ancestor frame. To speed up this search, Warren proposed the use of *binding arrays* indexed by variable names to store pointers to the values of the variables in the forward lists. Each process would maintain its own binding array, which can be viewed as its own copy of the nonsharable information in the local stack.

5.4 Imported Variables

The notion of extending the current frame to contain slots for unbound variables of the parent was first proposed by Lindstrom. In this method, the called frame is extended to contain slots for the new variables, and an *import vector* is used to associate unbound variables of the parent with slots in the called frame [9]. Similarly, after the last goal of the body is solved, an *export vector* is used to map the still unbound variables to slots in a new copy of the parent frame. Finding the value of an ancestor variable involves following a pointer chain through the import and export vectors of intervening frames in the environment to see if the variable has been bound in one of these frames.

5.5 Kabu-Wake

The *kabu-wake*[3] method uses a different approach than the others discussed so far (Yasuhara and Nitadori [18]). Instead of arranging for a process to have its own instance of a shared variable and preventing the binding of the shared variable itself, this technique allows a process to bind an ancestor variable, just as a Prolog process would, saving the address of the variable in a trail stack. When a new process is started, the system picks an existing stack, copies it for the new process, and uses a backtrack point as the starting point for the new process. The first thing the new process does is to simulate backtracking and unbind its copies of the shared variables. An advantage of this method is that it allows one to use any specially designed Prolog processors as nodes in a multiprocessor architecture. There is no need to access auxiliary structures to find the value of an ancestor variable; the overhead of sharing these variables is postponed until the stack is copied and initialized for a new process.

5.6 Comparing Closed Environments with Other Methods

Both closed environments and Lindstrom's import vector method are based on extending a frame to contain slots for some unbound ancestor variables. In Lindstrom's method, the import vector is a copy of the parent's frame, where the unbound slots are used to associate variables with their copies in the new frame. In a closed environment system, the interpreter makes a

[3]A Japanese term for starting a new tree by splitting off a portion from a living tree that includes part of the root.

copy of the parent frame and lets the unification algorithm bind the copy directly. Another difference is that Lindstrom's algorithm extends the bottom frame before unification, allocating slots for each unbound variable of the parent. The environment closing algorithm extends the bottom frame after unification, usually resulting in fewer extensions to the frame, since some parent variables are bound in the unification. Also, a frame is extended in the closed environment method only when there is an instance term that contains a reference to an unbound ancestor variable. When two unbound variables are unified, the parent is bound to a link to the descendant variable, and the descendant frame is not extended.

The biggest difference between closed environments and imported variables is that the import and export vectors implement sharing in the basic three-stack model. With closed environments, there is no need for import and export vectors, since all references can be resolved within the two frames used in unification and there are no references to ancestor frames.

Ciepielewski and Hausman implemented the directory tree method using four different techniques for managing directories and copying ancestor frames [3]. In one of these implementations, to which they also give the name "hash windows," space is saved in a process' directory by copying only the ancestor variables that are referenced by a process, and not the entire frame that contains the variable. Variables are identifed by a tuple `<c,n>` where `c` is the frame index and `n` is the variable's index within the frame. Directories are managed as hash tables. To find the value of a variable, a process will hash on the ID of the variable, and search for it in its local table.

Ciepielewski and Hausman describe two variations in the implementation of their form of hash windows that increase the proportion of local references for this model. The first variation is called "local contexts." When a new process is created, and given a directory that is a copy of its parent's directory, the most recently allocated context is copied, on the assumption that the new process will most likely refer to this context during the next unification. Variables in the local context are accessed directly, not through the hash table that represents the remainder of the environment. The second variation is to insert an ancestor variable into the hash table when it is first referred to (copy on read), not when the directory is created.

If the ID field of a link in a closed environment system is the depth of the frame in the tree of frames of a pure OR-parallel system, the closed environment technique is similar to Ciepielewski and Hausman's hash windows with local contexts and copy on read. The biggest difference is that the first access to an ancestor variable with hash windows requires a search through intervening frames, to see if the variable has been bound since it was created; with closed environments, no search is required. With hash windows, a copy of the ancestor variable is inserted into the current directory (and the intervening directories) once it is located. After the variable is brought into the current directory, all accesses to it are local, so from this point on the pattern of memory references could be quite similar to the pattern in a closed environment system (if we ignore the differences between searching a small local hash table and accessing a local array). Ciepielewski and Hausman report that roughly 50% of the memory references in this implementation will be to local addresses when the model is implementated on a multiprocessor with local memories.

In the closed environment implementations of the AND/OR Process Model, the environment ID field of a link is a one-bit binary number, since all references are confined to either the top or bottom frame being modified by an OR process. The main advantage of this is that a process does not need to maintain a table of frames, such as the directory of the directory tree methods, with entries for frames of ancestor clauses. Each AND process keeps just one frame, and each OR process keeps a pair of frames for each clause with a head that unifies with the goal solved by the process. The ID of a parent process takes the place of a pointer to an ancestor frame or parent directory, and operations that bind ancestor variables turn into operations that update the ancestor frame when the process that owns the frame receives a success message.

6 Summary

Work to date on runtime representations for parallel logic programs has been oriented toward extending the basic three-stack model of Prolog for parallel execution. This model has evolved into a very efficient representation for variable bindings in sequential systems, and it is desirable to reap the benefits of this work when implementing parallel logic languages.

In a three-stack model, the binding environment of a process is a list of stack frames. In a parallel system based on the three-stack model, there is a tree of stack frames, where the environment of any one process is determined by a path from the root to a leaf, and frames near the root are shared by processes. Two aspects of this technique demand the use of a common memory space for every process. Unification might involve a frame arbitrarily far back in the stack, and the unification of two unbound variables is represented in a way that may cause later unification steps to follow a chain of references further back in the stack.

References to shared ancestor frames pose problems for both "dance hall" and "boudoir" configurations of global memory in multiprocessors. One of the goals for closed environments was to design a method that is more modular and does not require a global, shared memory to represent binding environments. This goal is partially met in implementations of pure OR-parallel systems. The binding environment of a process is still a monotonically growing list of frames, and it is most likely that processes will share some frames, which means the environment is still part of a global structure. However, the frames are organized so that unification does not have to access ancestor frames, and references to ancestor frames are postponed until the environment closing operation is applied after the body of a clause is solved, so the percentage of nonlocal accesses should decrease overall.

For systems based on the AND/OR Process Model, there are no nonlocal memory references during the execution of a process. Steps in both the unification and environment closing algorithms are done with accesses to local frames only. Information will be passed between memories only when a success or start message is sent from a process in one memory to a process in another memory, in which case the frame that is the argument of the message must be copied to the destination memory. Once the frame is located in the memory of the receiver, no accesses to the memory space of the sender of the message are required.

Acknowledgements

Many thanks to Paul Bloch, David Meyer, Nitin More, Don Pate, and Tsuyoshi Shinogi for patiently listening to early versions of this paper, and for discussions that sparked many of the ideas presented here. Thanks also to Peter Borgwardt and Andrzej Ciepielewski for comments on the final draft.

References

[1] Borgwardt, P. Parallel Prolog using stack segments on shared memory multiprocessors. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb. 6–9), 1984, pp. 2–11.

[2] Ciepielewski, A. and Haridi, S. *Formal Models for Or-Parallel Execution of Logic Programs.* CSALAB Working Paper 821121, Royal Institute of Technology, Stockholm, Sweden, 1982.

[3] Ciepielewski, A. and Hausman, B. Performance evaluation of a storage model for OR-parallel execution of logic programs. In *Proceedings of the 1986 Symposium on Logic Programming*, (Salt Lake City, UT, Sep. 22–25), 1986, pp. 246–257.

[4] Conery, J.S. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, Univ. of California, Irvine, 1983. (Computer and Information Science Tech. Rep. 204).

[5] Conery, J.S. and Meyer, D.M. *OM: A Virtual Processor for Parallel Logic Programs.* Tech. Rep. 87-01, University of Oregon, 1987.

[6] Crammond, J.A. A comparative study of unification algorithms for OR-parallel execution of logic languages. In *Proceedings of the 1985 International Conference on Parallel Processing*, (August 20–23), 1985, pp. 131–138.

[7] van Emden, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming language. *J. ACM 23*, 4 (Oct. 1976), 773–742.

[8] Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.M., Rudolph, L., and Snir, M. The NYU Ultracomputer – Designing an MIMD shared memory parallel computer. *IEEE Trans. Comput. C-32*, 2 (Feb. 1983), 175–189.

[9] Lindstrom, G. OR parallelism on applicative architectures. In *Proceedings of the Second International Logic Programming Conference*, (Uppsala, Sweden, July 2–6), 1984, pp. 159–170.

[10] More, N. *Implementing the AND/OR Process Model.* Master's thesis, Univ. of Oregon, 1986.

[11] Pfister, G.F. and Norton, V.A. "Hot spot" contention and combining in multistage interconnection networks. In *Proceedings of the 1985 International Conference on Parallel Processing*, (Aug.), IEEE, 1985, pp. 790–797.

[12] Ross, M. and Ramamohanarao, K. Paging strategy for Prolog based dynamic virtual memory. In *Proceedings of the 1986 Symposium on*

Logic Programming, (Salt Lake City, UT, Sep. 22–25), 1986, pp. 46–57.

[13] Seitz, C.L. The cosmic cube. *Commun. ACM 28*, 1 (Jan. 1985), 22–33.

[14] Shinogi, T. and Conery, J.S. *A Pure OR-Parallel Interpreter of a Logic Language Under Closed Environments.* Tech. Rep., University of Oregon, 1987. (in preparation).

[15] Swan, R.J., Fuller, S.H., and Siewiorek, D.P. Cm*: A modular, multimicroprocessor. In *Proceedings of AFIPS NCC*, 1977, pp. 637–644.

[16] Warren, D.S. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb. 6–9), 1984, pp. 198–202.

[17] Wise, M.J. A parallel Prolog: The construction of a data driven model. In *Conference Record of the Symposium on LISP and Functional Programming*, (Pittsburgh, PA, Aug. 15–18), ACM, 1982, pp. 55–66.

[18] Yasuhara, H. and Nitadori, K. ORBIT: A parallel computing model of Prolog. *New Generation Computing 2*, (1984), 277–288.