Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors *

John S. Conery

University of Oregon

Received May 1988; Revised September 1988

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 is based on a procedure that transforms stack frames after unification, taking into account problems with common unbound ancestors and shared instances of complex terms. Closed environments were developed for the AND/OR Process Model, but may be applicable to other OR-parallel models.

Key Words: OR-parallelism, logic programming, non-shared memory.

1 Introduction

With the availabilty of commercial multiprocessors, many Prolog implementers are now working on parallel systems. In general, implementation techniques are adaptations of successful methods developed over the years for sequential Prolog systems. A number of research projects have started with the familiar "three stack" binding environment method used in most Prologs, and developed parallel versions that build cactus stacks. The goal is that when a new process is spawned, it can share the binding environment created by its parent process, and begin making new bindings for itself along a new branch in the stack.

There are two major advantages of this approach to implementing parallel Prolog. First, those aspects of the system that are largely independent of the binding environment (such as built-in procedures and control structures) are readily used in the parallel system. Second, it is expected that many programs will have many more parallel tasks than the host system has processors, and it will be beneficial to switch to a sequential execution mode when the system is fully loaded; if parallel mode execution is an adaptation of an efficient sequential system, the transition from parallel mode to sequential mode and back should be relatively smooth.

* *International Journal of Parallel Programming*, Vol. 17, no. 2, April 1988, pp. 125–152.

A well-known problem arises when one tries to turn a Prolog stack into a parallel cactus stack. Solutions to this problem (surveyed at the end of the paper) introduce a significant amount of runtime overhead. Another potential drawback to the cactus stack approach is that the system requires a shared memory multiprocessor, or an implementation of a single address space in a distributed memory multiprocessor. As we will see, Prolog systems make a large number of nonlocal memory references to addresses in the environment stack, so this is a potentially serious problem for large systems.

This paper introduces a new technique for representing variable bindings in parallel logic programs. The technique, known as *closed environments*, was designed for programs of the AND/OR Process Model, which exploits AND parallelism in addition to OR parallelism, but it may be applicable to parallel Prolog systems as well. The technique has much in common with the "three stack" method, but the main difference is that an active process can restrict its attention to a small set of stack frames, which are likely to be local, and the method will work well in a distributed memory address space.

The central thesis of this paper is that OR-parallel logic programs can be executed efficiently in non-shared memory systems. The memory model introduced here trades off overhead required to set up and return values from parallel processes, in return for simpler mechanisms for accessing values during subsequent unifications. The trade-off is clearly worth the effort in implementations of the AND/OR model, since the overhead is a small addition to work that must be done to exploit AND parallelism. What is not clear is whether the trade-off is beneficial for pure OR-parallel systems, where there is currently no overhead associated with returning results of procedure calls. However, even in these systems, the ability to execute in smaller, independent address spaces will have a positive effect on garbage collection, and the control introduced to return bindings may also introduce a method for controlling combinatorial explosion. These, together with the scalability and efficiency of non-shared memory systems, may make the technique viable for pure OR-parallel systems as well.

The paper begins with an overview of OR parallelism and the problem introduced when Prolog binding environment stacks are turned into cactus stacks. Following that are a high level description of the closed environment technique and how it has been implemented in two different systems, one interpreted and one compiled. As is the case with unification in sequential Prolog, the steps that maintain environments in closed form can be optimized by a compiler, and the compiler based system is much more efficient. Finally, closed environments are compared with other methods, and areas of future work are described.

2 Binding Environments for OR Parallelism

A logic program is a collection of *clauses*, as shown in Figure **??**. Execution of a program begins with a *goal statement*. An execution step consists of removing a goal from the goal statement and matching it with the head of a clause in the program, adding goals from the body of the clause (if any) to the goal statement, and repeating, until all goals are removed. Prolog is a high level language based on this formalism. It has a prescribed control strategy that determines the order in which goals and clause heads are paired, and allows for built-in operations, such as arithmetic and I/O, to be performed by a host system.

The control strategy of Prolog describes a depth-first search of a tree of possible goal derivations. At any point, the system tries to solve the first goal in the current goal statement. For any goal, if there are a number of clauses that have matching heads, the system proceeds with the first clause, but remembers that there are alternatives in case that clause does not lead to a successful solution of the remaining goals. A system exploits *OR parallelism* if it can spawn parallel processes at such a choice point. Instead of storing the location of the alternatives on a stack, and visiting them on backtracking, an OR-parallel system starts new processes for each alternative clause. The term "OR-parallel" is used because if any new process reaches a solution, the original goal has been solved. The term "parallel Prolog" most commonly refers to a Prolog system that exploits OR parallelism: the same built-in goals are supported, and many of the extra-logical features (such as the cut operation, which prunes the set of alternatives as the program executes) can be implemented as well.

The AND/OR Process Model is not a parallel Prolog; rather, it is a more abstract framework, defined for pure logic programs. It also allows for AND parallelism, which is exploited when the system tries to simultaneously solve more than one goal in a clause, in addition to OR parallelism. While the closed environment technique was designed for programs of the AND/OR model, it may be applicable to parallel Prologs, as well, since it addresses the same issues that arise when Prolog binding environments are adapted for an OR-parallel system. The description in the following sections is independent of the execution model; in later sections, the overhead of closed environments is discussed in the context of the two different language models.

The usual representation for binding environments in Prolog is known as the "three stack" method. When a clause is invoked, a stack frame with slots for the variables of the clause is allocated from the *local* stack. A *global* stack, or heap, is used to hold instances of complex terms that cannot be represented in a single cell in the local stack. Finally, the *trail* holds pointers to variables that must be reset to unbound when backtracking occurs. If the system binds an existing variable during the unification of one clause, and later it turns out the clause does not lead to a success, the variable must be unbound so that alternative clauses have a chance to bind it when they are invoked. The addresses of these variables are stored in the trail.

During the unification of a procedure call with the head of a clause, the system must access two stack frames. One, newly allocated on the top of the stack, is for the variables of the called procedure. The other, which can be anywhere in the stack, holds the variables from the context of the goal statement. These frames will be called the *input frames* in the discussion of unification and environment closing algorithms.

The tree in Figure **??** is an illustration of binding environments. A stack frame is drawn as a collection of cells for variable bindings plus a small box to indicate control information.[1] Given the initial goal statement, there are two possible sequences of derivations that lead to results. There is a choice point in the call to `p(X)` since there are two clauses with heads that match this call. A Prolog system would store this choice point, build a stack frame for the first clause, and

[1] In this picture, and most others throughout the paper, we will show a slot in a stack frame for each variable of a clause. One of the improvements in Prolog technology made in the last few years has been the ability of a compiler to decide that some variables are "temporary" and do not need to have space allocated for them in a stack frame when the clause is invoked. Others, known as "unsafe" variables, are moved to the heap instead of the stack. To keep the diagrams simple, we will ignore these optimizations, since they do not invalidate our observations about patterns of memory referencing.

continue on the left branch of the tree. If the user requests another answer, the system backs up to the choice point, discarding stack frames built since the creation of the choice point, and then proceeds down the right branch. An OR-parallel system would create parallel processes instead of a choice point, and grow the cactus stack shown in the figure.

The main problem in implementing OR-parallel systems is that using a cactus stack can lead to conflicting bindings if sibling processes are allowed to bind variables created by their parent. The example illustrates how this can occur. An empty slot in a frame indicates 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, and each leads to a valid solution of the original goal. 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 this example, each process is going to try to bind a variable in the shared segment of the stack, and the bindings will be conflicting. If either process is allowed to write its binding in the cell for `X`, the other process will fail.

The problem can be summarized as follows: whenever a new process is started, it needs to have its own copy of any unbound variables occurring in the stack so far. Each process must be able to bind these variables to values it requires in order to proceed with execution in that portion of the goal tree. The goal of efficient parallel Prolog implementation is to build a virtual stack for each process, so that it can share as much information as possible with sibling processes and make copies only of that information that has to be bound uniquely by that process.

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.() A recent paper by Ciepielewski and Hausman describes different techniques for implementing this method and gives some performance results.() Competing representations have been proposed by Borgwardt,() Lindstrom,() D. S. Warren,() Yasuhara and Nitadori,() 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.()

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 seen by a sequential Prolog program: variable bindings are kept in an array of local stack frames and assorted heap cells, unification of a goal and the head of a called clause often requires access to cells at random locations in the stacks, 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 dereferencing variables via 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*,() references to nonlocal addresses are slower, since the access has to pass through switches to route it to the module that owns the memory. If the single address space is simulated, as it would be on a hypercube, for example the Cosmic Cube,() nonlocal references would be even slower, since the routing is done with software instead of memory mapping hardware.

An alternative configuration for processors and memories is to connect every processor to every memory through a large switch, as in the NYU Ultracomputer.() In this system there is a uniform amount of time to reference each memory cell. However, the pattern of references to unbound ancestor variables might still cause some congestion and hardware level delays, since the address of the ancestor may become a "hot spot" of the type investigated by Pfister and Norton.() When one memory address is accessed very often, the entire system can be adversely affected. Not only are the processors that compete to access the hot spot blocked until their accesses are satisfied, other processors may block since their requests for switch configurations could be queued behind those waiting for the shared variable. However, since most references to the shared variable will be reads, a switch that combines read accesses may alleviate this problem.

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 locations anywhere in its stack. There are two situations where unification must have access to the value of an ancestor variable, both illustrated in Figure **??**. 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 data from Ross and Ramamohanarao.() 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 even 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 was designed to be efficient in a distributed memory. It is not simply an adaptation of the three-stack model, since 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 frame of the calling clause and the frame for the called procedure. 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.

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. 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 in the generalized three-stack models. One advantage of the closed environment approach is that accesses to variable bindings during unification are uniformly fast, since there is no need to search for a binding in intermediate frames and auxiliary structures 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.() More recently, they have been incorporated into a parallel virtual machine, named OM, that executes compiled logic programs according to the AND/OR Process Model.() The use of closed environments in OPAL and OM is discussed below, after the description of term representation and unification using closed environments.

3 Definitions

In both the three stack and closed environment representations, variables are represented as tagged *value cells* of a constant size. When a variable is unbound, the cell is a pointer to itself. When a variable is bound to an atomic term, the representation of the atom overwrites the variable. Complex terms must be represented by a collection of value cells. When an unbound variable is unified with a complex term, a sequence of cells is allocated for the term, and a pointer to the start of the sequence is written over the variable. The first item in the sequence is used to hold the functor and arity of the term, and the remainder are for the arguments.

Representing an unbound variable by a pointer to itself is convenient when two unbound variables are unified: we simply overwrite one cell with the contents of the other. The cell that is overwritten is now a pointer to the other cell, which is still an unbound variable. This can lead to a chain of pointers, where the value of a cell that contains a pointer to another cell is the value at the end of the chain. *Dereferencing* a value cell is the process of following a chain of pointers until a nonvariable cell or a self-referential pointer is found.

A *stack frame* is a collection of value cells. When a clause is invoked, the system allocates a new frame to hold the variables of the clause, and initializes them all to unbound variables.[2] In sequential Prolog systems, frames are allocated from a data area known as the *local stack*. Collections of value cells used to represent complex terms are allocated from a different stack, called the *global stack* or *heap*. Complex terms in the heap are also called *instance terms*.

An *environment* is the collection of values accessible to the program as it is executed. In a sequential system, this is the entire local and global stack, since, as was described in the previous section, variable-variable unification can give rise to the need to access a value cell anywhere in

[2]Another difference between compiled and interpreted Prologs is that in compiler-based systems, frames do not have to be allocated and initialized all at once. Again, for reasons of simplicity, we will use a more abstract description of frames and environments.

either stack. In a parallel Prolog system there are multiple environments; the environment of a process is the set of local stack frames created for it (which corresponds to a path from the root to a leaf in the cactus stack) plus the heap terms pointed to from value cells in these frames.

Given these definitions, we can now describe the differences between the closed environment representation presented here and the three stack systems. The first difference is in the representation of variables. The pointer field of variable is a composite structure known as a *link*, containing a frame ID and an offset. To dereference a variable, a process looks up the starting address of the frame, and then adds the offset. In other words, we are using a segmented addressing scheme, where the list of frame IDs and their addresses is used as a segment table. How frame IDs are defined, and thus the size and shape of a process segment table, varies from system to system. In the AND/OR model, which has a large number of small-grain processes, the environment of each process has at most two frames, and the segment tables are thus quite small.

Since variable addresses are segmented, a variable can be dereferenced to many different values, depending on the current state of the segment table of the process that does the dereferencing. Two or more processes can share the same frame `F1`, and any variable in `F1` that has a frame ID field `F2` will have a value that depends on how the process maps `F2`. As an example, the two processes in Figure **??** each have an environment with two frames. Both use frame `1` for the first frame in their environment, but they have different second frames. The two processes dereference variable `2` of frame `1` differently: one process thinks the value of this variable is the constant `a`, and the other thinks it is the constant `d`.

Another difference between three stack and closed environment systems is that when a complex term has an unbound variable as an argument, the unbound variable is not stored in the heap, but in one of the stack frames. In Prolog systems, when a complex term is built on the heap, and one of the arguments is an unbound variable of the current clause, that variable is unified with the corresponding heap term argument, and the value cell for the variable in the local stack is changed to a reference to the variable on the heap. Here, we will do the opposite, and leave the stack term unbound and put a pointer to it in the heap.[3] This policy means that complex terms can be shared, since the variables in the terms can be dereferenced to different values. In the example in Figure **??**, each process shares the same heap term, which is pointed to from the shared frame, but they dereference the arguments of the term differently.

Pointers to instance terms can use the absolute address of the term on the heap. This is done with the understanding that the cell that contains the pointer and the heap are in the same address space. In a non-shared memory system, when a processor builds an instance term, it will put the term on the heap in its own local memory. Later, if and when a frame is copied, 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 variables in the frames of E or in heap terms pointed to from E have frame ID fields that are not in E. What this means is that when a process dereferences a variable of a closed environment, it does not encounter

[3]This means we will use more space in our stack frames, since one of the space optimizations in Prolog is that if the argument `X` in a heap term `f(X)` does not occur elsewhere in the clause, then we do not have to allocate space for it in the local stack. The policy of pointing from the heap to the stack guarantees that we have to allocate a cell for `X` in the stack.

a reference to a frame that is outside the environment. This implies that all value cells in the frames of E are either unbound variables, atomic terms, links that dereference to other cells in E, or pointers to complex terms whose variable arguments dereference to cells in E. A closed environment containing just one frame is called a *closed frame*.

An important observation is that when all the frames of a closed environment reside in the same memory space, it is not necessary to make nonlocal accesses in order to dereference a variable. Processes with their own closed environments can run on different nodes of a nonshared memory system, and not have to make nonlocal accesses to other nodes when dereferencing a variable during unification. In the AND/OR implementation described later, processes are allocated to processors dynamically. Since the environment of each process is small (at most two frames) and closed, the task allocator does not have to take into consideration the interaction between two process environments when it decides which process should be relocated to a new processor. The only interaction between two processes will be based on messages between the two, and not on the fact that one process needs values from the environment of the other.

So far we have just described the structure of closed environments, but not how they are created and maintained in closed form. In order to keep a small number of frames in each environment, it is necessary to transform frames periodically. The next section describes such a transformation, and then rules for applying it so that every unification involves two closed frames.

4 Environment Closing

The following is an algorithm that transforms a closed environment of two frames. Before the transformation, there may be references from either frame to the other, so that neither is by itself a closed environment. After the transformation, one frame will be closed, and all inter-frame references will originate in the other frame, which will be called the *reference* frame. We say this operation *closes CF* with respect to *RF*.

PROCEDURE Close(CF,RF):

CF is the frame to transform into closed form; RF is the reference frame.

1. *Reverse the direction of all links pointing from* CF *to* RF:

   For each cell X in *CF* that is a variable with ID field equal to *RF*, dereference X to a term T, and then:

   - If T is a nonvariable term, bind X to T.
   - If T is an unbound variable of *CF*, bind X to a reference to T.
   - If T is an unbound variable of *RF*, make X an unbound variable of *CF*, and bind T to a reference to X.

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

   For each cell X in *CF* that now points to an instance term, if there is an argument A of the term that is a link with ID field *RF*, dereference the link to a term T, and then:

   - If T is a nonvariable term, replace A with T. If T is a pointer to an instance term, execute this step recursively on the arguments of T.
   - If T is an unbound variable of *CF*, replace A with a reference to T.
   - If T is an unbound variable X of *RF*, create a new unbound variable Z in *CF*, bind X to a reference to Z, and replace A with a reference to Z.

Figure **??** 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 together 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 in Figure **??**.

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

$\leftarrow$ `r(a)` $\wedge$ `s(f(V))` $\wedge$ `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 is the unification and closing step that was illustrated in Figures **??** and **??**.

The environment closing operation is the key to using the notion of a closed environment to solve the shared unbound ancestor problem in OR-parallel systems. A key point is that when a unification is based on two frames *F1* and *F2* that are members of closed environments *E1* and *E2*, it cannot bind a cell in a frame that is not in either *E1* or *E2*. In particular, if *F1* and *F2* are both closed frames, all bindings will be made to value cells in either *F1* or *F2*, and the environment consisting of the pair $\{F1, F2\}$ is closed after unification. At this time either *F1* or *F2* can be closed with respect to the other, and be used as a closed frame in a subsequent unification.

It is possible to guarantee two closed frames for each unification by observing the following rules. In this discussion, the names *top* and *bottom* are used for the two frames under consideration. The names come from their relative position in Ferguson diagrams, which are explained later in Section **??** and Figure **??**. *Top* is the frame of the calling procedure, and `bottom` is the newly allocated frame for the called clause.

- 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 pair of frames used in the first unification are both 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 is now ready to be used in the solution of siblings of the goal.
- After each unification of a goal with the head of a nonunit clause, close the bottom frame with respect to the top frame. The bottom frame is now ready to be used as 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

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

The environment closing algorithm, as described above, calls for modifications to either input frame, and, possibly, to instance terms pointed to from the frames. When implementing the algorithm, a decision must be made as to whether the input frames can be modified in place, or whether new arrays of cells are allocated to represent the modified frames. In the version implemented in OPAL, the closed form of *CF* is written into a newly allocated frame, so the original *CF* is not modified. In this system, we often need to make a copy of *CF* before closing it, and the decision to write the closed form of *CF* into a new block of cells combines the copying and closing operations. On the other hand, *RF* is modified in place, since it will never again be referenced, and it can be deallocated after *CF* is closed. In the OM implementation, unification and closing are done in environment registers, and a new frame is allocated for the modified *CF* only when it is extended or a copy of *CF* is required, so much less copying is done.

The algorithm presented earlier makes two passes over *CF*. A one-pass algorithm is possible, but in general the closed form of *CF* 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 *CF* 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.() 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 these improvements, however, there are situations where a separate two-pass closing operation would give a more compact frame.

As a final point of discussion, it is interesting to note that after closing the bottom frame with respect to the top frame before solving goals in the body of a clause, variable to variable references are pointing in the opposite direction than in most Prolog implementations. 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,[5] 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. Another optimization possible with OM is that even these copies of the parent frame are not necessary, but the cost is in extra instructions at the time results are passed back to the original parent frame.

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 unbound variables. What makes this representation useful is that this form of dereferencing is never needed during unifications;

[5] Unless there is a cut, which can be viewed as a commit operation.

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.

5 Applications

Three systems have been implemented using closed environments. One is a pure OR-parallel interpreter, based on the virtual machine described by Crammond.() 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 was encouraging, but meaningful comparisons were not possible because of differences in the implementations and the host systems. The other two systems based on closed environments are outlined below.

5.1 OPAL

OPAL is an interpreter for the AND/OR Process Model, written in C and running under Unix 4.3. 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 **??**). 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 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 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 when the next result from the same descendant is returned. After values have been extracted from the bottom frame in the closing operation, it can be discarded. Further details can be found in More's M.S. thesis.()

5.2 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 and build choice points, 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.

One reason compiled Prolog programs are faster than interpreted programs is that the compiler detects the type of top level terms, and generates instructions that take advantage of this knowledge. For example, if the head of a clause is `p(a,X)` the two unification instructions are `get_const` and `get_var`. The former knows a passed argument must be either an unbound variable, in which case it can store a constant over the variable, or it must be an equivalent constant. It does not have to call the general dereferencing routine once for each term (one from the head and one from the call) and then take into account the multiple cases for the possible type of each term. Similarly, `get_var` can count on the fact that the corresponding variable from the called clause is unbound, and simply assign it the passed argument.

This same treatment has been applied to the environment closing algorithm, so that the instructions generated by the compiler to perform the closing operation at runtime are tuned to each call and clause head. The instructions, called `cput` and `cget` (the `c` is for *closing*), are optimized to take advantage of the type of each argument, plus knowledge of what must have been true if the corresponding `put` or `get` instruction executed earlier during unification was successful.

In the code compiled for an OR process, corresponding to each `get` instruction there is a `cput` instruction that is executed after the unification succeeds and before the results are returned to the parent process. Similarly, in the body of an AND process, there are `cget` instructions corresponding to each `put` instruction, which are executed after the OR process returns a result. A simple example of the kind of optimization that is possible is the instruction `cput_const`, which is generated for a constant in the head of a clause. This instruction is actually a NOP in all situations. If `cput_const` is ever executed, it means the corresponding `get_const` must have succeeded, and any variable in the current copy of the parent frame must already be bound to a constant, and thus in closed form. The closing algorithm merely requires an operation on any argument that could still be an unbound variable after the clause body has been solved, and by the time this instruction is executed, the argument must have been bound to a constant.

Further optimizations are possible when the input/output modes of procedures are known. If the compiler knows a procedure will bind a variable to a ground term, it does not have to generate closing instructions for that argument position, since the object of closing that position is just to make sure it is not still an unbound variable.

6 Other Methods for OR-Parallel Binding Environments

6.1 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.() 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.

6.2 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.() 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.

6.3 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.() 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.

6.4 The SRI Model

D. H. D. Warren independently developed another model based on binding arrays at SRI in 1983. Dubbed the "SRI Model" in a recent paper, it has been adapted for a parallel version of the WAM (). This paper describes an implementation that allows for fast initialization of binding arrays, so that when a process switches from one task to another nearby in the global process tree, the processor's binding array can be quickly updated.

6.5 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.() 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.

6.6 Kabu-Wake

The *kabu-wake*[6] method uses a different approach than the others discussed so far (Kumon *et al*,() Yasuhara and Nitadori()). 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.

6.7 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 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 (or during, in the case of OM's `get` instructions), 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.() 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.

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

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.

There is a significant disadvantage to the closed environment scheme, and that is the requirement to close top environments after the successful solution of a called clause. In three stack models, there is no need to refer to the calling environment again after the solution of the last goal in the called clause. The following program illustrates:

```
← p(X,Y) ∧ q(Y).

p(A,B) ← r(A,B).
r(a,Z) ← s(Z).
```

In a Prolog system, after the last goal in the body of a clause for `r` is solved, the system goes directly to the call to `q` from the top level goal. Any bindings for `Z` are written directly into the value cell for the top level variable `Y`, since variable to variable unification leads to pointers from new variables to old, so the value of `Y` is all set for the call to `q`. In OM, `Y` was left as a downward pointing link, and in order to get a value for it we have to return results via closing

operations on the intervening environments. After `r` is solved, the environment of `p` is closed with respect to the environment of `r`, and then the environment of the top level is closed with respect to the environment of `p`.

While this is clearly a problem for pure OR parallel systems, it is not necessarily a drawback for process-oriented systems like the AND/OR model. In the latter systems, it is necessary to return values through OR processes for a number of reasons. When the system exploits AND parallelism, the OR process helps coordinate multiple results, and is a place where a cache of previously used results can be stored.() Also, the OR process has the potential of being used to control combinatorial explosion in large problems; if results are forced to pass through intervening calls, they may be buffered instead of being passed directly back to calling processes and used immediately.

A problem related to passing results through intervening calls can be seen when `p` is deterministic, *i.e.* no choice points are created in calls to `p`, `r`, or `s`. Prolog and parallel Prolog systems can reuse the stack space allocated for `p` in calls to `r` and `s`. Known as *last call optimization*, this is a generalization of the more familiar tail recursion optimization used in functional languages. The variables in `p` are never referenced again after the call to the last goal in the body of `p`, as long as the called goals do not create choice points; in that case the environments would have to be saved to allow backtracked goals to access them.

Last call optimization will be harder to use in AND parallel systems, as the following program illustrates. `map_foo` applies the composition of functions `foo` and `bar` to each element of a list:

```
← map_foo([1,2,3,4],X).


map_foo([],[]).
map_foo([X1|Xn],[Y1|Yn]) ←
       foo(X1,T) ∧ bar(T,Y1) ∧ map_foo(Xn,Yn).
```

In this program, an AND-parallel system can "unroll" the loop and start parallel computations of `foo` on each list element. But even when both `foo` and `bar` are deterministic, the system cannot employ last call optmization and reuse the frame for `map_foo` on each recursive call. Recursive calls set up parallel processes, and each process needs a frame to hold the value of `T` and `Y1` for that invocation. Since OM will eventually exploit AND parallelism, the lack of last call optimization is not expected to be a drawback to the use of closed environments.

On the other side of the coin, there is a potential advantage to explicitly handling each frame as it is passed back from descendant processes. Whenever a top frame is closed with respect to a bottom frame, the bottom frame and whatever structures it points to can be deallocated. Each alternative clause at an OR choice point has its own bottom frame, and choice points below that have their copies of this frame, so it is not necessary to keep this frame once it has been used as the reference frame in a closing operation. The copying done by the environment closing operation could also act as the copying phase of an incremental garbage collector. Using a distributed memory representation will already be a bonus for garbage collection, since each processor can be made responsible for maintaining its own memory space, and garbage collection is much easier in the smaller memory address spaces in these systems. More experience with large programs is needed to see if the explicit deallocation afforded by knowing when bottom

frames can be thrown away will offset penalties involved in handling the frames in intervening processes.

7 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. Many of the benefits are derived from compiler optimizations that allow classification of variables into temporary *vs.* permanent variables and runtime detection of last call optimization, and these are applicable to OR-parallel systems as well as sequential Prolog systems.

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 "dance hall" configurations such as the Ultracomputer and "boudoir" configurations such as a hypercube. 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 closed environment 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. A potential benefit of closed environments is more efficient garbage collection, as a result of smaller independent memory spaces and early detection of unused frames.

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 helpful comments. Mike Gorlick and Carl Kesselman were also influential in later developments. It was Carl who pointed out that the addressing scheme used in variable representations was a form of segmented addressing, and that this might lead to optimizations in building and accessing complex terms.

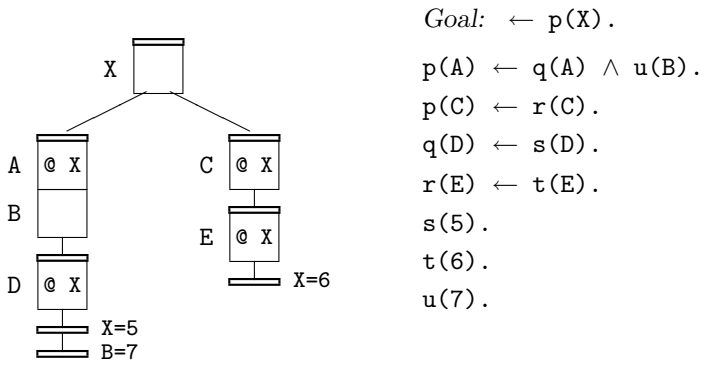
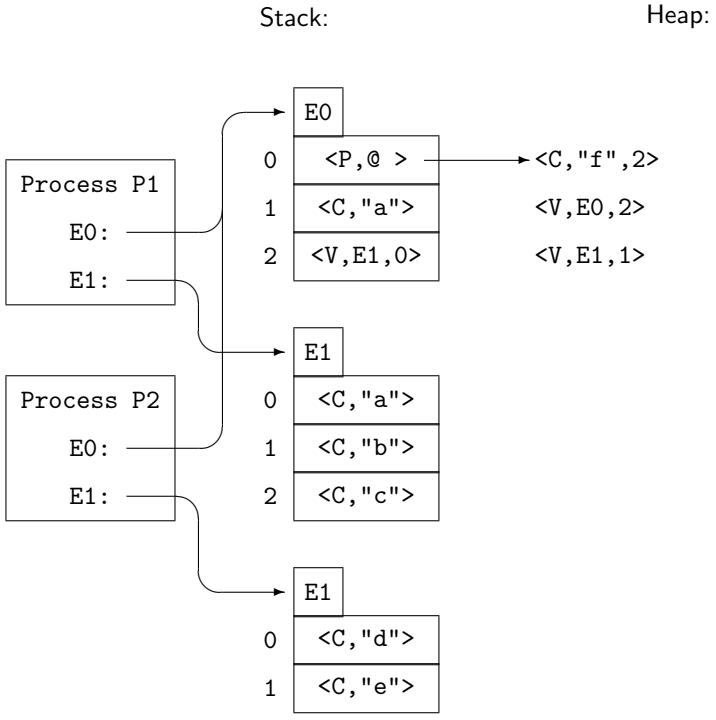
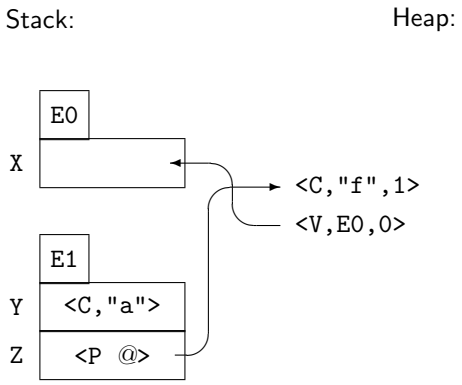


Figure 1: Environment Stack in OR-Parallel System



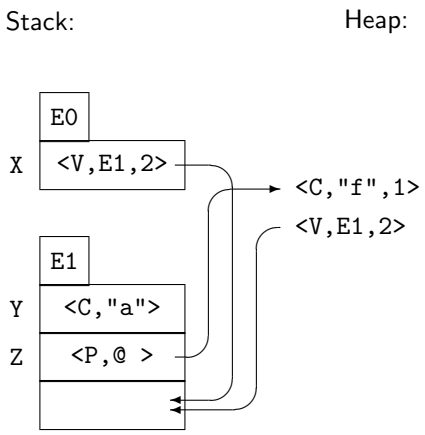
A value cell is shown in this figure as a tuple `<T,X>` *where* `T` *is a tag and* `X` *is a value. Possible tags are* `V` *for variable,* `C` *for constant (atom or functor name), and* `P` *for a pointer to an instance term. A process dereferences a link* `<V,E,N>` *by looking up the address of* `E` *and adding* `N`*. A link can be dereferenced to more than one variable, since each process interprets the environment ID field* `E` *in the link according to the set of frames in its own environment.* `P1` *dereferences the first variable of* `E0` *to* `f(a,b)`*, while* `P2` *would dereference the same variable to* `f(d,e)`*.*

Figure 2: Dereferencing a Link



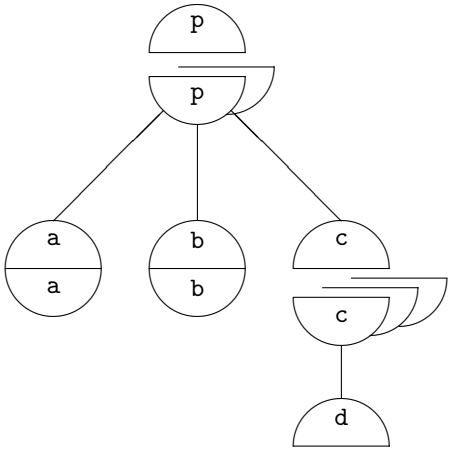
A unification may create links directly or indirectly between the two input frames, so that neither is guaranteed to be closed after the unification. This example shows two initially empty frames after unifying `p(a,f(X))` *with* `p(Y,Z)`.

Figure 3: Output of Unification



This figure shows the environments of Figure **??** *after closing* `E1` *with respect to* `E0`*. Note the addition of a new variable to* `E1`*, and the modification of the instance term.*

Figure 4: Environments After Closing *E1*




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

In a Ferguson diagram, upper half circles represent procedure calls, and lower half circles are heads of clauses. In OPAL, an AND process coordinates solution of goals in the body of a clause, and an OR process coordinates unifications of heads of clauses. Here there are two AND processes in the top plane: one for `p` *and one for* `c`*. OR processes for procedures with multiple candidates are drawn as overlapping bottom halves, such as the process trying to solve the call to* `c`*.*

Figure 5: Processes in OPAL

List of Figures