Optimizing And-Or Parallel Implementations

Gopal Gupta

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Bharat Jayaraman

Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260, U.S.A.

Abstract

This paper describes ways of improving the performance of combined and-or parallel implementations. The model for and-or parallel execution of logic programs that we consider integrates the class of methods for exploiting or-parallelism which perform variable access and task-creation operations in constant-time (such as Binding Arrays method and Versions Vectors method) and the RAP method for independent and-parallelism. The major run-time overhead in such and-or parallel models can be traced back to the overhead incurred for or-parallelism during processor task switching. We present a number of techniques for reducing this overhead, by exploiting and-parallelism and the semantics of Conditional Graph Expressions (CGEs). Essentially, we reduce the overhead by reducing the number of conditionally bound variables whose bindings need to be installed during task-switching. These techniques can be easily incorporated in a compiler, and can lead to substantial savings in execution time.

Keywords: Or-parallelism, And Parallelism, Binding Arrays, Versions Vectors, Conditional Graph Expressions.

1. Introduction

The obvious reason for combining and-parallelism and or-parallelism in a single framework is that any implementation that supports either alone is suboptimal compared with one that supports both. Another important reason is that when there is potential for both and- and or-parallelism in a single program, not exploiting and-parallelism can lead to unnecessary over-computation. For a trivial example, the pair of goals `?- append(X, Y, [1,..,m]), append(P, Q, [1,..,n])`, leads to a `O(m*n)` computational cost under a pure or-parallel or a sequential implementation, because all `n + 1` solutions for `P` and `Q` are re-computed for each of the `m + 1` solutions for `X` and `Y`. Instead of re-computing the solutions, if we compute the solutions only once and then combine them through a cross product, the computational cost would only be of the order `O(m + n)` (assuming that the time it takes to compute the cross-product of solutions is insignificant compared to the time it takes to re-compute the solutions). Note that goals where such solution sharing is possible frequently arise in database searching, symbolic integration, etc.

In an earlier paper [GJ89a, GJ89b], we presented an and-or parallel execution model that integrates the binding arrays method for or-parallelism [W84, W87] and the RAP method [D84, H86] for independent and-parallelism. In this paper, we first show that the proposed model is more general in that it can work with certain other environment representation schemes, especially version vectors [H87]. The main focus of this paper, however, is on techniques for improving the performance of a combined and-or parallel implementation, especially for the case when goals exhibit both and-parallelism and or-parallelism. The major run-time overhead in our and-or parallel model can be traced back to the overhead incurred for or-parallelism, i.e., the cost of task-switching. We present a number of techniques for reducing this overhead, by exploiting and-parallelism and the semantics of CGEs [H86]. Essentially, these techniques reduce the overhead by reducing the number of conditionally bound variables† (also called *conditional variables* in this paper) whose bindings need to be installed during task-switching. This is achieved by the following set of actions:

(i) Separating the local conditional variables from global conditional variables in the binding lists‡ and installing only global conditional variables during task-switching.

(ii) Recognizing conditional variables which are installed again immediately after they are removed from a processor's environment—a phenomenon which arises due to solution sharing as we shall shortly see—and saving the removal/installation steps altogether for them.

(iii) Taking advantage of the groundness information in CGEs to avoid installation of certain conditional variables.

These optimizations are such that they can be introduced by a compiler. As a result of these optimizations not only more parallelism is exploited, but the overhead incurred in implementing or-parallelism is also reduced. Thus, and-parallelism and or-parallelism are more efficient in our combined and-or parallel model compared to purely or-parallel [H87, CH86, W84, W87] or purely and-parallel models [D84, H86, LK88].

In order to show the generality of our and-or parallel execution, we first note that there are three

† conditionally bound variables are those variables which can receive more than one binding during execution.

‡ Note that binding lists are implemented as trail stacks in most parallel systems.

important criteria for an ideal or-parallel implementation: constant-time access to all variables, constant-time task-creation, and constant-time task-switching. By constant-time, we mean that the time taken is independent of the depth of the execution-tree and the size of the goal arguments. We have shown in [G90] that it is not possible to satisfy all three criteria simultaneously—a proof of this assertion is beyond the scope of this paper. Since an or-parallel implementation can satisfy at most two criteria, this can be used as a basis for classifying the various or-parallel systems proposed. For example, the Binding Arrays method [W84, W87] and the Version Vectors method [H87] offer constant-time task creation and variable access; the Directory Tree method [CH86] and the Environment Closing method [C87] offer constant-time variable access and constant time task-switch; and the Hash Windows method [B84] offers constant time task-creation and constant time task-switch. In our opinion, methods offering constant-time performance for variable access and task-creation are better than the others. This is because variable access and task-creation are the two most frequently performed operations and are entirely program-dependent, while task-switching depends on the scheduling strategy chosen and thus can be optimized by the run-time system. Our and-or parallel model has been implemented using the binding arrays approach, but we show that it can be used just as easily with the version vectors approach. Furthermore, although we use CGEs for and-parallelism, our technique can be used to implement languages such as Andorra Prolog [HB88, W90].

The rest of the paper is organized as follows: Section 2 briefly presents our model for combined and-or parallelism; section 3 shows that the model can be used with either binding arrays or version vectors; section 4 presents the optimization techniques for reducing task-switch time; and finally section 5 presents conclusions. We assume that the reader is familiar with or-parallel models [W84, W87, H87], Restricted And-Parallelism [D84, H86], and the Warren Abstract Machine (WAM) [W83].

2. A Model for Combined And-Or Parallelism

In this section we summarize our basic approach to combined and-or parallelism. More details, along with the design of an abstract machine which uses an extension of binding arrays for environment representation, can be found in [GJ89b, G90].

Figure 1 shows an extended and-or tree for a simple example. The extended and-or tree is an extension of the basic and-or tree. In addition to *and* nodes and *or* nodes, it has *cross-product* nodes to hold the cross-product of solutions from and-or parallel goals†, and *sequential* nodes which indicate the beginning of execution of *sequential goals* following the execution of a CGE. Nodes have space for their subgoals (also called goal-list), and or-nodes have space for the bindings of the variables occurring in the sub-goals. The binding of a conditional variable, along with its address, is also stored in a *binding list* associated with the node in which the binding is produced. The binding list serves the purpose of the trail stack during backtracking—all variables which appear in the binding list are marked to unbound—and also helps in updating the environment of a processor when it switches tasks and selects work at some other site in the tree.

† Note that in the extended and-or tree the the solutions to and-parallel goals are computed only once, and to produce all solutions the solutions of individual and-parallel goal have to be combined by a cross-product.

Cross-product nodes are introduced to facilitate sharing of solutions of and-parallel goals and serve to avoid the over-computation alluded to earlier. They are analogous to join-nodes in the PEPSys model [WR87], and are parents of and-nodes and sequential nodes. Each element in the cross-product set is a *tuple* which corresponds to one specific solution for the and-parallel goals. Thus the cross-product set corresponding to the cross-product node labeled `e X f` is $\{\langle \mathtt{e}^1, \mathtt{f}^1 \rangle, \langle \mathtt{e}^2, \mathtt{f}^1 \rangle, \langle \mathtt{e}^1, \mathtt{f}^2 \rangle, \langle \mathtt{e}^2, \mathtt{f}^2 \rangle\}$. The components of each tuple are addresses of terminal nodes of the and-branches. An and-branch is a branch rooted at an and-node, and consists of those nodes lying along a path from the and-node to a leaf node (which represents *one* solution for the associated and-parallel sub-goal). For example, nodes labeled `b` and $\mathtt{b}^1$ form an and-branch rooted at `b`. Note that we avoid an explicit join of bindings produced for variables in the and-parallel sub-goals. Instead, we represent the cross-product set using addresses of terminal nodes.

The structure of the extended and-or tree, except perhaps for the sequential nodes, should be self-evident from figure 1 (note that in the figure (`b` || `c`) indicates that `b` and `c` can be executed in and-parallel). We therefore clarify the role of sequential nodes. For example, suppose that the and-nodes `e` and `f` have produced solutions $\mathtt{e}^1$ and $\mathtt{f}^1$ respectively. A tuple $\langle \mathtt{e}^1, \mathtt{f}^1 \rangle$ would then be inserted in the cross-product set associated with cross-product node labeled `e X f`, and a sequential node labeled $\langle \mathtt{e}^1, \mathtt{f}^1 \rangle$ would be created corresponding to this tuple to solve goal `g`, where `g` is the next (sequential) goal to be solved after the execution of CGE containing `e` and `f` is over. The sequential nodes are inserted as children nodes of the corresponding cross-product node. The creation of other tuples $\langle \mathtt{e}^1, \mathtt{f}^2 \rangle$, $\langle \mathtt{e}^2, \mathtt{f}^1 \rangle$, etc., results in the creation of the corresponding sequential nodes. Similarly, when goals $\mathtt{g}^1, \mathtt{g}^2$, etc. are solved, tuples $\langle \mathtt{b}^1, \mathtt{g}^1 \rangle$, $\langle \mathtt{b}^1, \mathtt{g}^2 \rangle$, etc. are inserted in the cross-product set of the cross-product node `b X c`, and sequential nodes labeled $\langle \mathtt{b}^1, \mathtt{g}^1 \rangle$, etc. are created as children of that cross-product node for the execution of goal `d`.

Three points should be noted in the above discussion : (a) Since the cross-product set is represented using address of terminal nodes, a processor before executing a goal corresponding to sequential node labeled with the tuple $\langle t_1, t_2, \ldots, t_n \rangle$, should first update its environment with the bindings made in the and-branches corresponding to $t_1, t_2, \ldots, t_n$. (b) Initiating the execution of each sequential goal, such as $\langle \mathtt{e}^1, \mathtt{f}^1 \rangle$ depends only on the completion of $\mathtt{e}^1$ and $\mathtt{f}^1$ and not on other and-branches of `e` and `f`. Hence execution of a sequential goal may be initiated, even though all the solutions to preceding and-parallel goals may not have been found yet. (c) There are many sources of parallelism: (i) two or more or-nodes could be executed in parallel giving rise to or-parallelism; (ii) two or more and-nodes could be executed in parallel giving rise to and-parallelism; (iii) two or more cross-product tuples could be executed in parallel, or an or-node and a cross-product tuple could be executed in parallel, giving rise to *consumer instance parallelism* (using the terminology in [RK89]).

The above is a general scheme for exploiting and-or parallelism in logic programs. An abstract machine architecture similar to the WAM, using an extension of binding arrays for representing multiple or-parallel environments, has been designed for this model [GJ89b]. The data areas of this abstract machine consist of a local stack, for allocating space to various nodes and local variables, and a global stack (or heap), for allocating the heap variables. The extended and-or tree lies in node stacks accessible to all processors. Execution is started by placing the root node in the node stack of one of the processors. Processors traverse the branches of the tree, executing sub-goals in the nodes, and growing and contracting the tree in the process. Since the number of branches in the and-or tree would be much larger than the number of processors, each processor ends up executing more than one branch of the tree. This is accomplished through backtracking on success/failure. The movement of a processor from its current site to the place where work is available is called *task switching*. Task switching involves updating the environment so that all the bindings can be correctly accessed at the new site of work.

Before we describe how environment can be represented in our model, we show that solution sharing indeed incurs less overhead than re-computing solutions. Consider two independent sub-goals `a, b` where `a` has 4 solutions and `b` has 3. If we re-compute the 3 solutions for `b` for every solution of `a`, then all the variables (conditionally or unconditional) appearing in sub-goals `a` and `b` would be accessed *at least once* during unification. In general they would be accessed more than once: first, to be initialized with a value, and perhaps later for unification with another value. (For example, in the WAM, a conditional (logical) variable is accessed once during the `put_variable` instruction to allocate space and later again when it receives a binding value.) If instead of re-computing the solutions for sub-goals, a processor shares the and-branches produced for `a` and `b` by other processors, then the only overhead it incurs is in installing bindings from binding-lists of the nodes in the and-branches. Since every conditional variable in the and-branch would be accessed only once during binding installation, and, in general, the number of conditional variables is much smaller than the total number of variables in the and-branch, the number of variable accesses are much less during installation than during re-computing. Note that we have not even considered the overheads arising out of environment and node allocations during re-computing, which are not incurred during solution-sharing and which make it even more efficient.

3. Environment Representation

In the next two subsections, we show how the two most well known environment representation schemes with constant time task-creation and variable-access but non-constant task-switching time [W87, H87] can be extended to incorporate and-parallelism. The extensions ensure that bindings to all variables are accessed in constant-time even in the presence of and-parallelism.

3.1. Extending the Binding Arrays Method

It should be first noted that the binding arrays method [W84, W87] will not work correctly in the presence of and-parallelism. To see the problem, consider the goals (p, (q1 || q2), r), where q1 and q2 also exhibit or-parallelism. Suppose further that goal p has been completed. In order to execute goals q1 and q2 in and-parallel, it is necessary to maintain separate binding arrays for them. As a result, the binding-array offsets for any conditional variables that come into existence within these two goals will overlap. Thus, when goal r is attempted, we are faced with problem of merging the binding-arrays for q1 and q2 into one composite binding-array or maintaining fragmented binding-arrays.

To solve the above problem, first recall that in the binding-array method [W84, W87] an offset-counter is maintained for each branch of the or-parallel tree for assigning offsets to conditional variables. However, offsets to the conditional variables in the and-parallel branches cannot be uniquely assigned, since there is no implicit ordering among them; at run-time a processor can traverse them in any order.

We introduce one more level of indirection in the binding array to solve this problem. In addition to the binding array, each processor also maintains another array called the *base array*. As each and-node is created, it is assigned a unique integer id. When a processor encounters an and-node, it stores the offset of the next free location of the binding array in the ith location of its base array, where i is the id of the and-node. The offset-counter is reset to zero when an and-node is created. Subsequent conditional variables are bound to the pair $\langle i, v\rangle$, where v is the value of the counter. The counter is incremented after each conditional variable is bound to this pair. The dereferencing algorithm is described below:

```
deref(V)          /*unbound variables are bound to themselves*/
term *V {
    if V→tag == VAR
            if not V→value == V
                    deref(V→value)
            else V
    else if V→tag == NON-VAR
            V
        else {          /*conditional var bound to <i, v>*/
            val = BA[v + base[i]];        /*BA is the binding array.*/
            if val→value == val
                    V
            else deref(val) }}
```

Note that access to variables is constant-time, though the constant is somewhat larger compared to the binding-arrays method for pure or-parallelism. Also note that now the base array is also to be updated on a task-switch. More details of this extension can be found elsewhere [GJ89b, G90]

As binding to conditional variables are made they are recorded in the binding array as well as in the binding list. There are two situations where a processor needs to update its binding array: The first is when the processor decides to continue with the execution of a sequential node, after the completion of an and-branch and generation of a tuple thereafter. Before it can do so, it needs to update its binding array with the bindings created for the conditional variables in other and-branches in the tuple. Note that this updating overhead is in lieu of re-executing the and-branches. The second situation where the binding array needs to be updated arises when a processor runs out of work. The processor then selects a site in the tree where work is available and task-switches to start execution from there. Task-switching involves removing from the binding array those bindings made to conditional variables between the current node and the common ancestor node of the current node and the new site, and installing those bindings made to conditional variables between the new site and the common ancestor (we call the former operation binding *removal* and the latter binding *installation*). Note that if a sequential node (with associated tuple $\langle t_1, t_2, \ldots, t_n\rangle$) is encountered during task-switching then all the bindings made to conditional variables in the and-branches $t_1, t_2, \ldots, t_n$ are removed from, or installed into, the binding array depending upon where the sequential node lies.

3.2. Extending the Version Vectors Method

In the version vectors method [H87], conditional variables are bound to a vector of values. The length of the vector is n, where n is the number of processors. A processor with processor id i stores its corresponding binding in the ith slot of the associated vector. During task-switching processors install the correct binding of all the conditional variables, which appear in the binding lists of the intervening nodes, in the appropriate slot in the corresponding version vectors.

The problem that arises in the versions vectors model due to the presence of and-parallelism is the following: Given the goal and the tree shown in fig. 1, if a processor with id i is working on the and-parallel sub-goal `b` and another processor with id j is working on sub-goal `c`, then conditional bindings imparted by any one of them are not visible to the other. This is because binding generated by processor i is stored in the ith slot of the version vector, while that generated by j is stored in the jth slot. Since during variable access processors look up only their associated slot, bindings produced by i are not visible to j and vice versa. Thus, before processor i or j can continue with the sequential goal `d` (whose associated tuple is, say, $\langle b^1, c^1 \rangle$) processor i must update its slots in the version vectors for conditional variables which have been imparted a binding by processor j during execution of and-branch c^1, while processor j must update its slots in the version vectors for conditional variables which have been imparted a binding by processor i during execution of and-branch b^1. The dereferencing algorithm remains the same as in pure or-parallel case. The version vectors are needed to be updated in a manner similar to the binding arrays method during task-switching. For the extended versions vectors method the operations of *binding removal* and *binding installation* are taken to mean removal and installation, respectively, of bindings from version vectors of conditional variables rather than from the binding array.

It is perhaps obvious now, given an environment representation method for implementing or-parallelism with constant-time task creation and variable access, but non-constant time task-switching, how to incorporate it in the extended and-or tree. The part of the environment which is maintained privately by each processor, i.e. bindings of conditional variables, would have to be updated for each of the and-branches $\mathtt{t}_1$, $\mathtt{t}_2$, ..., $\mathtt{t}_n$ before a processor can continue with a sequential node labeled with the tuple $\langle \mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n \rangle$. In addition, if such a sequential node is encountered during task-switching, all the conditional bindings along and-branches $\mathtt{t}_1$, $\mathtt{t}_2$, ..., $\mathtt{t}_n$ would also have to be installed in, or removed from, the private part of the environment depending on where the sequential node lies. For example, the private part of a processor in case of the binding arrays method is its private binding array while in case of the versions vectors method it consists of its private slots in the version vectors.

4. Optimizations

In this section we present techniques for reducing the major source of overhead in our combined and-or model, namely, task-switching, which involves traversing the tree and updating the environment. Earlier attempts to reduce task-switching overheads can be found in [H87, W87]. However, these techniques (known as *variable promotion* and *delayed installation*) are meant for purely or-parallel systems, and as such are applicable to and-or parallel systems too. In this paper we focus only on techniques that are possible due to the presence of and-parallelism. These techniques are (i) not very difficult to implement, (ii) can result in substantial gain in time, (iii) can be easily incorporated in a compiler [G90], and (iv) are independent of the work-scheduling strategy chosen. As a result of these optimizations, not only is more parallelism exploited due to introduction of and-parallelism in an or-parallel system the overhead incurred in implementing or-parallelism is also reduced.

4.1. Splitting the Binding List

The first optimization is based on the observation that after an and-parallel subgoal `G` has been solved, subsequent goals need only the bindings produced for `G`'s unbound variables. For example, in the CGE

(independent(X, Y) | a(X, 2) || b(Y, 8)), d(0, X, Y).

if `independent(X,Y)` evaluates to true because both `X` and `Y` are unbound variables, then the sub-goal `d`, as well as other following sub-goals, use only the bindings produced for `X` and `Y`. Thus, once all the solutions for `a` and `b` have been produced, the processors picking up sequential sub-goal `d` for execution need install only those conditional variables which appear in the arguments of the sub-goals `a` and `b` and not those which may have come into existence within the and-branches of `a` and `b` during execution. However, there are two difficulties with this approach.

(i) The variables in the top-level goal (for example, variables `X` and `Y` above) may dereference to a conditional variable which may get bound within the and-branch (fig. 2). The binding of such conditional variables, thus, may lie in the binding list of nodes deep down in the and-branches. Since we cannot tell *a priori* the node which is going to bind a variable, accessing the binding of these variables for binding installation or removal requires traversing all the binding lists within the and-branch.

(ii) Conditional variables occurring within structures occurring in the and-branches may be accessed indirectly later, e.g., when that structure is accessed from some other goal not in the and-branch during unification (Fig. 3). Installing or removing the bindings of such conditional variables again requires traversing all the binding lists within the and-branch.

It turns out that although we cannot completely avoid traversing binding lists, we can considerably reduce the amount of work that is involved in installing the conditional variables in the and-branches.

If we follow WAM conventions, then in case (ii) all conditional variables occurring within structures would be global variables and would be allocated space on the heap. Thus, if we maintain separate binding lists for global conditional variables and local conditional variables† and install bindings appearing only in the binding list of global variables, then the problem mentioned in (ii) would be eliminated. As a result of maintaining separate binding lists the amount of work done during binding installation would be decreased since we only install global conditional variables, saving the work for local conditional variables.

However, splitting the binding list and installing bindings from only global binding lists solves the problem mentioned in case (ii) but it doesn't solve the problem mentioned in case (i). Consider case (i) again. Note that:

(a) A variable in the argument of an and-parallel sub-goal (e.g. `X` and `Y` above) cannot dereference to a variable (conditional or non-conditional) which comes into existence within the and-branch because of the WAM convention that "newer" variables always point to "older" ones.

† In an implementation which represents binding lists as a trail stack, the trail stack would be split into two.

(b) If a variable in the argument of an and-parallel sub-goal (e.g. X and Y above), which dereferences to an unbound conditional variable in an ancestor node, gets bound within the and-branch, then this binding would be imparted to the unbound conditional variable since in WAM variables are dereferenced before they are bound. The address-value pair for this binding would be kept in the global or local binding list within the and-branch depending on whether the conditional variable is global or local.

Examining case (b) further, if the conditional variable is global and the pair is kept in the global binding list then no problems arise since bindings in global binding lists are installed during task-switching. If the conditional variable is local then it will *not* be installed during task-switching since processors do not install from local binding lists. To remedy this situation we "globalize" such local conditional variables before initiating the execution of the sub-goal, so that if it gets bound within the and-branch its address-value pair is kept in the global binding list and gets properly installed. By globalizing we mean that the variable is moved from local stack to the heap. The globalization step can be carried out when the argument variable is being loaded into argument registers (during WAM `put_value` instruction). This results in a new instruction (which we call `put_and_value`) similar to WAM's `put_value` instruction but which also globalizes the variable to which the argument variable dereferences to. This new instruction is very similar to WAM's `put_unsafe_variable` instruction except that in the latter a local variable is globalized only if it is allocated space in the *current* environment.

As a special case if the argument variable dereferences to itself (i.e. it is the first occurrence of a local conditional variable) then the variable itself is globalized. As in the previous case the globalization step can be carried out when the argument variable is being loaded into argument registers (during WAM `put_variable` instruction). This results in a new instruction (which we call `put_and_variable`) similar to WAM's `put_variable` instruction but which also globalizes the argument variable.

The net result of the above two new instructions is to globalize local conditional variables to which variables in the arguments of an and-parallel sub-goal might dereference to. As a result of globalization, if a processor picks a sequential goal labeled $\langle \mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n \rangle$ then it need install only the bindings in global binding lists of and-branches $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n$. Similarly, if this sequential node is encountered during task-switching, binding installation/removal need be done for only bindings occurring in global binding lists of and-branches $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n$. Thus the work of installing local conditional variables is saved. Empirical studies [TD87] have shown that in many benchmark programs the number of local conditional variables is a significant percentage of the total number of variables which come into existence during execution (approximately 20% on average, for large benchmarks). Thus, not having to install bindings from local binding-lists during task-switching leads to considerable savings in time.

One should, however, note that binding installation from the local binding lists of an and-branch cannot be avoided if the processor task-switches to a node in the and-branch for the purpose of finding a solution for the associated and-parallel sub-goal. This is because these bindings may be needed during execution of the sub-goal. Thus, installation of bindings from the local binding list for an and-branch can be avoided only if a solution has already been found and the processor wants to continue with the execution of sequential node or a node below the sequential node. For example, in figure 1 if a processor switches to node labeled **e X f** it needs to install bindings from both local and global binding lists along the and-branch $\mathtt{c}$-$\mathtt{c}^1$-$\mathtt{h}^1$, while a processor switching to node $\langle \mathtt{b}^1, \mathtt{g}^1 \rangle$ needs to load only from the global binding lists along and-branches $\mathtt{b}$-$\mathtt{b}^1$ and $\mathtt{c}$-$\mathtt{g}^1$.

We illustrate the split-binding-list optimization through an example, consider the program taken from the benchmark suite of the ROLOG compiler [RK89] and reproduced in appendix A (program A1). Consider the query:

```
?- fibper(200, 30, X).
```

and the clause for fibper.

```
fibper(Fib, Perfect, Prime) :-
    (ground(Fib, Perfect) ∧ independent(Fibo, Perfecto) | fib(Fibo,Fib) || perfect(Perfecto, Perfect)),
    Prime is Fibo+Perfecto,
    isprime(Prime).
```

There are 13 solutions for fib(fibo, 200) and 3 solutions for perfect(Perfecto, 30), and 12 solutions in all for the top-level query. Consider the CGE in the clause for fibper, the AO-WAM byte-code [GJ89b] generated for the CGE would be the following:

```
fibper/3:
        ....
        ....
        check_me_else seq_code             %jump to seq_code if any check fails.
        check_ground Fib                   %check if Fib is ground.
        check_ground Perfect               %check if Perfect is ground.
        check_independent Perfecto, Fibo   %check independence of Fibo & Perfecto.
        alloc_cross_prod 3, ADDR           %Allocate cross-prod. node for parallel execution
        put_and_value Fib, A1              %load argument registers for fib using put instructions
        put_and_variable Fibo, A2          %modified for split-binding-list optimization.
        push_and_call fib/2                %push the call for fib in a queue, to be
                                           %picked up by another processor.
        put_and_value Perfect, A1          %load argument registers for perfect using put instructions
        put_and_variable Perfecto, A2      %modified for split-binding-list optimization.
        call perfect/2, 3                  %start execution of perfect.
ADDR:       /* code for goals Prime = Fibo+Perfecto, isprime(Prime) starts here*/
        ....
        ....
seq_code:   /* code for sequential execution of fibper starts from here.*/
```

Bytecode for Fib+Perfect Query

The only conditional variables that can get bound within the and-branches are Fibo and Perfecto. Note that no structures are present in the programs for fib and perfect, so most variables are allocated space in the node-stack rather than the heap. However, due to the `put_and_variable` instruction, Fibo and Perfecto would be globalized and allocated in the heap. Since there are no other global variables the global binding lists of the and-branches corresponding to sub-goals fib and perfect would contain only one variable each. Thus when a processors picks a tuple from the cross-product node of the goals fib and perfect, it has to update its private environment (recorded in binding arrays or versions vectors) only with 2 variables as a result of this optimization. Note that the number of local conditional variables which are created and bound within the and-branches corresponding to sub-goals fib and perfect is fairly large since clauses for both have a number of conditional variables and these clauses are recursively called a large number of time within the and-branch. Thus, for a low cost of splitting the binding list, we gain considerable efficiency since installation of (a large number of) local variables can be avoided during task-switch.

4.2. Common Tuple-Elements

The second optimization is concerned with reducing the number of conditional bindings to be installed during task-switching by recognizing the and-branches which are immediately installed again after removal.

When a processor is backtracking and possibly removing bindings of conditional variables bound in the and-branches of a cross-product tuple, it is very likely that after getting to the cross-product node it will pick up another tuple to continue execution (the tuple set is recorded as a sequence of tuples in the cross-product node). The bindings of conditional variables bound in the and-branches corresponding to the new tuple would be installed before execution is begun. However, the new tuple might have some elements common with the old tuple whose bindings were just removed, which we would have to install again. Thus, a natural improvement would be to save the removal/installation steps for the common elements in the current tuple and the next. This improvement has two advantages – not only less work is done, the contention for the node-stacks and binding lists is also reduced.

For example, suppose we have the following goal `?- (true, a || b)`, during whose execution `a` and `b` are executed in and-parallel. Let us assume that solutions $\mathtt{a}^1$, $\mathtt{a}^2$ and $\mathtt{a}^3$ are found for sub-goal `a` and solutions $\mathtt{b}^1$ and $\mathtt{b}^2$ are found for sub-goal `b`. Using a straightforward algorithm (the one Prolog uses to find all solutions) the cross-product tuple set would be generated with the following order of tuples: $\langle\langle \mathtt{a}^1, \mathtt{b}^1\rangle, \langle \mathtt{a}^1, \mathtt{b}^2\rangle, \langle \mathtt{a}^2, \mathtt{b}^1\rangle, \langle \mathtt{a}^2, \mathtt{b}^2\rangle, \langle \mathtt{a}^3, \mathtt{b}^1\rangle, \langle \mathtt{a}^3, \mathtt{b}^2\rangle\rangle$. We refer to the algorithm which generates this set as the naive algorithm. Suppose further that the tuple $\langle \mathtt{a}^1, \mathtt{b}^1\rangle$ is picked by a processor to continue the execution of the sequential goal. Once the execution of the sequential goal is done, the processor starts to backtrack. When the processor reaches the sequential node corresponding to $\langle \mathtt{a}^1, \mathtt{b}^1\rangle$ during backtracking, it further backtracks through the tuple and removes the conditional bindings along the and-branches $\mathtt{a}^1$ and $\mathtt{b}^1$. After the binding removal is done it picks another tuple from the cross-product node and continues with the execution of the sequential goal to find another solution. Now consider what happens if the tuple $\langle \mathtt{a}^1, \mathtt{b}^2\rangle$ is selected as the next tuple for execution. Since bindings along the and-branch $\mathtt{a}^1$ are being installed again, the effort spent in removing them during backtracking is gone to waste. A more efficient strategy, once the sequential node is reached during backtracking, would be for the processor to select the tuple it wants to execute next and then remove and install bindings from only those and-branches *in which the two tuples differ*. Thus, in our example, we should simply remove bindings in and-branch $\mathtt{b}^1$ and install those along $\mathtt{b}^2$ and continue with the execution of the sequential goal. Note that this saves the work of installing and removing bindings along $\mathtt{a}^1$, thus saving some time in switching from one tuple to another. The savings can be even more if the arity of the tuples is larger and more elements are common among the two tuples.

The incorporation of the optimization described above poses one problem. When we backtrack to a sequential node, we would like to choose the tuple which has the maximum number of elements common with the current tuple. This can be a non-trivial task since it would involve comparing the current tuple with all the existing tuples in the cross-product node. However, there is an efficient method of generating the cross-product tuple set such that two adjacent tuples differ in their component elements *only in one position.* When a processor reaches a sequential node during backtracking, it selects the tuple after the current one in the tuple set. In this way the processor just has to remove/install bindings from only one and-branch. If we generate the above tuple set in a way such that adjacent tuples differ only in one element then it would look as follows : $\langle\langle \mathtt{a}^1, \mathtt{b}^1\rangle, \langle \mathtt{a}^1, \mathtt{b}^2\rangle, \langle \mathtt{a}^2, \mathtt{b}^2\rangle, \langle \mathtt{a}^2, \mathtt{b}^1\rangle, \langle \mathtt{a}^3, \mathtt{b}^1\rangle, \langle \mathtt{a}^3, \mathtt{b}^2\rangle\rangle$.

After a processor finishes executing its current tuple and picks the next one in the tuple-list, hoping that it would have to remove/install variables only along one and-branch, it is possible that that tuple may have already been picked up by another idle processor. In such a situation, the processor picks the next available untried tuple in the tuple list. The next tuple obtained thus may differ in more than one element positions, but still it would be the untried tuple "nearest" to the current one and hence would require least amount of removal/installation steps (because tuples in the tuple-list are arranged in a fashion that consecutive tuples differ in only one element position). However, now the processor has to do an element-wise comparison of the current tuple and the chosen tuple to determine the elements in which the two differ before removing/installing variables along the corresponding and-branches.

We now give an algorithm for generating the cross-product set with the tuples in the order desirable for common tuple-elements optimization. The algorithm has the same complexity as the naive algorithm. We assume that solutions to each and-parallel sub-goal are organized as separate circular lists in the corresponding and-nodes. Also assume that the first tuple is arbitrarily generated by selecting one arbitrary element from each solutions list. (Actually, in a practical system there are some advantages in choosing the first tuple carefully which we will not discuss here to keep matters simple.) To understand how the algorithm works we first describe the naive algorithm informally through a series of steps below:

(i) Copy first tuple into the current tuple. Let the right most element position in the current tuple be the current position. Print current tuple.

(ii) If the elements at the current position have gone through one complete cycle in their solution list then move current position one element to the left in the current tuple. Repeat this step until a position is found whose elements have not gone through one complete cycle in the solution list. If we move beyond the leftmost position, the entire cross-product set has been generated and the algorithm terminates.

(iii) Replace all elements at positions to the right of current position in the current tuple by elements at corresponding position in the first tuple.

(iv) Replace the element in current position in the current tuple by the next element in the corresponding solution list. Print current tuple. Move current position to the right most element in the current tuple. Go to step (ii).

In the naive algorithm, as a result of step (iii) adjacent tuples differ in their elements in more than one position. If we omit this step then step (i), (ii) and (iv) would give an algorithm which generates the cross-product set with the desired characteristic. We call this derived algorithm the optimized algorithm. Note that in the optimized algorithm, determining that a position in the current tuple has cycled through the solution list is difficult because in each iteration the cycle starts from a new element. To overcome this problem we have to remember the element from where the cycle of a particular solution list started.

Below we give precise program code for the optimized algorithm. Note that `next` is a function which, given a solution, returns the next element in that solution's corresponding circular solution list. The variable `arity` is initialized to the size of the tuple (number of sub-goals being executed in and-parallel). Array variable `first_tuple` is used for storing the first tuple. while array variable `ctuple` is used for remembering where the cycle started in different solutions lists during each iteration. The elements of the cross-product set appear in an array called `tuple_set` once the program terminates. Variables `top` and `base` are pointers into the tuple_set array and serve as pointers to current position and current tuple respectively. We use the programming syntax of C to describe the algorithm. Figure 4 illustrates how this algorithm works by a simple example.

```
/*ele_t is the type of tuple elements*/
ele_t ctuple[arity], first_tuple[arity], tuple_set[MAXSIZE];
ele_t *top, *base, *temp; int j,k;
compute_tuple_set()
 {base = tuple_set;
  top = base + arity - 1;
  for (j = 0 ; j < arity ; j++)
    {
    tuple_set[j] = first_tuple[j];        /* Copy the first tuple into tuple_set*/
    ctuple[j] = first_tuple[j];         /* Start the cycle from the first tuple*/
    }
  while(true)
   { while(next(*top) == ctuple[top - base])  /*Starting from top element*/
       { ctuple[top - base] = *top;      /*in tuple_set for each element */
         top = top - 1; check if the cycle is completed*/
        if (top < base)                    /*If cycles completed for all elements, exit.*/
           { printf("Cross-Product Set Enumeration Over");
             return;
           }
       }
    temp = base; base = base + arity; /*Start generating next tuple in tuple_stack.*/
    for (k = 0 ; k < arity ; k++)
         base[k] = temp[k];      /*Create copy of most recent tuple in tuple_stack.*/
    top[arity] = next(*top);       /*Modify the top element of the copied tuple.*/
    top = base + arity - 1;       /*Update top and repeat the cycle.*/
   }
 }
```

Program 1: Optimized Generation of the Cross-product Set

We illustrate this optimization too with the help of a program similar to fib + perfect = prime program shown in appendix A. The program is shown in appendix A (program A2). Consider the CGE shown in the query. There are 10 solutions for fib (Fibo respectively equal to 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34), 3 solutions for perfect (Perfecto respectively equal to 1, 6, and 28) and 4 solutions for fact (Facto = 1, 2, 6, and 24 respectively). Suppose a processor generates the solution Fibo = 5 for the first and-branch. Let's assume that the first two solutions for perfect and fact have already been produced in the second and third and-branch. The processor would then construct the cross product of Fibo = 5 with Perfecto = 1, 6 and Facto = 1, 2 and then continue with the sequential goal, picking the tuple $\langle$Fibo = 5, Perfecto = 1, Facto = 1$\rangle$. Once the sequential goals following the CGE have been performed, the processor would backtrack and try to pick the next tuple. If it chooses the next tuple to be $\langle$Fibo = 5, Perfecto = 6, Facto = 1$\rangle$ then it has to remove and install bindings from the and-branch corresponding to sub-goal perfect only. Thus, the work of removal and installation along the and-branches corresponding to sub-goals fib and fact is saved. Note that in this case the work saved due to common tuple elements is not much because the number of bindings installed/removed from an and-branch is only one due to the split-binding-list optimization. However, if the number of bindings in the global binding lists of the and-branches was large (for example, if there were structures in the program containing conditional variables) then the work saved would be much more.

This optimization would be particularly useful for data-base applications, where large amount of or-parallelism within independent and-parallelism is found.

4.3. Ground CGEs Optimization

Frequently the CGEs are of the form

(ground(X,Y) | b(X,Y) || c(X,Y) || d(X,Y,Z)).

In such CGEs, all variables appearing in some of the sub-goals (e.g. sub-goals `b` and `c`) would be bound to *ground* values if the condition succeeds. In such cases, there is no need for processors to install bindings from binding lists (global or local) of and-branches of such sub-goals. This is due to the following facts :

(i) As mentioned before, a sub-goal uses bindings of only those variables of another sibling sub-goal which appear in the sibling sub-goal's arguments. Thus, a sequential sub-goal accesses only the bindings of variables appearing in the arguments of and-parallel sub-goals.

(ii) If a conditional variable is ground before the condition in the CGE is evaluated, then the node where the variable is bound to ground value would be an ancestor node of the cross-product node corresponding to the CGE. The bindings of such a conditional variable, thus, would already have been installed by a processor before it reaches the cross-product node corresponding to the CGE.

(iii) Semantics of CGEs require that only variables occurring to the left of a CGE can occur in its condition part.

Hence, if we can infer from the evaluation of the condition of a CGE that all variables in arguments of an and-parallel sub-goal in the CGE are ground, then, during task-switching, by (ii) and (iii) the bindings of such variables must already have been installed when a processor reaches the cross-product node corresponding to the CGE. Now, since there are no unbound variables in arguments of the sub-goal, by (i) there is no need for a processor to install bindings from binding lists (global or local) within the sub-goal's and-branch.

Hence, if all variables appearing in the arguments of a sub-goal are ground, the processor need not install bindings from that sub-goal's and-branch at all. Thus, in the CGE above there is no need for a processor to install bindings for sub-goals `b` and `c` because all the variables present in their arguments have to be bound to ground values in order for the condition in the CGE to evaluate to true. The same, however, cannot be said for sub-goal `d`. Thus, a processor does need to install bindings from `d`'s branch, since `d` has a potential conditional variable, `Z`, as its argument (note that work in installing bindings in `d` can be reduced by splitting the binding lists). We believe that the ground-CGEs optimization would greatly improve the performance of the system, since the ground condition would be frequently found in CGEs.

There are many cases for which we can detect at compile-time that all arguments of a predicate would be ground if the condition in the CGE evaluates to true (for example, if the set of variables appearing in arguments of a sub-goal is a subset of set of variables being checked for groundness in the condition, such as in the example above). In general, however, it is not always possible to gather this information at compile time, especially if the condition has occurrences of the boolean operator *or* in it. For example, given the condition `(ground(X, Y) or independent(X, Y))` we cannot conclude at compile time that if this condition evaluates to true, then variables `X` and `Y` would be bound to ground values. The only way to find out if all variables of a sub-goal are grounded is to check their bindings explicitly at run time. This can easily be incorporated in a compiler which generates appropriate code to check for the groundness of the bindings of variables appearing in the sub-goal. The check can be conducted by using the `check_ground` instruction of RAP-WAM [H86a], for instance. At run time if all variables occurring in a sub-goal are found to be ground, this information can be stored in the and-node by setting a bit. When a processor encounters an and-branch whose and-node's bit is set, it doesn't install bindings from binding lists of that branch.

Note again, however, that if a processor task-switches to a node in an and-branch to continue its execution, bindings have to be installed from both the global and local binding list, similar to the case of splitting binding-lists optimization.

We illustrate this optimization with the help of another example program given in appendix A (program A3). For the query shown in program A3, to find out if John is the common ancestor of `Simon` and `Mary` and if yes to find out all the common descendents of `John` and `Beth`.

```
?- X = john, Y = beth,
   (ground(X) | ancestor(simon, X) || ancestor(mary, X) || ancestor(Z, X)),
   ancestor(Z, Y).
```

Suppose a processor wishes to task-switch to the sequential node corresponding to the sub-goal `ancestor(Z, Y)`, then it doesn't have to install conditional bindings from and-branches corresponding to `ancestor(simon, X)` and `ancestor(mary, X)` since all their arguments are ground. Note that the clause for ancestor has conditional variables in it and since it may be recursively called a number of times the number of bindings that might have to be otherwise installed may be quite large, depending upon the depth of recursion in the and-branch. However, bindings will have to be installed from the and-branch clause `ancestor(Z, X)`, since `Z` may be bound within the and-branch.

This optimization can also be viewed as a special case of split-trail optimization where no binding installation/removal is necessary for an and-branch all variables in whose corresponding sub-goal are bound to ground-terms.

5. Conclusion

In this paper we presented several techniques for improving the performance of a combined and-or parallel implementation. Because our combined and-or model is based on an environment representation that incurs a high overhead in task switching (but offers constant-time variable access and task creation), our main focus was on techniques for minimizing the cost of task-switching. The proposed techniques take advantage of and-parallelism and significantly reduce this overhead by reducing the number of conditional variables whose bindings need to be installed during task-switching. The techniques we have presented are quite general and can easily be implemented through WAM like instructions [G90]. An implementation of a system based on an extended WAM and incorporating these optimization is under way. Although our implementation uses the binding arrays method (suitably extended), we showed in this paper that our model for and-or parallelism can also work with other similar environment representation methods, including the version vectors method.

Acknowledgements

Thanks to D.H.D. Warren, Peter Szeredi and Feliks Kluzniak for their insightful comments, and for their suggestions for improving the presentation. The first author is supported by U.K. Science and Engineering Research Council grant GR/F 27420.

A summary of this report appears in the Proceedings of 1990 North American Conference on Logic Programming published by the MIT press.

References

[B84] P. Borgwardt, "Parallel Prolog using stack segments on shared memory multiprocessors," In *Proc. of the 1984 Int'l Symp. on Logic Prog.*," pp. 2-11.

[C87] J. S. Conery, "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors," *1987 IEEE Int'l Symp. in Logic Prog.*, pp. 457-467

[CH86] A. Ciepielewski, B. Hausman, "Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs," In *1986 IEEE Symp. on Logic Prog.*, pp. 246-257.

[D84] D. DeGroot. "Restricted AND-parallelism," *Int'l Conf. on Fifth Generation Computer Systems*, Tokyo, Nov., 1984.

[G90] G. Gupta, "And-Or Parallel Execution of Logic Programs: Abstract Machine Design and Implementation," Ph.D. Thesis, Univ. of N. Carolina at Chapel Hill, In preparation.

[GJ89a] G.Gupta and B. Jayaraman, "A Model for And-Or Parallel Execution of Logic Programs," *Proc. of the 18th Int'l Conf. on Parallel Processing,* 1989.

[GJ89b] G. Gupta and B. Jayaraman, "Compiled And-Or Parallel Execution of Logic Programs," In *Proc. of the North American Conf. in Logic Prog.*, MIT Press, Oct. 1989.

[H86] M. V. Hermenegildo, "An Abstract Machine Based Execution Model for Parallel Execution of Logic Programs," Ph.D. Dissertation, Dept. of Computer Sciences, The University of Texas at Austin, Aug., 1986, TR-86-20.

[H86a] M. V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs". *3rd Int'l Conf. on Logic Prog.*, London, 1986.

[H87] B. Hausman *et. al.*, "Or-Parallel Prolog made efficient on shared memory multiprocessors," in *1987 IEEE Int'l Symp. in Logic Prog.*

[HB88] S. Haridi and P. Brand, "The Andorra Parallel Logic Language," In *Proc. of the Int'l Conf. on Fifth Generation Computer Systems*," Tokyo, 1988.

[LK88] Y-J. Lin and V. Kumar, "AND-parallel execution of Logic Programs on a Shared Memory Multiprocessor : A Summary of Results," in *Fifth Int'l Logic Prog. Conf.,* Seattle, WA, 1988.

[RK89] B. Ramkumar and L. V. Kale, "Compiled Execution of the REDUCE-OR Process Model," In *Proc. of North American Conf. on Logic Prog.*, MIT Press, 1988.

[TD87] H. Touati and A. Despain, "An Empirical Study of the Warren Abstract Machine," In *Proc. of the Symp. on Logic Prog.*, Salt Lake City, 1987, pp. 114-124.

[W83] D. H. D. Warren, "An Abstract Instruction Set for Prolog", Tech. Note 309, SRI International, 1983.

[W84] D. S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategies," In *The 1984 Int'l Symp. on Logic Prog.,* Atlantic City, pp. 198-202.

[W87] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog – Abstract Design and Implementation Issues," *1987 IEEE Int'l Symp. in Logic Prog.*

[W90] D. H. D. Warren, "Extended Andorra Prolog", Internal PEPMA report.

[WR87] H. Westphal, P. Robert, J. Chassin and J. Syre, "The PEPSys Model: Combining Backtracking, AND- and OR-parallelism," In *1987 IEEE Int'l Symp. in Logic Prog.*, San Francisco, pp. 436-448.

Appendix A

```
prime(N,U) :- int_upto(N,U), isprime(N).

isprime(X) :- sqrt(X, Limit),
              indivisible(X,2,Limit).

indivisible(X,S,Limit) :- S > Limit.

indivisible(X,S,Limit) :- S =< Limit,
                          M is X mod S, M =0̄,
                          S1 is S+1,
                          indivisible(X,S1,Limit).

sqrt(N,Root) :- sqrt(N,1,Root).

sqrt(N,Root, Root):- N1 is (Root)*(Root),
                     N1 >= N.
sqrt(N,Try, Root):- N1 is (Try)*(Try),
                    N1 < N,
                    TryNew is Try+1,
                    sqrt(N,TryNew, Root).

int_upto(X,Upto) :- int_upto(X, 1, Upto).

int_upto(X, X, Upto) :- X =< Upto.
int_upto(X, Y, Upto) :- Z is Y + 1, Z =< Upto, int_upto( X, Z, Upto ).

% fib(N,Upto)

fib(X,U) :- fib(0,1,X,U).
fib(X,Y,X,U).
fib(X,Y,F,U) :- Z is X+Y, Y =< U, fib(Y,Z,F,U).

% perfect numbers

perfect(N,Upto) :- int_upto(N,Upto), divisors_sum(N).

divisors_sum(N) :- sqrt(N, Limit), divisors_sum(N,2,1, Limit).

divisors_sum(N,Next,N, Limit) :- Next > Limit.

divisors_sum(N,Limit,PartSum, Limit) :- N is Limit * Limit,
                                        N is PartSum + Limit.

divisors_sum(N, Limit, N, Limit) :- N < Limit * Limit.

divisors_sum(N,Next,PartSum, Limit) :- Next < Limit,
                                       0 is N mod Next,
                                       M is N // Next,
                                       Next1 is Next+1,
                                       NewPartSum is PartSum + Next + M,
                                       NewPartSum =< N,
                                       divisors_sum(N,Next1,NewPartSum, Limit).

divisors_sum(N,Next,Partsum, Limit) :- Next < Limit,
                                       M is N mod Next, M =0̄,
                                       Next1 is Next+1,
                                       divisors_sum(N,Next1,Partsum, Limit).

fibper(Fib, Perfect, Prime) :-
    (ground(Fib, Perfect) ∧ independent(Fibo, Perfecto) | fib(Fibo,Fib) || perfect(Perfecto, Perfect)),
    Prime is Fibo+Perfecto,
    isprime(Prime).
```

Program A1

```
fact(X, U) :- fact(1, 1, X, U).
fact(X, C, X, U).
fact(X, C, Y, U) :- C1 is C+1, Z is X*C1, Z < U, fact(Z, C1, Y, U).

  /*fib, perfect and prime same as in program A1*/
```

```
?- (ground(Fib, Perfect, Fact) ∧ independent(Fibo, Perfecto, Facto),
        fib(Fibo,Fib), perfect(Perfecto, Perfect), fact(Facto, Fact),
            Prime is Fibo+Perfecto+Facto, isprime(Prime).
```

Program A2

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).

parent(john, jack).     parent(john, phil).     parent(jill, simon).
parent(jack, simon).    parent(phil, mary).     parent(beth, oprah).
parent(beth, steve).    parent(oprah, mary).    parent(mary, clare).
parent(simon, david).   parent(simon, pat).     parent(mary, sue).

?- X = john, Y = beth,
   (ground(X), ancestor(simon, X) || ancestor(mary, X) || ancestor(Z, X)),
   ancestor(Z, Y).
```

Program A3