CIS-TR-85-02

AND Parallelism and Nondeterministic in Logic Programs

John S. Conery
Department of Computer and Information Science
University of Oregon

Dennis F. Kibler
Department of Information and Computer Science
University of California, Irvine

AND Parallelism and Nondeterminism in Logic Programs

John S. Conery

University of Oregon

Dennis F. Kibler

University of California, Irvine

ABSTRACT

This paper defines an abstract interpreter for logic programs based on a system of asynchronous, independent processors which communicate only by passing messages. Each logic program is automatically partitioned and its pieces distributed to available processors. This approach permits two distinct forms of parallelism. OR parallelism arises from evaluating non-deterministic choices simultaneously. AND parallelism arises when a computation involves independent, but necessary, subcomputations. Algorithms like quicksort, which follow a divide and conquer approach, usually exhibit this form of parallelism. These two forms of parallelism are conjoinly achieved by the parallel interpreter.

1.0 Introduction

In 1980, we started our investigation into the possibility of using a logic programming language for programming highly parallel, dataflow style multiprocessors. At that time, it was not at all clear that there was any parallelism in logic programs beyond the "obvious" parallel tree search[1]. Our initial goals were to:

▹ Identify potential sources of parallelism in logic programs.

▹ Define the parallelism in terms of an abstract interpreter which would be independent of any particular hardware organization (*i.e.* do not base the interpreter accept only deterministic programs, which we feel is too restrictive. Nondeterminism is a major advantage of logic programming. Our interpreter maintains a correct interpretation of nondeterminism while performing both AND and OR parallelism. Moreover the method for achieving AND parallelism has some of the properties of *intelligent backtracking* so new processes are not spawned promiscuously.

We begin this paper by defining several different ways of achieving parallelism in logic programs. Then we briefly describe our basic AND/OR process model. The next four sections develop our mechanism for achieving AND parallelism, giving specific algorithms and underlying data structures. We use the map coloring example of Pereira and Porto and matrix multiplication to illustrate simple AND parallelism. In the last sections we discuss the difficult case of backtracking while in the midst of AND parallelism. Again we use the map coloring example to illustrate how the various data structures are updated. Finally we summarize our contribution and suggest some future work.

2.0 Sources of Parallelism

An interpreter of logic programs starts with a goal statement such as

$$\leftarrow \mathtt{p(X)} \wedge \mathtt{q(X)} \wedge \mathtt{r(Y)}.$$

and tries to derive the null clause through a series of resolutions. The interpreter must *solve* each subgoal in the goal statement, by finding a clause in the program with a head that matches the subgoal, and then forming a new goal statement from the remaining subgoals and the body of the selected clause. When there are many possible clauses in the program with heads that match a subgoal, a tree of derived goal statements is defined. A typical (sequential) implementation, such as a standard Prolog interpreter, performs a depth first search of this tree. The initial subgoal. Here the results of the parallel search are passed to a uniprocessor inference mechanism This form of parallelism will be useful for distributed database types of applications. Examples of search-parallel systems include the work of Warren[6)], Taylor[7)], and Minker[8)].

OR parallelism is exploited when a system can perform parallel computations based on the fact that a subgoal matches the heads of more than one clause. The parallel tree search is one form of OR parallelism; the interpreters of Haridi[9)] and Furukawa[10)] are also based on this form of parallelism. The mechanism for OR parallelism of the AND/OR process model is slightly different, as will be explained in the next section.

AND parallelism arises when two or more subgoals from a single goal statement are solved in parallel. This can be quite complicated. Referring to the example above, `p(X)` and `q(X)` have a variable in common, and the system must ensure that the processes that solve these subgoals bind `X` to the same value.

Stream parallelism is sometimes identified with AND parallelism, as described by Kowalski[11)], Clark[12)], and Shapiro[13)]. This comes from viewing the processes that solve subgoals as coroutines that communicate via shared variables. Referring back to the example clause, subgoals `p(X)` and `q(X)` could be solved simultaneously by viewing the shared variable `X` as a stream. One of the subgoals will be the producer of values for the stream, the other will be the consumer. Execution of producers and consumers proceeds simultaneously on different processors. The interpreters of Clark and Gregory[14)] and Shapiro[13)] are implementations of what we call stream parallelism.

The important distinction between our implementation of AND parallelism and stream parallelism is that current implementations of stream parallelism restrict their inputs to deterministic programs. On the other hand, our implementation of problem.

▷ **cancel**: This message is sent by a parent to a descendant when the parent knows it does not need any more answers from that descendant.

An AND process is created to solve a goal statement, such as the one given in the previous example. It solves the goal statement by creating descendant OR processes for each subgoal, and waiting for success messages from each. AND parallelism arises when an AND process is able to create more than one such descendant at any one time.

The subproblem for an OR process is always exactly one subgoal. This goal comes from its parent's goal statement. The OR process can solve this subgoal in one of two ways: the subgoal may match the head of a unit clause, in which case the OR process immediately sends a success message, or the subgoal may match the head of a nonunit clause, in which case the OR process creates a descendant AND process to solve the body of the matching clause. OR parallelism is exploited when more than one AND descendant is active at any time. In alternative approaches to OR parallelism, a process is given the entire context of the problem solving state. If successful, such a sprouted process reports back to the top level goal. In our approach an OR process receives only a single goal and reports back to its immediate parent. This reduces the size of messages and set-up time for OR processes. For a fuller description of comparison of this form of OR parallelism with the goal tree search described in section 2, see Conery's thesis[4)].

4.0 Parallel AND Processes

A *sequential* AND process solves its subgoals one at a time; it can mimic a sequential interpreter by solving the subgoals from left to right. The transmission of redo messages to previously solved goals is equivalent to backtracking. A *parallel*

A second argument against solving all subgoals at once is that by waiting until some of the variables of a subgoal `G` are bound via the solution of other subgoals, the OR process created to solve `G` may be more efficient: there are often fewer solutions, and fewer fruitless choices made in constructing those solutions[9].

Finally, and of most practical importance, some subgoals *fail* if an attempt is made to solve them before a sufficient set of variables are instantiated; these are the subgoals with *mode declarations*[18]. For example, in the goal statement

$$\leftarrow \texttt{length(L,N)} \land \texttt{X is 5 * N.}$$

the goal of multiplying `N` by five fails unless `N` is instantiated to an integer in the solution of the first subgoal. In some systems, `X is 5 * N` is written `sum(X,5,N)`, which is not solvable until any two of the three arguments are bound to integers. This subgoal named `sum` is said to have a *threshold* of two[15].

An effective method for achieving AND parallelism is thus a problem of correctly ordering the subgoals, of deciding which subgoals must be solved sequentially and which can be solved in parallel. The implementation of AND parallelism defined in Conery's thesis[4] has three major components. There is an *ordering algorithm* that automatically decides the order in which the subgoals should be solved, a *forward execution* component which creates the descendant OR processes in the proper order, and a *backward execution* component to handle fail and redo messages and decide which subgoal(s) must be re-solved before resuming forward execution.

5.0 Ordering of Subgoals

The basis for the ordering of subgoals in the body of a clause is the sharing of variables. Whenever two or more subgoals have a variable in common, one of the subgoals will be designated the *generator* for that variable, and it will be solved before the others, which are now *consumers*. The solution of the generator to be a subgoal that is a generator for one of the variables in G. A predecessor, in general, is either an immediate predecessor or a predecessor of an immediate predecessor.

The head of the clause is a special case. It is considered to be the generator of every variable that is bound when the AND process is created. The ordering algorithm of the next section requires this information, so the head of the clause (*i.e.* the subgoal being solved by the parent OR process) is included in the state information of a parallel AND process The head is also the consumer of variables that are not instantiated when the AND process is created. Where it is important, the head will be included in the pictures of dataflow graphs in later sections.

7.0 The Ordering Algorithm

The ordering algorithm must designate a generator for every variable in the goal statement. It does this automatically, without relying on control annotations supplied by the programmer. Logically, any subgoal can be the generator of any variable that appears as one of its arguments. The only exceptions are determined by evaluable predicates or user defined procedures with mode declarations. The ordering algorithm is used primarily to ensure that mode constraints are not violated, and secondarily to produce an efficient ordering. Another constraint on the form of the graph is that the graph must be acyclic; the reasons for this will be obvious when the forward execution algorithm is explained in the next section.

There are a number of rules one can use to identify generators. The first, mentioned above, is that the head of a clause is the generator for all variables that are instantiated when the clause is invoked.

Second, some of the subgoals in the body may have I/O modes. These subgoals may be evaluable predicates, for which the system already knows the modes, or

The only heuristic currently implemented in the ordering algorithm is the *connection rule*. Briefly, when the connection rule is applied in order to find a generator for a variable V, it looks for a subgoal that (1) contains V as an unbound variable, and (2) consumes variables for which generators are already known. The connection rule is stated more concisely as step 3.a of the ordering algorithm Figure 1. The basis for using this rule is that, in general, a subgoal will have fewer possible solutions, and thus be more easily solved, when some of its variables are bound[9)].

Figure 1. Subgoal Ordering Algorithm

Finally, if a variable is not assigned a generator through application of the previous three rules, the leftmost subgoal in which the variable occurs is designated as the generator for that variable. This rule, called the "leftmost rule," is simply a default, and is included to make sure that every variable will have a generator. It is most often invoked when there are no bound variables in the head. In these cases, the leftmost rule is applied once, and then the connection rule finds generators for the remaining variables.

Note that since mode declarations are known before a clause is called, the second rule can be applied when clauses are compiled or first loaded by the interpreter. The other rules are applied at runtime, once the pattern of variable instantiation in the clause is known. As an optimization, our current interpreter keeps dataflow graphs for clauses in a cache, indexed by the pattern of variable instantiation in the head. Whenever a new AND process is created, this cache is searched to see if an existing graph can be used. This optimization is quite effective, especially when recursive clauses are being interpreted.

8.0 Examples of Subgoal Orderings

The ordering algorithm will be illustrated by four examples. The dataflow example of where the connection rule implements the optimal ordering, in the sense that the solution of either **author(P,X)** or **loc(X,I,D)** will be more efficient if **X** is bound.

Deterministic Function: Figure 2c

– Insert Figure 2c here –

The following clause illustrates the general form of a deterministic function, in this case an implementation of a divide and conquer algorithm.

```
solve(P,Q) ←
    divide(P,P1,P2) ∧
    solve(P1,Q1) ∧
    solve(P2,Q2) ∧
    combine(Q1,Q2,Q).
```

On every call to **solve**, **P** will be bound to a term representing the input problem. As a result, **Q** will be bound to a term representing the output of the function. The optimal ordering of subgoals is: divide problem **P** into independent subproblems **P1** and **P2**; then solve **P1** and **P2** in parallel via the recursive calls, thereby instantiating **Q1** and **Q2**; when both are done, construct answer **Q** from partial answers **Q1** and **Q2**. This sequence of events is implied by the picture in Figure 2c; exactly how it is achieved is described in the next section, on forward execution. This graph can be produced by repeated application of the connection rule, so mode declarations are not required. In general, however, mode declarations will be required when ordering the subgoals in the body of a clause that implements a function.

Map Coloring: Figure 2d

The goal of the following procedure is to see if there is an assignment of one of four colors to the regions of a map* such that no two adjacent regions have the same color:

* The map to be colored by this clause is shown in Figure 6, along with procedure code and other information used by the interpreter.

states: *blocked*, *pending*, or *solved*. A subgoal is in the blocked state when an OR process has not yet been created for it. A subgoal is in the pending state when an OR process has been created for it, but the process has not yet sent back either a success or fail message. Finally, a subgoal is in the solved state after the OR process that was created for it has sent back a success message.

Forward execution is essentially a graph reduction procedure. Whenever the AND process receives a success message from a descendant, it means the corresponding subgoal can be resolved away from the body of the clause; in the dataflow graph, the node for the subgoal and all arcs leaving it are removed from the graph. The AND process succeeds after a success message has been received from every descendant, *i.e.* after the graph has been completely reduced. Recall that a success message from an OR process created to solve a subgoal `G` has the general form `success(G`θ`)`, where `G`θ is a copy of `G` with (possibly) some variables bound. The graph reduction step is accomplished by resolving `L` with the current set of subgoals in the body of the clause. If `G` is a generator of a set of variables, then some of those variables may be instantiated in `G`θ. Envision values flowing from `G` to the consumers, as the resolution of `G`θ with the remaining subgoals causes those variables to be bound in the resolvent.

The criterion for deciding when to start an OR process for a subgoal is that a subgoal is ready to be solved only when all of its predecessors have been solved, *i.e.* when the corresponding node in the dataflow graph has no incoming arcs. If the graph is acyclic, and each subgoal can be solved, then eventually a process will be started for every subgoal. A more formal presentation of the forward execution algorithm is given in Figure 3. Figure 4 shows the parallel solution of two sample goal lists as sequences of graph reductions.

Insert – Figure 3. Forward Execution Algorithm – here

Insert – Figure 4. Graph Reductions – here

Figure 3 shows that the ordering algorithm will be applied after every success message is received. This is necessary for those cases when a generator binds its variable `V` to a non-ground term containing a new variable `V'`. If there is more than function shown here is sequential in nature, since the results of the multiplications are be summed serially.

Analysis of the bodies of mmt and mmc shows that since the recursive call can be done at the same time as the call to the lower level function, the time required to solve a problem of size n is proportional to the time required to solve the largest subproblem, rather than proportional to the sum of times to solve both subproblems. The time required to compute the product of the two matrices is thus the time required to distribute the last of the row/column pairs to the process that performs an inner product, plus the time required to do that inner product. For the multiplication of $n \times n$ arrays, this time is $O(n+n+n)$, *i.e.* $O(n)$[5].

The table in Figure 5 shows the results of some simulations of matrix multiplication, giving number of steps required, and the simulated time used. The results support the claim that parallelism in deterministic functions can be exploited by the AND parallelism of the AND/OR Process Model.

10.0 Backward Execution

The purpose of backward execution is to coordinate the actions of the generators in their production of terms for the variables of the goal list. If there are n variables in its goal list, an AND process is expected to construct as many n-tuples of terms as possible. A subset of these n-tuples belong to the relation defined by the clause the AND process is interpreting.

A straightforward model for generating tuples is provided by the nested loops of a procedural language, such as Pascal. For example, a nested loop implementation of the previously defined map coloring problem is of the form:

the domains of the generated variables are finite), and inherits the same efficiency fo the Prolog version of the nested loop model.

11.0 Data Structures for Backward Execution

Adoption of the nested loop model for constructing tuples of terms in a parallel AND process requires a *linear ordering* of subgoals and implementation of a *reset* operation. These will be defined in this section; the actual sequence of events carried out in backward execution will be described in the next section. Examples will refer to the clause and dataflow graph of Figure 6.

Insert – Figure 6.0 Map Coloring Problem – here

Many of the data structures require a means for identifying a particular subgoal in the body of a clause. The technique used is to refer to a subgoal by a term of the form `#N`, where `N` is the place the subgoal occupies in the text of the clause. With respect to the example of Figure 6, the term **`#2`** refers to **`next(C,D)`**, the second subgoal in the body.

Recall that in the nested loop model, after a generator `G` has assigned its last value to a variable, the previous (next outer) generator assigns a new value, and `G` is reset. In a parallel AND process, we cannot use textual ordering to decide the relative position (inner vs. outer) of generators. The solution is to define a linear ordering of subgoals, independent of the textual ordering in the original program, and use the linear ordering to decide when to reset a generator with respect to other generators.

For reasons that will be explained later, the linear ordering is actually an ordering of all subgoals, not just the generators. The only constraint on the relative position of any two subgoals in the linear ordering is that a generator must always come before all subgoals that consume its variable. In the current implementation, the linear ordering is obtained via a breadth first traversal of the dataflow graph. The linear ordering of the subgoals of Figure 6 is

$$[\#1, \#3, \#4, \#6, \#5, \#2, \#7, \#8]$$

loop model. If a failed subgoal does not consume a variable V, there is no sense in obtaining a new value of V via a redo to the generator of V; instead, obtain a new value for a variable consumed by G.

Associated with each subgoal is a *redo list*. The redo list determines which generators will be sent redo messages and in what order redo messages will be sent. The redo list for a subgoal G contains G and every predecessor of G, sorted according to the linear order (with subgoals that are earlier in the linear order occurring later in the redo list). Redo lists are created at the same time the linear ordering is made. Redo lists for the subgoals of the example problem are shown in Figure 6.

Finally, an AND process maintains a structure called the *failure context* to keep track of the failed subgoals and decide exactly which generator should be sent the next redo message. The failure context is initially the empty list, and as fail messages are received, subgoal numbers are added to this list.

12.0 Processing of Backward Execution

An overview of backward execution is that when a fail message is received, the backward execution algorithm is called to trace out a path in the dataflow graph that extends back from the failed subgoal toward the root of the graph. The failure context reflects the current state of this path. When a generator is encountered along this path, it is sent a redo message. This path should eventually include every predecessor of the failed subgoal, if required. If the failure context ever grows to extend beyond a subgoal with no predecessors, or to include the head of the clause, then the AND process fails.

The desired backward path is simply the redo list for the failed subgoal. This list contains every predecessor of the subgoal, *i.e.* it contains every generator that could possibly affect the set of values consumed by the subgoal. An AND process will always be able to determine which generator to re-solve when it first receives a fail message, since all it has to do is send a redo message to one of the immediate predecessors of the failed subgoal. However, once the backward execution algorithm has embarked on a backward path, subsequent failures from subgoals not on this path can cause difficulties. This is known as a *multiple failure*; rules for handling to be canceled. If a subgoal consumes a variable that is generated by any generator that was either sent a redo or reset, then that subgoal must be canceled, since it consumes values that are being changed. The processes for these subgoals will be replaced when all of their predecessors are once again in the solved state. Note that a subgoal might have a new process created immediately, in the case that all of its generators were simply reset.

The processing of success messages has to be modified slightly, in order to accommodate the failure context. When a success message is received, and the forward execution algorithm starts a set of new processes, the subgoals corresponding to the new processes must be removed from the failure context list. Thus the failure context list grows and shrinks as generators are sent redo messages and then respond with additional answers. The failure context shrinks all the way back to the empty list when a new process is started for the subgoal that originally failed. A more concise presentation of the backward execution algorithm is in Figure 7.

Insert – Figure 7. Backward Execution Algorithm – here

13.0 Map Coloring Example

The complete parallel solution of the map coloring problem is presented in this section. This very complicated example has three parts. First, we show the forward execution phase as the first tuple of colors was created. Next, we show what happened during backward execution. In the current implementation of the parallel interpreter, the final solution was obtained very quickly, with few redo messages to processes for generators. The third part of the example shows what might happen with a slightly different sequence of events, when a multiple failure would occur. In this example we again use **#N** to denote sub-goals and **↑N** to denote the OR process created to solve subgoal **N**.

When the AND process was first created, and after the subgoals were ordered, the only subgoal for which an OR process could be started was **#1:next(A,B)**. Eventually, this sent back **success(next(red,blue))**, binding **A** to **red** and **B** to **blue**. Next, processes for the three generators in the middle row of the graph were started. All three succeeded, and as the success messages arrived, the following one value really has no effect, and the replacement of a process such as that for subgoal #8 can be avoided when its variables do not change values. The state of the AND process after this transition: subgoals #1, #4, and #6 solved; subgoals #3 and #8 pending; subgoals #2, #5, and #7 blocked; failure context [#5].

The fail message from the original process for subgoal #2 then arrived. Since that process had been canceled, this message was ignored. The successes from the original processes for #7 and #8 arrived, and they also were ignored. Note that even though there is a process for #8 at this time, it has a different ID than the original process. The AND process always ignores messages from processes it has canceled.

The success from #3 arrived, with `next(red,yellow)`, binding `C` to `yellow`. New processes for #2, now `next(yellow,blue)`, and #5, now `next(blue,yellow)`, and #7, now `next(yellow,red)`, were created. Since there is a new process for #5, it was removed from the failure context. The state of the AND process: subgoals #1,3, #4, and #6 solved; subgoals #2, #5, #7, and #8 pending; failure context [].

Finally, all of the pending processes sent success messages; the order is irrelevant. In particular, note that ↑8 could have sent its success message before the success from ↑3 in the previous paragraph. After the last was received, the AND process sent its parent the message

success(color(red, blue, yellow, blue, red))

Case 2: #2 fails first.

When the state of the AND process had subgoals #1, #3, #4, and #6 solved, with the remaining subgoals pending and an empty failure context, two fail messages were on the way. The next sequence of transitions shows what would have happened had the failure from ↑2 arrived first. This sequence involves multiple failures.

The fail message from ↑2 arrives, the failure context is set to [#2], which is the prefix of the redo list [#2,#4,#3,#1]. ↑4 is sent a redo message, and then the remaining subgoals in the linear ordering are handled as follows:

▹ #6: Reset.

▹ #2: Already failed.

▷ #8: Canceled (since D, E reset), replaced by new process for original D and E.

The state of the AND process is now: subgoals #1, #4, and #6 solved; subgoals #3 and #8 pending; subgoals #2, #5, and #7 blocked; failure context [#5]. The current values of the variables are A = red, B = blue, C unbound, D = blue, and E = red. Note that this is the same state as earlier (in case 1), when #5 and #2 were failures and the fail message from #5 arrived first.

14.0 Conclusion

In this paper we have shown an effective procedure for translating logic programs into a collection of communicating processes. Futhermore we have defined an abstract interpreter for these processes which permits both AND and OR parallelism. Neither technique for achieving parallelism makes any additional assumptions about the logic program and both types of parallelism operate conjoinly on the same logic program. The details of achieving AND parallelism with semi-intelligent backtracking are complicated and rest on ideas about goal ordering and dataflow. All of the essential algorithms and data structures have been presented in this paper.

Parallel solution of the body of a clause is essentially an attempt to create a dataflow graph from the body, and then solve the subgoals in the order specified by the graph. This attempt is successful when the subgoals all succeed, which is often the case when the clause implements a deterministic function. However, in nondeterministic functions and relations, it is not always the case that subgoals can be solved on the first attempt. When a subgoal fails, an interpreter must re-solve a previously solved subgoal, hoping the next solution creates new variable bindings that allow the failed subgoal to be solved.

Backward execution is the name of the mechanism in parallel AND processes that determines which subgoals must be re-solved in response to failures. The mechanism is quite complicated, and requires a large overhead in terms of data structures to represent the state of each subgoal and the state of the process as a whole. Fortunately, the overhead does not interfere with forward execution; it is only when subgoals fail that the rather awkward backward execution mechanism is invoked.

Prolog Programs on a Broadcast Network. In *1984 International Symposium on Logic Programming*, pp. 12-21, 1984.

7) Taylor, S., A. Lowry, G. Q. Maguire, Jr., and S. J. Stolfo. Logic Programming Using Parallel Associative Operations. In *1984 International Symposium on Logic Programming*, pp. 58-68, 1984.

8) Eisinger, N., S. Kasif, and J. Minker. Logic Programming: A Parallel Approach. *Proceedings of the First International Logic Programming Conference*, pp. 1-8. Faculté des Sciences de Luminy, Marseille, Sept, 1982.

9) Ciepielewski, A. and S. Haridi. Formal Models for OR-Parallel Execution of Logic Programs. CSALAB Working Paper 821121, Royal Institute of Technology, Stockholm, Sweden, 1982.

10) Furukawa, K., K. Nitta, and Y. Matsumoto. Prolog Interpreter Based on Concurrent Programming. *Proceedings of the First International Logic Programming Conference*, pp. 38-44. Faculté des Sciences de Luminy, Marseille, Sept, 1982.

11) Kowalski, R. A. Predicate Logic as a Programming Language. *IFIPS 74.*

12) Clark, K. L., and F. McCabe. The Control Facilities of IC-Prolog. In D. Michie, Ed., *Expert Systems in the Microelectronic Age*, Edinburgh University Press, 1979.

13) Shapiro, E. Y. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, Institute for New Generation Technology, Tokyo, Japan, 1983.

14) Clark, K. L. and S. Gregory. A Relational Language for Parallel Programming. *Proceedings of the Conference on Functional Programming 1Languages and Computer Architecture*, pp. 171-178. ACM, October, 1981.

15) Zara, R. V. A Semantic Model for a Language Processor. *Proceedings of the A.C.M. National Meeting*, pp. 323-339, 1967.

16) Hewitt, C. and G. Attardi. Act I for Parallel Problem Solving. Technical Report, MIT.

The ordering algorithm uses the following variables:

B The set of literals in the body. Initialized to contain every literal in the body; when a literal is designated as a generator, it is removed from B.

S The set of variables for which generators have been specified.

U The set of variables for which generators are not known yet. Note that the union of S and U contains every variable in the clause.

The algorithm:

1. Identify as many generators as possible using mode declarations; remove those literals from B, and initialize S to be the variables generated by these literals.

2. Add to S the variables instantiated in the head of the clause; the head is the generator of these variables. Initialize U to be the set of all remaining variables.

3. Repeat until $B = []$ or $U = []$:

   a. Make a set LS with every literal in B that has at least one variable in U and one variable in S {note: if S is empty, LS will also be empty}.

   b. If LS is empty after step (a), find the leftmost literal in B that has a variable in U and add it to LS {LS contains just this one literal}.

   c. For every literal L in LS, assign L as the generator of any variables in U that occur in L. Remove these variables from U and add them to S. Remove L from B.

Figure 1. The Literal Ordering Algorithm

Clause:

$$query(P, I) \leftarrow author(P, X)\ \&\ loc(X, I, D).$$

Call:

$$\leftarrow query(P, uci).$$

$$\begin{tikzcd}
 & query(P, I) \arrow[dr, "I"] & \\
author(P, X) \arrow[ur, "P"] & & loc(X, I, D) \arrow[ll, "X"]
\end{tikzcd}$$

Figure 2b.

$color(A,B,C,D,E) \leftarrow next(A,B) \,\&\, next(C,D) \,\&\, next(A,C) \,\&\, next(A,D) \,\&$
$next(B,C) \,\&\, next(B,E) \,\&\, next(C,E) \,\&\, next(D,E)$

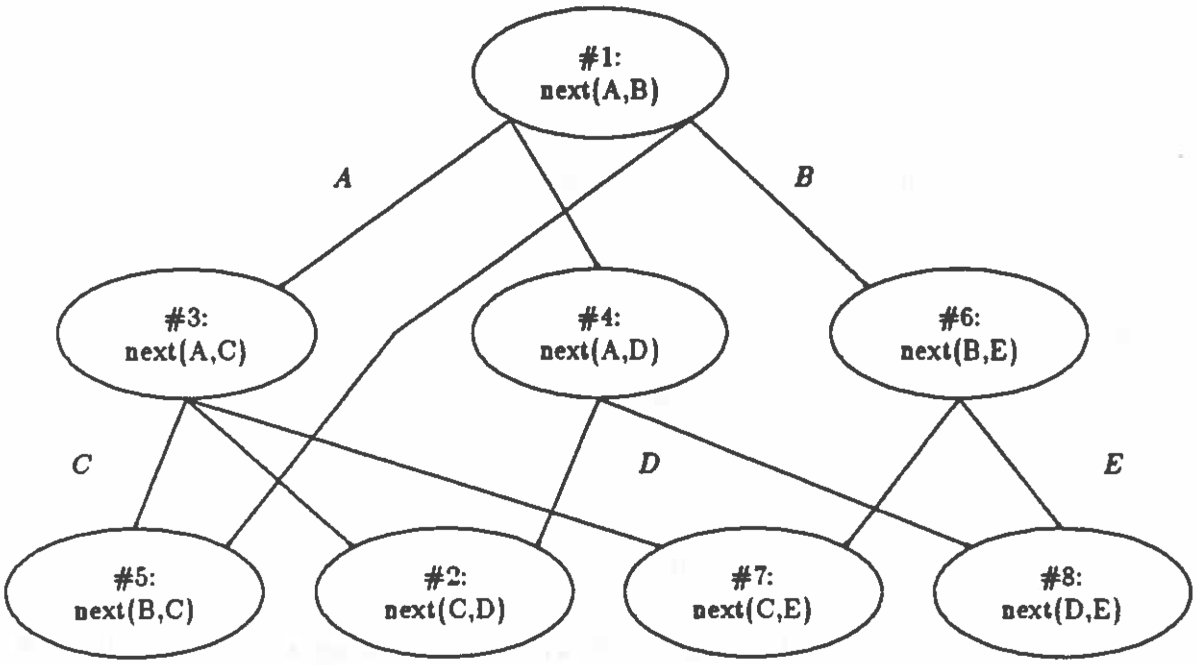
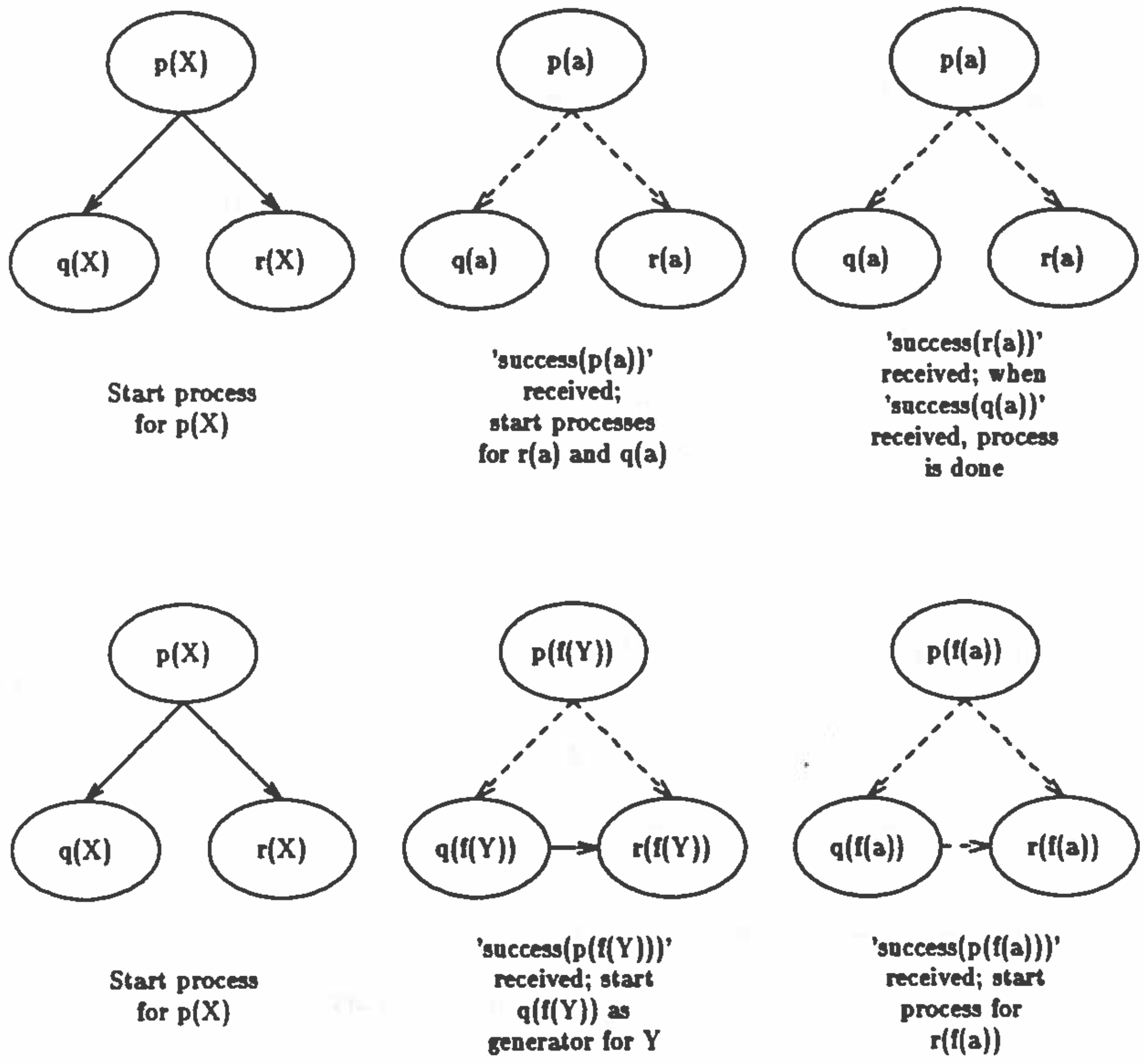


Figure 2d.


p(X)
q(X)
r(X)
Start process for p(X)
p(a)
q(a)
r(a)
'success(p(a))' received; start processes for r(a) and q(a)
p(a)
q(a)
r(a)
'success(r(a))' received; when 'success(q(a))' received, process is done
p(X)
q(X)
r(X)
Start process for p(X)
p(f(Y))
q(f(Y))
r(f(Y))
'success(p(f(Y)))' received; start q(f(Y)) as generator for Y
p(f(a))
q(f(a))
r(f(a))
'success(p(f(a)))' received; start process for r(f(a))

This figure shows two possible sequences of graph reductions during forward execution for the goal statement

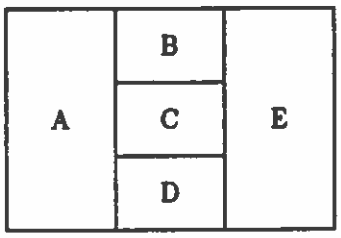
$\leftarrow\ p(X)\ \&\ q(X)\ \&\ r(X).$

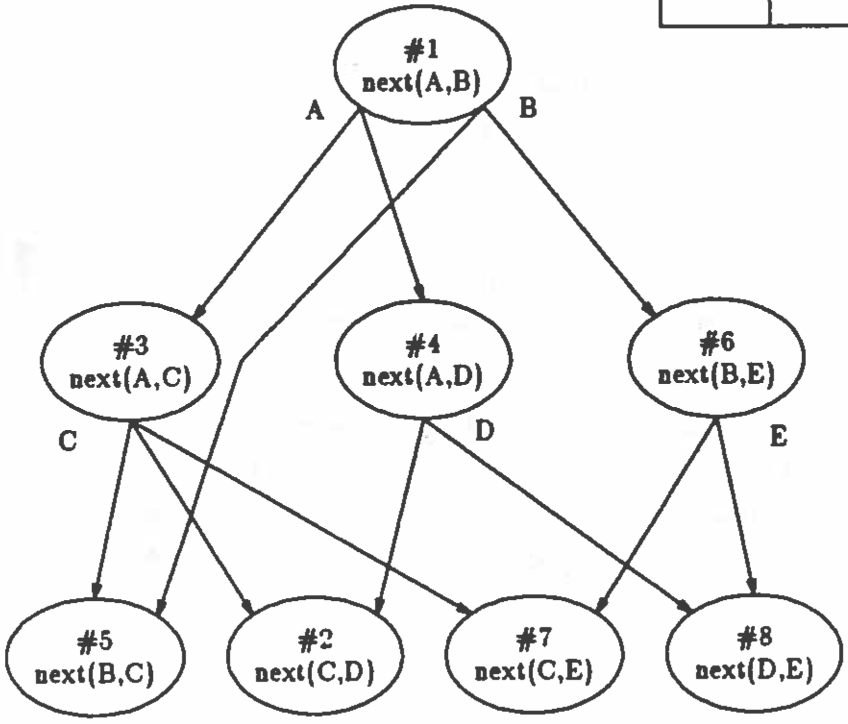
Nodes and arcs drawn with dotted lines have been removed. In the first sequence, $p(X)$ generates the ground term a, and $r(a)$ and $q(a)$ can be solved in parallel. In the second sequence, X is bound to $f(Y)$, making the remaining literals $r(f(Y))$ and $q(f(Y))$. The literal ordering algorithm must be called to decide on a generator for Y; then that generator will be solved before the other literal.

Figure 4: Sequences of Graph Reductions

color(A, B, C, D, E) ← *next(A, B)* & *next(C, D)* & *next(A, C)* & *next(A, D)* &
next(B, C) & *next(B, E)* & *next(C, E)* & *next(D, E)*.

next(red, blue) ← .	*next(red, yellow)* ← .	*next(red, green)* ← .
next(blue, red) ← .	*next(blue, yellow)* ← .	*next(blue, green)* ← .
next(yellow, red) ← .	*next(yellow, blue)* ← .	*next(yellow, green)* ← .
next(green, red) ← .	*next(green, blue)* ← .	*next(green, yellow)* ← .





A map with five regions, the coloring problem as a list of eight borders, and the dataflow graph created by the ordering algorithm. This method of solving a map coloring problem in logic was originally used by Pereira and Porto [46] to illustrate intelligent backtracking.

Figure 6. The Map Coloring Problem