Backward Execution Based on Dynamic Data Dependencies*

David M. Meyer
University of Oregon
Eugene, Oregon

CIS-TR-89-10
May 18, 1989

Abstract

A backward execution algorithm for nondeterministic AND-Parallel logic programs is described. The algorithm extends the class of semi-intelligent backward execution algorithms allowing them to efficiently execute programs with dynamic data dependencies. This is accomplished by generalizing the data dependency graph from an instance of data dependency for a clause to a static description of the set of possible data dependencies. This static description makes the algorithm suitable for compilation to a simple abstract instruction set.

Department of Computer and Information Science
University of Oregon

*Supported by NSF Grant CCR-8707177

1 Introduction

In recent years, much attention has been paid to the implementation of AND parallelism in logic programs. A major problem one faces in such an implementation is the management of nondeterminism. Nondeterminism refers to the fact that, for a given logic program, there may be more than one way to satisfy a goal statement. In sequential systems (*e.g.* Prolog), nondeterminism is implemented by simple chronological backtracking: if the currently executing goal fails, the implementation backtracks to the most recent goal with an untried alternative. This method is unsatisfactory in an AND parallel setting, however, where the most recently solved goal may not be the correct backtrack point.

There are several related methods for exploiting nondeterminism in independent AND parallel systems. Called *semi-intelligent* backtracking algorithms, these schemes rely on a combination of static and dynamic information about producer/consumer relationships within a clause to determine how to retry a previously solved goal after a failure[1,2,3,6,12]. The basic task of these algorithms is to coordinate *retry* and *reset* operations so that a consumer literal [1] sees all possible combinations of bindings from its predecessors.

The semi-intelligent algorithms can be broadly classified according to their treatment of the data dependencies that arise among the literals in a clause body. The first class places emphasis on compile time construction of a fixed set of clause level data dependencies [1]. These fixed data dependencies are used to compute the backtrack points for a given failure. The purely static character of these algorithms requires that backtrack points be chosen conservatively: a backtrack point must be correct for any invocation of the clause in which the failed literal resides. As a result, these algorithms are typically unable to adapt to favorable run time conditions, and may select a goal which cannot cure the current failure.

The second class of semi-intelligent algorithms also use fixed data dependencies. The distinguishing characteristic of these algorithms is that they maintain the dynamic failure history of the literals in a clause body, which typically results in better precision than is achieved by their static counterparts [3,6,12]. However, these algorithms also assume that the fixed data dependencies are exact, and require dynamic reconstruction of these dependencies if and when they change. This requirement limits the efficiency these algorithms can achieve in executing programs without fixed data dependencies, since the cost of dynamic data dependency reconstruction can easily erase the run time advantages of increased precision.

In this paper, we present a new type of data dependency graph, called a *generator inheritance graph*, which allows us to implement semi-intelligent algorithms in the sec-

[1] We use the terms *literal* and *goal* interchangeably.

ond class using a static representation of the data dependencies. Static representation of these data dependencies is crucial, since it allows us to design simple and efficient abstract machine implementations. The rest of this paper is organized as follows: Section 2 reviews the general problem faced by the semi-intelligent algorithms. Section 3 describes the details of a basic backward execution algorithm, and section 4 describes generator inheritance graphs, and an extension to the algorithm described in section 3 which uses these graphs.

2 The Semi-Intelligent Backtracking Problem

The main problem confronting the semi-intelligent backtracking algorithms can be stated as follows. Let L_f be a literal that has failed, let $B_{L_f} = \{L_i, \ldots, L_j\}$ represent the set of semi-intelligent backtrack points for L_f, and let $L_b \in B_{L_f}$ be the backtrack literal selected after L_f fails. The basic problem is to remember the backtrack points that were considered when L_f failed, so that they may be reconsidered if L_b later fails. This failure history, denoted H_{L_f}, is simply the set of unused backtrack points $B_{L_f} \backslash \{L_b\}$. Semi-intelligent behavior thus requires that the representation of B_{L_b} be updated with H_{L_f} whenever L_b is selected as the backtrack literal.

The semi-intelligent backtracking algorithms compute B_{L_f} using information derived the data dependencies between the literals in a clause body. These data dependencies are typically represented by a data dependency graph (DDG), constructed at compile time, in which the nodes represent the literals of the clause body and there is a directed edge between one node and another if the first node generates the value of a variable consumed by the second node. The generator/consumer relationship depicted by a DDG frequently reflects a *linear ordering* that has been assigned to the literals in the clause body. The linear order assigns an integer index to each literal such that if literal i generates a variable consumed by literal j, then $i < j$.

The structure of B_{L_f} can be elucidated by considering the ways in which L_f may fail. First, L_f may have immediately rejected the arguments supplied by its predecessors (*i.e.* L_f failed to unify with a the head of a clause in the program). We call this *consumer failure*; consumer failure may be cured by backtracking to one of the predecessors of L_f in B_{L_f}. A second type of failure, called *generator failure*, occurs when all of the bindings generated by L_f are rejected by one or more of L_f's successors. In this case, the successors may be solvable by the next value from another predecessor and one of the values from L_f. Generator failure may be cured by backtracking to an element of B_{L_f} that is not a predecessor of L_f. In the following discussion, we use $B_{CF(L_f)}$ to denote the set of backtrack points that could cure consumer failure and $B_{GF(L_f)}$ to denote the set of backtrack points that could cure generator failure, with $B_{L_f} = B_{CF(L_f)} \bigcup B_{GF(L_f)}$.

If we assume that the data dependency graph G for a clause is fixed, then $B_{GF(L_f)}$ is simply the predecessor set of L_f. L_b is the least element of $B_{GF(L_f)}$, and is known at compile time. However, in the case of generator failure, the set of backtrack points must be computed dynamically. One way to ensure that the appropriate backtrack points are contained in $B_{GF(L_f)}$ when L_f fails is to have the rejecting successor of L_f, say L, update B_{L_f} with H_L when it fails. This is the basis of the algorithms described by Lin, Kumar, and Leung [6] and Woo and Choe [12].

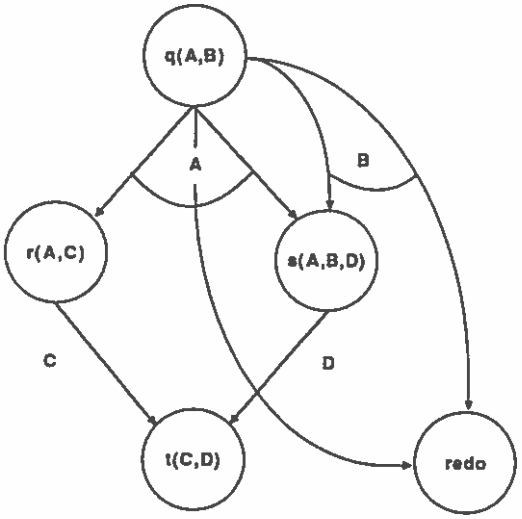
Another approach is to pre-compute the set of *possible* elements of B_{L_f}, called the *candidate* set of L_f, and record only that failure history required to discriminate among the possibilities. In this case, the failure history recorded is a set of failed successors, called *marks*, for each literal. An important aspect of this representation is that for a fixed data dependency graph, the candidate sets are fixed. The candidate set for L_f given data dependency graph G, denoted $candidates_G[L_f]$, is computed as follows. First, define $pred_G[L_f]$ to be the set of predecessors of L_f in G, *i.e.* $\{p \mid$ there exists a path from p to L_f in $G\}$. The successor set of L_f, $succ_G[L_f]$, is defined similarly. Then $candidates_G[L_F] = pred_G[L_f] \bigcup cp_G[L_f] \setminus \{L_f\}$, where $cp_G[L_f] = \bigcup_{x \in succ_G[L_f]} pred_G[x]$. This is the basis of the algorithm described by Conery [3].

The goal of this work is to extend Conery's algorithm to efficiently execute AND parallel logic programs with dynamic data dependencies. To accomplish this, we will define the candidate sets using with a new static data dependency representation. This representation will allow us to avoid dynamic reconstruction of the data dependency graph, while maintaining the fixed data structures and literal selection scheme of the original algorithm. In the following sections we review the original algorithm, and describe an extended algorithm which meets the requirements described above.

3 The CS Algorithm

Conery's backward execution algorithm, which we will call the candidate set (CS) algorithm, proceeds in two phases. The first phase, called the *marking* phase, begins when an AND process receives a fail message from a process corresponding to one of the literals it its body, say literal i. The AND process records the failure history by adding the index i to the marks set of each of literal i's predecessors. The next phase, called the *search* phase, selects the appropriate backtrack literal by searching for the candidate literal latest in the linear ordering (*i.e.* furthest to the right in the clause body) which has i or a successor of i in its marks set. That is, when literal i fails, the backtrack literal j has the property that

$$j = max\{c \mid c \in candidates_G[i] \wedge (marks[c] \bigcap (\{i\} \bigcup succ_G[i]) \neq \emptyset)\}$$



index	literal	candidates
1	q	{2,3}
2	r	{1,3}
3	s	{1,2}
4	t	{1,2,3}
5	redo	{1}

The psuedo-literal "redo" is used to coodinate backtracking activities when a redo message is received

Figure 1: DDG for `p(A,B) :- q(A,B),r(A,C),s(A,B,D),t(C,D)`

where $marks[i]$ denotes the marks set of literal i. Now, if no such literal exists, the AND process fails. Otherwise, the backward step is affected by canceling or resetting the successors of j, untrailing the appropriate variables, and sending a redo message to the process corresponding to literal j.

The use of failure history is illustrated by the example in Figure 1. In this example, suppose that `s` rejects the bindings provided it by `q` (consumer failure). In this case, the failure of `s` will cause `q` to be marked with the index of `s`, and `q` will be subsequently selected as the backtrack literal. Suppose now that `s` accepts the bindings from `q`, but `t` rejects all values of `D` generated by `s`, so that `s` fails (generator failure). In this case, the failure of `t` will have caused `r` and `s` to be marked. When `s` fails, `r` will be selected as the backtrack literal. This selection reflects the fact that, while `t` had previously failed with all values of values of `D` generated by `s`, it may be solvable with the next value of `C` from `r` and some other value of `D` from `s`.

4 Generator Inheritance Graphs

The CS algorithm performs well for the special case of fixed data dependencies. However, if data dependencies change during clause execution, correct operation of the algorithm will generally require reconstruction the clause's data dependency graph, and thus its candidate sets. These changes occur when a literal which has been designated as the

generator of a variable fails to produce a ground binding for that variable. In this section we will see how the CS algorithm can be extended to handle dynamic data dependencies without recomputation of the candidate sets. This is accomplished by applying the algorithm to new type of data dependency graph, which we call a *generator inheritance graph* (GIG).

A generator inheritance graph statically describes the ways in which clause execution can cause generatorship of a variable to be propagated from its initial generator to some other literal. The propagation of variable generatorship, called *generator inheritance*, occurs when the generator of a variable fails to produce a ground binding for that variable. We depend on static analysis to uncover the program points at which generator inheritance is possible. [2]

Abstractly, a generator inheritance graph can be viewed as the union of a finite sequence $G_0, \ldots, G_n$ of data dependency graphs. G_0 is called the *initial DDG*, and corresponds to the DDG used by the algorithms described previously. For $0 < i \leq n$, G_i is called an *induced DDG*, and is derived from G_{i-1} when static analysis determines that a generator of G_{i-1} may fail to produce a ground binding for one or more of its generated variables.

The derivation of G_i from G_{i-1} proceeds as follows. Let $E(G)$ and $V(G)$ denote the edge and vertex sets, respectively, of a graph G. An edge in $E(G)$ has the form (a, b, l), representing a directed edge from a to b with label l. Let $Con(G, X)$ be the set of consumers of variable X in G, and let $generates(G, g)$ be the set of variables generated literal by g according to the description of DDG G. Now, when analysis determines that a generator g in G_{i-1} may fail to produce a ground binding for a generated variable, say X, a new graph G_i is constructed to reflect the change in generatorship. G_i will be the same as G_{i-1}, except that the consumer of X with the smallest index will become the generator of X, the edges from g to the other consumers of X will be removed, and there will be new edges connecting the new generator of X to the other consumers of X.

More formally, generatorship of X is propagated from g to μ, where $\mu = \min(Con(G_{i-1}, X))$. The set of edges removed from G_{i-1} to reflect the new generator of X, $ER(G_{i-1}, g, X)$, is computed as follows: $ER(G_{i-1}, g, X) = \{(g, k, X) \mid k \in Con(G_{i-1}, X) \backslash \{\mu\}\}$. The set of new edges, denoted $EA(G_{i-1}, g, X)$, is computed as follows: $EA(G_{i-1}, g, X) = \{(\mu, k, X.\mu) \mid (g, k, X) \in ER(G_{i-1}, g, X)\}$. Finally, the edge set of the DDG *induced* by G_{i-i}, g, and X, denoted $E(G_{i-1}, g, X)$, is computed as follows: $E(G_{i-1}, g, X) = E(G_{i-1}) \backslash ER(G_{i-1}, g, X) \bigcup EA(G_{i-1}, g, X)$. The edge set of the induced DDG G_i is constructed by forming the union of the $E(G_{i-1}, g, X)$, computed for each variable X which g is designated to generate (in G_{i-1}) but may fail to ground.

[2] See, for example, [9,10].

When an edge is added to an induced DDG during the computation of $EA(G_{i-1}, g, X)$, its label is expanded to record the fact that g is the generator of X in G_i. For purposes of our construction, we treat the new label as the name of a new "pseudo" variable. This maintains the generator/consumer semantics of the edges in a GIG. An edge labeled $X.L_0.L_1.....L_k$ records the information that L_0 is the generator of $X.L_0$, L_1 is the generator of $X.L_0.L_1$, and in general, L_j is the generator of $X.L_0.L_1.....L_j$. Notice that if at some point during clause execution, L_j becomes the actual generator of X, then the literals $L_0, L_1, \ldots, L_{j-1}$ will be predecessors of L_j, since each of these literals will have previously failed to produce a ground binding for X. This information will become important later when we compute the dynamic predecessor set for j.

An example generator inheritance graph derivation is shown in Figure 2. In this example, static analysis has discovered that that in literal one in G_0 may fail to ground X. Since literal two is the consumer of X with the smallest index, it "inherits" generatorship of X. The relevant sets for G_0, literal one, and X are

$$\begin{aligned} E(G_0) &= \{(1,2,X.1),(1,3,X.1),(1,4,X.1),(2,4,Y.2)\} \\ ER(G_0,1,X) &= \{(1,3,X.1),(1,4,X.1)\} \\ EA(G_0,1,X) &= \{(2,3,X.1.2),(2,4,X.1.2)\} \\ E(G_0,1,X) &= \{(1,2,X.1),(2,4,Y.2),(2,3,X.1.2),(2,4,X.1.2)\} \end{aligned}$$

Since the only variable generated by literal one is X, $E(G_1)$ is simply $E(G_0,1,X)$, as shown in Figure 2. Now, suppose that analysis discovers that literal two may also fail to ground X. In this case, generatorship of X is propagated to literal three, resulting in graph labeled G_2. Finally, a generator inheritance graph for the clause is obtained by forming the union of the edge sets of the G_i's, as shown by the final graph in the figure.

4.1 Extending the CS Algorithm

Our philosophy has been to minimize extensions to run time behavior of the CS algorithm and concentrate the major effort at "compile time". As a result, the extended algorithm follows the same two-phase approach as the original algorithm: the marks set is constructed in the first phase, and the candidate sets are searched during the second phase. The extended algorithm is defined in terms of following three sets, which are computed using a generator inheritance graph G.

- $candidates_G[i]$ – The candidate set for literal i. The computation of the candidate set for literal i is the same as in the original CS algorithm, except that the predecessor and successor sets used in the computation are derived from the generator inheritance graph G.

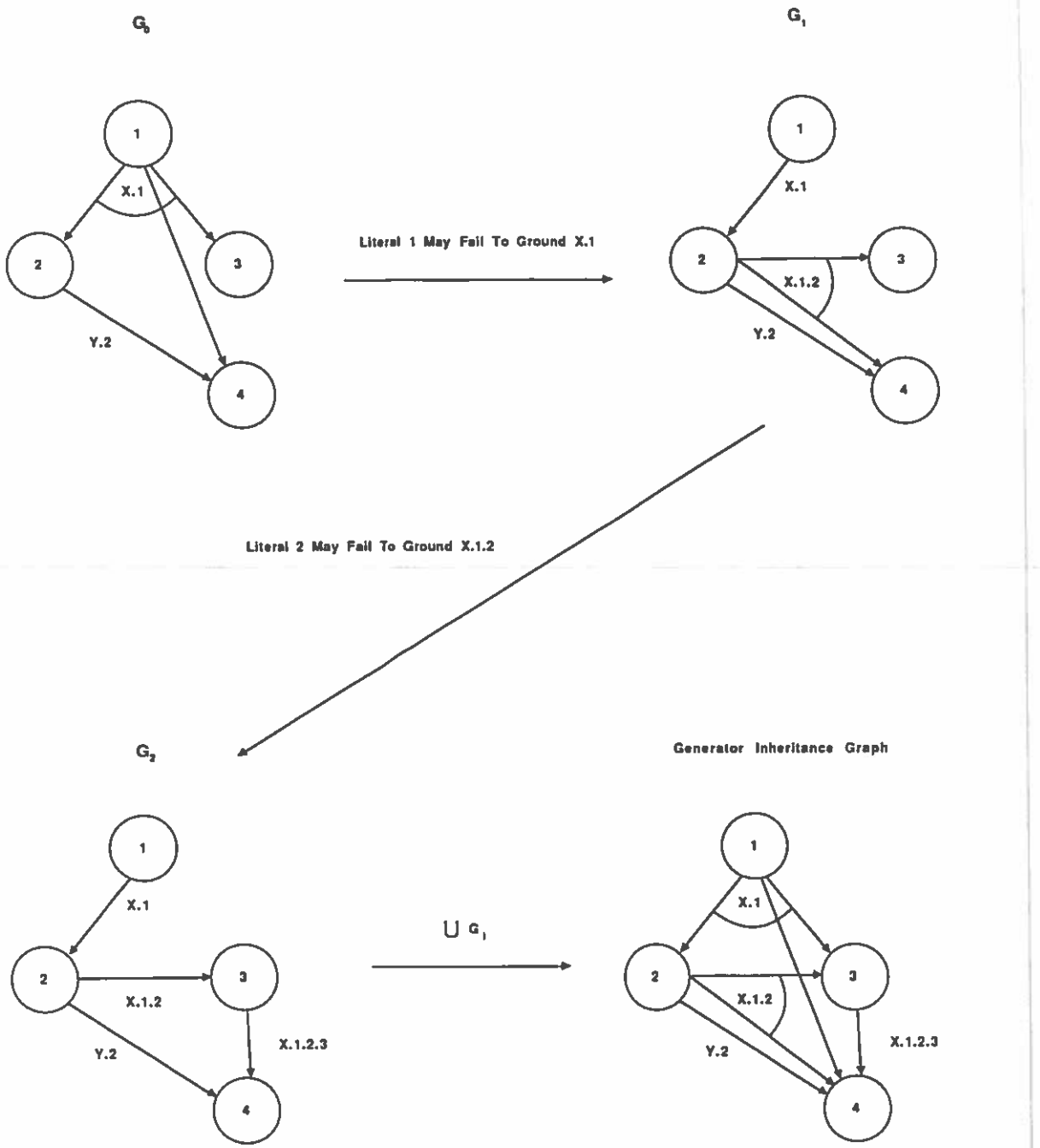


Figure 2: Initial DDG, Induced DDGs, and GIG

- $initial\_predecessors_G[i]$ – The set of literals that are known at compile time to be predecessors of i. Note that $initial\_predecessors_G[i] = pred_{G_0}[i]$.
- $consumes_G[i]$ – The set of variables consumed by literal i, *i.e.* $\{X \mid (g, i, X.\alpha) \in E(G)\}$.

The marking phase of the extended algorithm is divided into two sub-phases. During the first sub-phase, the set of literals which are statically known to be predecessors of the failed literal, say literal i, are marked with the index i. The static predecessors of literal i are simply the elements of $initial\_predecessors_G[i]$. The next sub-phase marks the dynamic predecessors of literal i. The dynamic predecessor set of literal i is the set of inherited generators which have actually generated a variable consumed by literal i at the time that literal i fails. To determine which, if any, of the inherited generators are actual generators when a literal fails, a generator attribute is maintained for each variable in the clause. When literal i fails, any inherited generator which is a dynamic predecessor of i is marked with the index i.

Notice that we can use the generator inheritance graph to restrict the set of variables and generators that must be inspected during construction on the dynamic predecessor set. In particular, the set of variables that must be inspected when literal i fails is simply $\{X \mid X.g.\alpha \in consumes_G[i]\}$. Furthermore, we need only check the generators in α for generatorship of X.

During the search phase of the extend algorithm, the candidate sets are searched in the same manner as the original CS algorithm. That is, the backtrack literal, if it exists, is a literal j such that

$$j = max\{c \mid c \in candidates_G[i] \wedge (marks[c] \bigcap (\{i\} \bigcup succ_G[i]) \neq \emptyset)\}$$

The difference between the two algorithms occurs at "compile time". In the original CS algorithm, the graph G used in the above expression is a data dependency graph; in the extended algorithm, it is a generator inheritance graph. To complete our extension, we show that searching candidate sets computed using a generator inheritance graph preserves the correctness of the original CS algorithm.

4.2 Extended Candidate Sets

The CS algorithm uses the candidate sets to restrict the search for an appropriate backtrack literal. In the original algorithm, a literal L_c is candidate to cure the failure of L_f if L_c is either a predecessor of L_f or a predecessor of a successor of L_f. In this case, the predecessors of L_f correspond directly to the literals that must complete execution before L_f is allowed to execute. However, this is not necessarily the case for generator

inheritance graphs. Instead, some predecessors may correspond to inherited generators, and are therefore "conditional". Now, suppose that an inherited generator L_{IG} is not actually a generator at the time L_f fails. Then since L_{IG} is not a predecessor of L_f, we may not have required that L_{IG} execute before the L_f. In this case, backtracking to L_{IG} may not cure the failure at hand. Thus the candidate sets computed using a generator inheritance graph may contain irrelevant literals.

As an example of how these irrelevant literals arise, consider the generator inheritance graph in Figure 2. Now, suppose literal four fails, and literal two is the actual generator of X (as depicted by the graph labeled G_1 in the figure). Notice that literal three is a candidate to cure the failure of literal four, since it is a predecessor (in the GIG) of literal four. Backtracking to literal three is a bad choice, however, since it cannot cure this failure when literal two generates X. The correct choice is literal two. Thus for this failure sequence, literal three is a superfluous candidate.

Surprisingly, the presence of these superfluous literals in a candidate set does not affect the correctness of the algorithm. To see this, observe that the CS algorithm requires only that the correct backtrack literal be a member of the failed literal's candidate set. This is because the superfluous literal will not be marked during the failure sequence, and thus will not be selected as the backtrack literal. Indeed, the backtrack literal could be correctly defined to be the largest element of the set $\{l \mid l \in V(G) \wedge (marks[l] \bigcap (\{i\} \bigcup succ_G[i]) \neq \emptyset)\}$.[3] This being the case, we can show that our extensions to the CS algorithm preserve its correctness by showing that the candidate sets computed using a generator inheritance graph contain the appropriate literals for any data dependency that could arise during clause execution.

Theorem 1 *Let $G_0, G_1, \ldots, G_n$ be the sequence of induced DDGs derived during construction of a generator inheritance graph G. Then*

$$candidates_{G_0}[i] \subseteq candidates_{G_1}[i] \subseteq \ldots \subseteq candidates_{G_n}[i] = candidates_G[i]$$

Proof: To simplify our discussion, we make the following definitions. Let $path_G(X,Y)$ be true *iff* there exists a path from X to Y in G, let $GF_{G_k}[i]$ be the set $\{(X,Y) \mid path_{G_k}(i,Y) \wedge path_{G_k}(X,Y)\}$, and let Π_1 be the standard projection function which selects the first element of a tuple.

To prove the theorem, we use the following technical lemmas.

Lemma 1 $GF_{G_0}[i] \subseteq GF_{G_1}[i] \subseteq \ldots \subseteq GF_{G_n}[i] = GF_G[i]$

Proof: Notice that for $0 \leq i < n$, a path is never removed from G_i during the construction of G_{i+1}. In particular, suppose that a generator $g \in V(G_i)$ may fail to

[3] In fact, Conery initially describes the the search phase of the original CS algorithm in this way [3].

ground variable X. Let $\mu = \min(Con(G_i, X))$. Then for $a \in Con(G_i, X) \setminus \{\mu\}$, the construction ensures that the paths containing an edge (g, a, X) in $E(G_i)$ are rerouted through the edges (g, μ, X) and $(\mu, a, X.\mu)$ in $E(G_{i+1})$. Thus by construction, $path_{G_i}(g, a) \Longrightarrow path_{G_{i+1}}(g, a)$. □

Lemma 2 $pred_{G_0}[i] \subseteq pred_{G_1}[i] \subseteq \ldots \subseteq pred_{G_n}[i] = pred_G[i]$

Proof: The proof follows from the construction, as described in Lemma 1, and the definition of $pred_G[i]$. □

Lemma 3 $cp_{G_k}[i] = \Pi_1(GF_{G_k}[i])$

Proof: $b \in cp_{G_k}[i] \iff \exists x \in G_k \mid x \in succ_{G_k}[i] \wedge b \in pred_{G_k}[x]$
$\iff \exists x \in G_k \mid path_{G_k}(i, x) \wedge path_{G_k}(b, x)$
$\iff (b, x) \in GF_{G_k}[i]$
$\iff b \in \Pi_1(GF_{G_k}[i])$. □

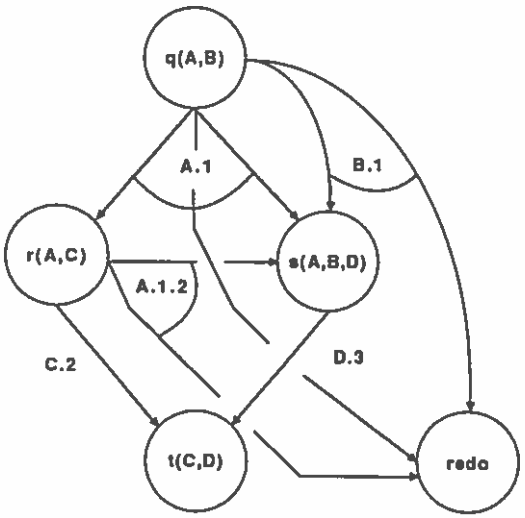
Corollary 1 $cp_{G_0}[i] \subseteq cp_{G_1}[i] \subseteq \ldots \subseteq cp_{G_n}[i] = cp_G[i]$

The theorem follows immediately from Lemma 2, Corollary 1, and the definition of $candidates_G[i]$. Since the sequence of induced DDGs includes all of the data dependencies that might arise from G_0 in the global environment, the candidate sets computed using the GIG G contain the correct backtrack literal for any failure sequence.

4.3 Discussion

Our goal has been to extend the CS algorithm so that the effect of run time changes in clause level data dependencies are minimized, while at the same time retaining the precision and efficient representation provided by the original algorithm. For this reason, the operation and data structures of the extended algorithm closely parallel those of the original algorithm. Indeed, the run time behavior of the algorithm remains largely intact, with the exception that in some cases variable generator must be inspected while constructing the marks set.

These extensions are illustrated by the example in Figure 3. This example shows the GIG that results when the graph in Figure 1 is used as an initial DDG, and global analysis reveals that `q` may fail to ground `A`. Now, consider the case in which `s` immediately fails due to the bindings supplied it by its predecessors (consumer failure). Further, suppose that execution of `q` has failed to ground `A`. In this case, `q` will be marked with the index of `s`, since `q` is an initial predecessor of `s`. Next, `r` is checked for generatorship of `A`, since `r` is a possible dynamic predecessor of `s` (`A.1.2` is consumed by `s`). Since `r` is



index	literal	candidates
1	q	{2,3}
2	r	{1,3}
3	s	{1,2}
4	t	{1,2,3}
5	redo	{1,2}

Figure 3: Generator Inheritance Graph and Its Candidate Sets

the current generator of A, it will be marked with the index of s. r is then selected as the backtrack literal, since it is the marked candidate of s with largest index.

The run time costs of the extensions result from managing the generator attributes. These costs, along with the associated space costs, have been shown to be negligible [5,7]. For example, if we assign an integer index to each variable in a clause, then we can represent the variable generators for a clause with m variables and n literals as a vector of $m * \log n$ bits. The generator of variable k is kept in the $\log n$ bits from $k * \log n$ through $(k+1) * \log n - 1$.

We can use this representation to efficiently implement recording and checking of generator attributes. To record the fact that literal i has become the generator of variable j, we update the generator attribute vector A as follows. Let $Shift_j = j * \log n$, let J_0 be a bit mask of size $m * \log n$ which has zeros in positions $j * \log n$ through $(j+1) * \log n - 1$ and ones in the other positions, let I be the $\log n$-bit representation of i, padded on the right with $(m-1) * \log n$ zeros, and let $\triangleright$ be the logical right shift operator. [4] Then the updated version of A is simply $(A \wedge J_0) \vee (Shift_j \triangleright I)$.

Checking variable generatorship is also very efficient. We check literal i for generatorship of variable j as follows. Let J_1 be a bit mask of size $m * \log n$ which has ones in positions $j * \log n$ through $(j+1) * \log n - 1$ and zeros in the other positions, and let $\triangleleft$ and $\otimes$ be the logical left shift and exclusive-or operators, respectively. Then i is the

[4] $X \triangleright Y$ represents Y right shifted X bit positions.

generator of j *iff* $((A \wedge J_1) \lhd Shift_j) \otimes I)$ is zero. Since J_0,J_1,I, and $Shift_j$ are fixed values, they can be computed at compile time. Given these efficient representations, the extended algorithm runs with cost comparable the original CS algorithm [7].

Finally, notice that the extended algorithm described in this paper can be viewed as a backward execution analog to DeGroot's RAP scheme [4]. In particular, the independence and ground checks employed by the RAP algorithm during forward execution to construct the dynamic data dependencies are analogous to the generator checks used by the extended algorithm during backward execution. Further, the edges added during construction of a generator inheritance graph correspond directly to the edges (implicitly) added by the RAP algorithm when it discovers variables are nonground or dependent.

5 Summary and Future Work

This paper has described a new type of data dependency graph, and shown how these graphs can be used to extend the CS algorithm. The extended algorithm improves on the algorithms in its class by using a static representation of the set of possible data dependencies for a clause. The extended algorithm attempts to minimize run time expense by computing the set of possible data dependency graphs at compile time, and using variable generator attributes to discriminate among the possible graphs at run time. A simple abstract machine based on the extended algorithm has been implemented, and various optimizations based on this representation have been reported [8].

Areas for further study include analysis techniques for GIG construction, and optimizations based on these analyses. For example, in some cases, it is known that a generator will *always* leave a generated variable unbound [11]. This information allows edges to be removed from the corresponding GIG, which in turn reduces the amount of generator checking that must be done at run time.

Acknowledgments

I wish to thank Prof. John Conery for his valuable suggestions for improvements and careful proof reading of earlier drafts of this paper.

References

[1] Chang, J., Despain, A.M., and DeGroot, D. AND-parallelism of logic programs based on static data dependency analysis. In *COMPCON Spring 85*, (Feb.), IEEE, 1985, pp. 218–225.

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

[3] Conery, J.S. Implementing backward execution in nondeterministic AND-parallel systems. In *Proceedings of the Fourth International Conference on Logic Programming*, (Melbourne, Australia, May 25–29), 1987, pp. 633–653.

[4] DeGroot, D. Restricted AND-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, (Tokyo, Japan, Nov. 6–9), 1984, pp. 471–478.

[5] Kumar, V. and Lin, Y. A Data-Dependency-Based Intelligent Backtracking Scheme for Prolog. *Journal of Logic Programming 5*, 2 (1988).

[6] Lin, Y., Kumar, V., and Leung, C. An intelligent backtracking algorithm for parallel execution of logic programs. In *Proceedings of the Third International Conference on Logic Programming*, (London, England, Jul. 14–18), Springer-Verlag, 1986, pp. 55–68.

[7] Meyer, D.M. *Architected Failure Handling For AND-Parallel Logic Programs.* OM note 89-02, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, 1989.

[8] Meyer, D.M. and Conery, J.S. *Architected Failure Handling For AND-Parallel Logic Programs.* Tech. Rep. CIS-TR-89-05, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, March 1989. Submitted to NACLP '89.

[9] Warren, R., Hermenegildo, M., and Debray, S.K. On the Practicality of Global Flow Analysis of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Aug. 1988, pp. 684–699.

[10] Winsborough, W. *Automatic, Transparent Parallelization of Logic Programs at Compile Time.* PhD thesis, The University of Chicago, Aug. 1988.

[11] Winsborough, W. and Waern, A. Transparent And-Parallelism in the Presence of Shared Free Variables. In *Proceedings of the Fifth International Logic Programming Conference/Symposium*, Seattle, Aug. 1988.

[12] Woo, N.S. and Choe, K. Selecting the backtrack literal in the AND process of the AND/OR Process Model. In *Proceedings of the 1986 Symposium on Logic Programming*, (Salt Lake City, UT, Sep. 22–25), 1986, pp. 200–210.