Compilation of Logic Programs for Restricted And-Parallelism

Dean Jacobs and Anno Langen
Computer Science Department, University of Southern California
Los Angeles, California 90089-0782

1. Introduction

A variety of different techniques for executing logic programs in parallel have been proposed. One common approach is to provide new language constructs for explicitly controlling the order in which goals execute. Languages containing such constructs include Concurrent Prolog[Shapiro 86], Parlog[Clark 86], and Epilog[Wise 82, Wise 86]. An alternative approach is to have the implementation schedule goals automatically. Conery [Conery 81, Conery 87] has developed mechanisms for monitoring program variables and scheduling goals at run-time. This approach provides a great deal of parallelism, however, it can incur considerable overhead. Chang[Chang 85a, Chang 85b] schedules goals on the basis of a static analysis of the program at compile-time. This approach reduces run-time support, however, it may miss important opportunities for parallelism.

DeGroot [DeGroot84] has developed the *Restricted And-Parallelism* (RAP) execution model for logic programming which combines compile-time analysis with run-time monitoring. In this model, programs are compiled into special control expressions, called *Execution Graph Expressions* (EGEs), which initiate goals on the basis of tests on program variables. The RAP model is easier to implement than a general graph-based scheme such as Conery's because EGEs have a linear form; goals are collected into groups for scheduling. This permits simple "fork/join"-style synchronization mechanisms. In general, this linear form restricts the amount of parallelism that can be achieved because initiation of a goal cannot be made dependent upon the completion of an arbitrary collection of other goals. DeGroot argues that RAP can still achieve more than enough parallelism to keep a moderately-sized multiprocessor computer busy. Recent empirical evidence [Carlton 88] lends support to these arguments.

Research on the RAP model has focused primarily on the problem of implementing EGEs efficiently on multiprocessor computers. Hermenegildo [Hermenegildo 87] has extended Warren's Abstract Machine for Prolog [Warren 83] to support RAP. The complementary problem of compiling clauses into EGEs has been addressed only briefly [DeGroot 87a]. This problem is difficult for two reasons. First, determining whether an EGE is correct for a clause may entail considerable reasoning about tests and goals. Second, choosing from among various correct alternatives may require a rather subjective assessment of their benefits. This is because different EGEs may achieve more parallelism in different circumstances. Moreover, an EGE which achieves a great deal of parallelism may be unacceptably large.

In this paper, we introduce a novel approach to generating correct EGEs and choosing between them. In our approach, a clause is first converted into a graph-based computational form, called a *Conditional Dependency Graph* (CDG), which achieves *Maximal And-Parallelism* (MAP). MAP is the maximal and-parallelism possible while maintaining correctness. This CDG is then gradually transformed into an EGE, potentially at a loss of parallelism, using two rewrite rules operating on hybrid expressions. Since these rules are sound, in the sense that they always produce correct results from correct sources, correct EGEs are always produced. Compilation algorithms are defined within this framework by giving heuristics for choosing where and how rules should be applied. We briefly discuss initial work on the design of such heuristics.

This paper is organized as follows. In section 2, we introduce the RAP model, define EGEs, and motivate the compilation problem. In section 3, we define correctness and MAP. In section 4, we define CDGs and prove that a clause can be converted into a CDG which achieves MAP. In section 5, we introduce our compilation framework, define the rules for transforming CDGs into EGEs, and prove their soundness. In section 6, we develop certain algorithms for reasoning about CDGs. In section 7, we summarize this work and discuss our current research.

2. Restricted And-Parallelism

There are two basic kinds of parallelism available in a logic program: and-parallelism, in which independent parts of the same solution are pursued simultaneously, and or-parallelism, in which different solutions are pursued simultaneously. In general, it can be difficult to exploit or-parallelism effectively since the pursuit of alternatives which are not used can waste considerable resources. Many implementations therefore limited the amount of or-parallelism which can occur.

The RAP model was designed to exploit only and-parallelism. Or-parallelism occurs whenever concurrently executing goals produce different values for a shared variable. The RAP model avoids or-parallelism by ensuring that goals are scheduled for concurrent execution only if they do not share variables. This is accomplished by compiling clauses into special control expressions, called Execution Graph Expressions, which initiate goals on the basis of dependency tests.

2.1 Execution Graph Expressions

We use the following terminology in discussing EGEs. *Variable identifiers* appear textually within goals, e.g., x and y appear in $A(x,y)$, and are bound to terms when a program executes. *Variables* are unnamed objects which appear inside terms and may be unified with other terms. Let variable identifiers x and y be bound to terms t and u respectively. x becomes *further instantiated* when a variable in t becomes unified with a term. x is *grounded* if t contains no variables. x and y are *dependent* if t and u contain a common variable, otherwise they are *independent*. Two goals are *dependent* if there is a variable identifier of the first goal and a variable identifier of the second goal which are dependent, otherwise they are *independent*.

EGEs are defined inductively as follows. Every goal is an EGE and for EGEs E_1 and E_2 the following are EGEs.

$(PAR\ \ E_1\ E_2)$
$(SEQ\ \ E_1\ E_2)$
$(IF\ P\ \ E_1\ E_2)$
$(CPAR\ P\ \ E_1\ E_2)$

Execution of an EGE proceeds from the outermost expression inwards. A *PAR* expression executes the subexpressions concurrently. A *SEQ* expression executes the subexpressions sequentially from left to right. An *IF* expression executes E_1 or E_2 depending on whether the condition P is true or false. A *CPAR* expression may be viewed as an abbreviation for

$(IF\ P\ \ (PAR\ \ E_1\ E_2)$
$\quad\quad (SEQ\ \ E_1\ E_2))$

A condition P is a conjunction of basic tests on variable identifiers and is used to determine whether goals are independent. Two kinds of basic tests are provided: Ixy holds iff x and y are independent and Gx holds iff x is grounded.

We now give several examples of EGEs. The clause $H(x,y)$:- $A(x)$, $B(y)$ can be compiled into the EGE $(CPAR\ Ixy\ \ A(x)\ B(y))$ since $A(x)$ and $B(y)$ can execute in parallel as long as x and y are independent. Similarly, the clause $H(x)$:- $A(x)$, $B(x)$ can be compiled into the EGE $(CPAR\ Gx\ \ A(x)\ B(x))$ since $A(x)$ and $B(x)$ can execute in parallel as long as x is grounded. The clause $H(x,y)$:- $A(x)$, $B(y)$, $C(x,y)$ can be compiled into the EGE

$(IF\ Gx\ \ (PAR\ \ A(x)\ (CPAR\ \ Gy\ \ B(y)\ C(x,y)))$
$\quad\quad (IF\ Gy\ \ (PAR\ \ B(y)\ (SEQ\ \ A(x)\ C(x,y)))$
$\quad\quad\quad\quad (IF\ Ixy\ \ (SEQ\ \ (PAR\ \ A(x)\ B(y))\ C(x,y))$
$\quad\quad\quad\quad\quad\quad (SEQ\ \ A(x)\ (SEQ\ \ B(y)\ C(x,y))))))$

which executes one of four sub-EGEs depending on x and y.

2.2 The Compilation Problem

In this section, we motivate our compilation techniques by way of several examples. These examples show that a compiler must be able to

- reason about independence of goals to ensure correctness,
- accurately determine when losses of parallelism occur, and
- choose between different alternatives when losses of parallelism occur or when the "perfect" expression is unacceptably large.

To determine whether an EGE is correct, a compiler must be able to infer that, whenever two goals might execute in parallel, the sequence of tests that lead to that point were sufficient to ensure that the goals were independent. As an example, consider the first branch of the last EGE in the previous section. Since x is grounded on this branch, we know x and y are independent so $A(x)$ can execute in parallel with the other goals. In addition, $B(y)$ and $C(x,y)$ can execute in parallel if y is grounded. Similar arguments show that the other branches are correct. In general, the inferencing

necessary to determine whether an EGE is correct can be quite complex. For example, in the EGE (*IF Ixy* (*SEQ* E_1 E_2) E_3) it may be necessary to analyze E_1 to determine if *Ixy* still holds when E_2 starts executing.

The following example demonstrates that the linear form of EGEs restricts the amount of and-parallelism that can be achieved. In this example, and throughout this paper, we consider only those EGEs which lead to dependent goals executing in the same order as they appear in the original clause. This restriction is in the spirit of Prolog; we rely on the programmer to determine the best order in which to execute dependent goals. The limitations of EGEs are still present if goal reordering is permitted.

Consider the clause $H(x,y)$:- $A(x)$, $B(x)$, $C(y)$, $D(x,y)$. An EGE of the form
(*IF Gx* E_1 (*IF Gy* E_2 (*IF Ixy* E_3 E_4)))
tests all relevant initial cases for this clause and therefore can achieve the maximal possible parallelism. Now consider the third branch, where x and y are independent but neither is grounded. Suppose we decide to construct E_3 using the subexpression (*PAR* (*SEQ* $A(x)$ $B(x)$) $C(y)$) for which there is no loss of parallelism. Under no condition can $D(x,y)$ be initiated in parallel with this subexpression since $D(x,y)$ cannot start until after $A(x)$ finishes. We must therefore compose $D(x,y)$ sequentially after this subexpression. But if $A(x)$ grounds x and $C(y)$ finishes before $B(x)$, then $D(x,y)$ will needlessly wait for $B(x)$ to finish. As an alternative, if we try starting with (*PAR* $A(x)$ $C(y)$) then $B(x)$ will end up needlessly waiting for $C(y)$ to finish. Finally, if we let $A(x)$ execute on its own, then $C(y)$ will end up needlessly waiting for $A(x)$ to finish. Thus, every possible EGE for E_3 leads to losses in parallelism. This is because initiation of a goal cannot be made dependent on the completion of an arbitrary collection of other goals. Note that none of these expressions is clearly the best with respect to parallelism alone, since for each of them there are circumstances where it does better than the others.

We conclude this section with a discussion of certain implementation details of the RAP model and the way in which these details affect compilation. In general, evaluation of a test such as *Ixy* requires scanning the terms bound to x and y at run-time. Since such terms can be arbitrarily large and may have to be scanned quite frequently, testing can introduce considerable overhead. An important part of the RAP model is an algorithm which efficiently computes an *approximation* to these tests. This algorithm is safe in the sense that it never allows a test to succeed when it should have failed. Thus, independent goals may be viewed as being dependent but not the converse.

It is our view that the inexactness of tests should not directly affect the compilation process; the same strategy for deriving EGEs should be used regardless of whether tests are exact or inexact. However, the inexactness of tests should *indirectly* affect compilation. Consider the EGE (*IF Ixy* (*SEQ* E_1 E_2) E_3) introduced earlier. Suppose analysis of E_1 shows that *Ixy* still holds when E_2 starts executing. If the tests were exact, we would be at liberty to retry *Ixy* in E_2 anyway; at worst we would be performing a redundant test. However, since the tests are inexact, if we retry *Ixy* in E_2 it may fail, thereby reducing the amount of parallelism achieved. Thus, it is important to avoid redundant testing as much as possible.

3 Maximal And-Parallelism

In this section, we present a formal notion of correctness and define MAP to be the maximal and-parallelism possible while maintaining correctness.

Execution of a clause begun in some initial state is correct if the following two restrictions are observed.

- Dependent goals never execute concurrently.
- Dependent goals never execute out of order.

The second restriction is in the spirit of Prolog; we rely on the programmer to determine the best order in which to execute dependent goals. These restrictions are characterized by the following constraint on scheduling.

Definition: Constraint on Scheduling

(*) A goal should not be initiated while there is a dependent goal to its left which has not finished executing. ∇

Theorem: (*) Characterizes the Correctness of Execution

Dependent goals never execute concurrently or out of order if and only if (*) is observed.

Proof Outline:

Only if: Suppose (*) is not observed.

At the point (*) is violated, a goal B is initiated while there is a dependent goal A to its left which has not finished executing. If A is currently executing, then two dependent goals are executing concurrently, violating the first restriction. If A is not executing, then A and B are executing out of order, violating the second restriction.

If: Suppose (*) is observed.

First, we show that concurrently executing goals A and B, A left of B, must be independent. Under (*), A and B must be independent at the time B is initiated. We can show that they will remain independent as long as B is executing since no goal can get access to variables of both A and B. Second, we show that dependent goals A and B, A left of B, cannot execute out of order. Under (*), B cannot be initiated until A has finished. ∇

MAP is the maximal and-parallelism possible while maintaining correctness.

Definition: Scheduling Discipline for MAP

(**) A goal should be initiated *as soon as* all dependent goals to its left have finished executing. ∇

Definition: Correctness and MAP

Let E be the representation of a clause in some computational formalism such as EGEs. E is *correct* if execution of E begun in any initial state observes (*). E *achieves MAP* if execution of E begun in any initial state implements (**). ∇

4 Conditional Dependency Graphs

In this section, we introduce CDGs and prove that a clause can be converted into a CDG which achieves MAP.

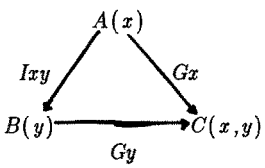
A CDG is a directed acyclic graph where the vertices are goals and each edge is labeled by a condition. The CDG $\Theta(L)$ associated with a clause L has a vertex for each goal of L and an edge from goal A to goal B if A is left of B. The condition labeling edge $<A,B>$ is *indep*(A,B), defined as follows.

Definition: indep

The condition *indep*(A,B) is the conjunction of the following tests: Gx for every variable identifier x occurring in both A and B, and Ixy for every variable identifier x occurring in A but not B and variable identifier y occurring in B but not A. ∇

It is easy to show that goals A and B are independent if and only if the condition *indep*(A,B) holds. Note that there is no need to test whether a variable identifier occurring in both goals is independent from other variable identifiers since we require that it be grounded.

As an example, the clause $H(x,y)$:- $A(x)$, $B(y)$, $C(x,y)$. given in section 2.1 is associated with the following CDG.



The CDG execution model is as follows.

Definition: CDG Execution Model

Perform the following two step execution cycle repeatedly. A cycle should start as soon as a goal finishes or a variable identifier is further instantiated.

1) Edge Removal: Remove every edge whose origin has finished executing. If the conditions hold on all edges going into a goal, then remove those edges.

2) Goal Initiation: Initiate all goals with no incoming edges. ∇

The following theorem shows that $\Theta(L)$ achieves MAP.

Theorem: $\Theta(L)$ achieves MAP

Execution of $\Theta(L)$ begun in any initial state implements (**).

Proof:

We must show that a goal B is initiated during execution of $\Theta(L)$ as soon as all dependent goals to its left have finished executing. Consider the change which results in all goals left of B being either finished or independent of B for the first time. A CDG execution cycle will start as soon as this change occurs. Since all edges going into B come from goals on its left, all such edges will be removed after the first step of this cycle. Thus, B will be initiated in the second step of this cycle. It remains to be shown that B could not have been initiated in a previous cycle. During every previous cycle, there was some dependent goal A to B's left which had not finished executing. The edge $<A,B>$ initially in $\Theta(L)$ could not have been removed after step one in any of these cycles. Thus, there was always an edge going into B and B could not have been initiated in step two. ∇

5 A Compilation Framework

In this section we introduce our framework for compiling clauses into EGEs. In our approach, a clause L is first converted into the CDG $\Theta(L)$ which achieves MAP. This CDG is then gradually transformed into an EGE, potentially at a loss of parallelism, using two rewrite rules operating on hybrid expressions. We show the rules are sound in the sense that they always transform correct hybrids into correct hybrids. This ensures that any EGE derived from $\Theta(L)$ will be correct since $\Theta(L)$ is correct.

The hybrid expressions consist of EGEs with CDGs as well as goals as elementary components. The rewrite rules replace a CDG within a hybrid by an EGE containing more refined sub-CDGs. When an *IF* expression is introduced, facts about dependencies between variable identifiers become known. Such facts are used to simplify conditions on the edges of sub-CDGs -- edges whose conditions have been simplified to true are removed. This embodies the process of reasoning about sequences of tests. In general, facts can become invalidated during execution and inferencing is necessary to determine where they can be applied. Such inferencing can be complex since the question of whether a particular goal can invalidate a particular fact depends on what other facts are known.

To facilitate simplification, we introduce the notion of an *Extended CDG* (ECDG) $<\Gamma, C>$ consisting of a CDG Γ together with a set of facts C referred to as its *context*. Hybrid expressions consist of EGEs with ECDGs as well as goals as elementary components. We arrange it so that, roughly speaking, each fact in the context of an ECDG holds before its first goal is initiated. We use two functions for manipulating contexts which are defined in the next section. The first, $post(C,\Gamma)$, returns the set of facts in context C which are maintained by execution of CDG Γ. The second, $simplify(\Gamma, C)$, returns the CDG Γ simplified under the context C.

Our first rewrite rule, called the Split Rule, introduces *PAR*, *SEQ*, and *CPAR* expressions.

Definition: The Split Rule

Input to the Split Rule consists of a ECDG $<\Gamma, C>$ and a partitioning of Γ into two sub-CDGs α and β. The Split Rule may be applied only if there are no edges from β to α. If there are no edges at all between α and β, then the result of the Split Rule is $(PAR \ <\alpha, C> <\beta, C>)$. Otherwise, let $CG_{\alpha\beta}$ be the conjunction of the conditions on edges from α to β. If $CG_{\alpha\beta}$ is equivalent to *false*, then the result is $(SEQ \ <\alpha, C> <\beta, post(C,\alpha)>)$ otherwise the result is $(CPAR \ CG_{\alpha\beta} \ <\alpha, C> <\beta, post(C,\alpha)>)$. ∇

Our second rewrite rule, called the If Rule, introduces *IF* expressions.

Definition: The If Rule

Input to the If Rule consists of a ECDG $<\Gamma, C>$ and a condition P. Let $C_T = C \cup P$ and $C_F = C \cup \neg P$ where $C \cup X$ denotes the context C extended with all facts derivable from X. The result of the If Rule is $(IF \ P \ <simplify(\Gamma, C_T), C_T> <simplify(\Gamma, C_F), C_F>)$. ∇

As an example, we use the two rewrite rules to derive the EGE

$$
\begin{array}{l}
(IF\ Gx\ \ (PAR\ \ A(x)\ (CPAR\ \ Gy\ \ B(y)\ C(x,y))) \\
\qquad (IF\ Gy\ \ (PAR\ \ B(y)\ (SEQ\ \ A(x)\ C(x,y))) \\
\qquad\qquad (IF\ Ixy\ \ (SEQ\ \ (PAR\ \ A(x)\ B(y))\ C(x,y)) \\
\qquad\qquad\qquad (SEQ\ \ A(x)\ (SEQ\ \ B(y)\ C(x,y))))))
\end{array}
$$

given for the clause $H(x,y)$:- $A(x)$, $B(y)$, $C(x,y)$. in section 2.1. The CDG associated with this clause was given in section 3. We first apply the If Rule with the condition Gx. When the "then" branch is simplified, both edges coming out of $A(x)$ are removed, since Gx implies Ixy. Applying the Split Rule to separate $A(x)$ from the other goals results in the PAR expression with a sub-CDG in the second half. Splitting this sub-CDG results in the CPAR for $B(y)$ and $C(x,y)$.

When the "else" branch is simplified, the condition on the edge from $A(x)$ to $C(x,y)$ is replaced by *false*. We now apply the If Rule with the condition Gy to this CDG. When the "then" branch is simplified, the edges from $A(x)$ to $B(y)$ and $B(y)$ to $C(x,y)$ are both removed. Applying the Split Rule to separate $B(y)$ from the other goals results in the PAR expression with a sub-CDG in the second half. Splitting this sub-CDG results in the SEQ for $A(x)$ and $C(x,y)$. The other two branches are derived in an analogous manner.

We now prove the soundness of the rewrite rules. To accomplish this we define an execution model for hybrid expressions which is a straight-forward combination of the EGE and CDG execution models. At the outermost level, hybrids execute according to the EGE model. The CDG in an ECDG occurring in place of a goal is initiated at the point that goal would be initiated, executes according to the CDG model, and finishes as soon as all its goals have finished. Using this execution model, we define a notion of correctness for hybrids.

Definition: Correctness of Hybrids

A hybrid H is *correct under some context* if during execution of H begun in any initial state satisfying that context

1) goals are scheduled correctly and

2) for each ECDG $<\Gamma, C>$ in H, every fact in C which is *relevant* to Γ holds before the first goal in Γ is initiated. ∇

The relevant facts for Γ are those facts which can affect its simplification, as described in the next section.

Theorem: Soundness of the Split Rule

If the Split Rule is applied to an ECDG $<\Gamma, C>$ in a source hybrid which is correct under some context, then the result hybrid will also be correct under that context.

Proof Outline:

First, we argue that goals in the result hybrid will be scheduled correctly. It is always correct when α and β execute concurrently in the result hybrid since they would have executed concurrently in the source hybrid. It is always correct when α executes sequentially before β in the result hybrid since there are no edges from β to α.

Second, we argue that appropriate contexts are introduced. If α and β execute concurrently in the result hybrid, then there are no dependencies between them and it follows from results in the next section that neither can invalidate relevant facts for the other. If α executes sequentially before β

then all facts in C will hold before α executes and, by the correctness of *post*, all facts in *post*(C,α) will hold before β executes. ∇

Theorem: Soundness of the If Rule

If the If Rule is applied to an ECDG $<\Gamma, C>$ in a source hybrid which is correct in some context, then the result hybrid will also be correct in that context.

Proof:

It is clear that appropriate contexts are introduced. The fact that goals will be scheduled correctly follows directly from the above statement and the correctness of *simplify*. ∇

We now argue that every EGE derived in our framework is correct. The initial context $\Omega(L)$ for a clause L is derived using the fact that the first occurrence of a "local" variable identifier is guaranteed to be independent from all other variable identifiers. For example, the initial context for the clause $H(x)$:- $A(x)$, $B(y)$, $C(x,y)$ consists of the fact Ixy. Note that input mode information about the arguments of a procedure, e.g., that some argument is always grounded, can be easily incorporated into the initial context. Such information may be derived from global program analysis[Chang 85b] or programmer annotations[Shapiro 87].

Compilation begins with the initial hybrid $<$*simplify*($\Theta(L)$,$\Omega(L)$),$\Omega(L)>$. The correctness of $\Theta(L)$ and *simplify* ensures that the initial hybrid is correct under $\Omega(L)$. By the soundness of the rewrite rules, every EGE derived from the initial hybrid will be correct under $\Omega(L)$.

6 Simplification Algorithms

In this section, we develop algorithms for maintaining contexts and simplifying CDGs. We define the following two functions which appear in the previous section.

post(C,Γ) returns the set of facts in context C which are maintained by execution of CDG Γ.

simplify(Γ , C) returns the CDG Γ simplified under the context C.

In our framework, the fact that variable identifiers x and y are independent is denoted Ixy and the fact that they are dependent is denoted Dxy. The fact that variable identifier x is grounded is denoted Ixx and the fact that it is not grounded is denoted Dxx. We write Fxy when we want to discuss a fact independently of whether it is of the form Ixy or Dxy.

Definition: Contexts

A *context* C is a set of facts which satisfies the following properties.

$Ixy \in C$ iff $Iyx \in C$
$Dxy \in C$ iff $Dyx \in C$
$Ixy \in C \Rightarrow Dxy \notin C$
$Dxy \in C \Rightarrow Ixy \notin C$
$Ixx \in C \Rightarrow Ixy \in C$ for all relevant y ∇

As a first step in computing *post*, we define the function $MF(C, G\vec{v})$ which determines the set of facts in context C which are maintained by execution of a single goal $G\vec{v}$ with variable identifiers v_i. We assume a fact Fxy is maintained iff, according to C, the v_i are either independent from x or

independent from y. Note that output mode information about the arguments of a procedure, e.g., that some argument always becomes grounded, can be incorporated into the function MF to allow more facts to be maintained. Such information may be derived from global program analysis[Chang85b] or programmer annotations[Shapiro87]. In the following definition, we write $I\vec{v}x \in C$ to denote $\forall i(Iv_i x \in C)$.

Definition: Maintained Facts

$MF(C, G\vec{v}) = \{Fxy \in C \mid I\vec{v}x \in C \vee I\vec{v}y \in C\}$ ∇

The following theorem shows that MF is correct and complete.

Theorem: Correctness and Completeness of MF

$Fxy \in MF(C, G\vec{v})$ iff $Fxy \in C$ and Fxy holds after execution of $G\vec{v}$ begun in any state satisfying C.

Proof:

To invalidate Ixy, x and y must be further instantiated so that the terms they are bound to share a common variable; variables must be unified with terms containing this common variable. To invalidate Dxy, x and y must be further instantiated so that the terms they are bound to no longer share common variables; all common variables must be unified with ground terms. A goal may perform such actions iff it has access to a variable in the term bound to x and a variable in the term bound to y. $G\vec{v}$ has access only to variables in the terms bound to the v_i. Thus, $G\vec{v}$ can invalidate Fxy iff $I\vec{v}x \in C \vee I\vec{v}y \in C$ does not hold. ∇

As a second step in computing *post*, we define the function *propagate*(C,S) which returns the set of facts in context C which are still valid after a *sequence* of goals S has finished executing. We write $< >$ to denote the empty sequence and $G.S$ to denote the sequence S with the goal G appended to the front.

Definition: propagate

propagate$(C, < >) = C$

propagate$(C, G.S) =$ *propagate*$(MF(C, G)\ ,\ S)$ ∇

The correctness and completeness of *propagate* follows directly from the correctness and completeness of MF.

Intuitively, *post*(C,Γ) should be the intersection of *propagate*(C,S) over all possible sequences of goals S from Γ. We now prove certain properties of MF which show that it is not necessary to consider all such sequences. The following theorem characterizes the propagation of a context across two goals. We write $I\vec{v}\vec{w} \in C$ to denote $\forall i,j(Iv_i w_j \in C)$.

Theorem: Propagation Across Two Goals

$Fxy \in MF(MF(C, G\vec{v})\ ,\ H\vec{w})$ iff $Fxy \in C$ and at least one of the following conditions hold.

1) $I\vec{v}x \in C \wedge I\vec{w}x \in C$
2) $I\vec{v}x \in C \wedge I\vec{w}y \in C \wedge I\vec{v}\vec{w} \in C$
3) $I\vec{v}y \in C \wedge I\vec{w}y \in C$
4) $I\vec{v}y \in C \wedge I\vec{w}x \in C \wedge I\vec{v}\vec{w} \in C$

Proof:

First, we show that these conditions are sufficient. In the first case, $I\vec{v}x \in C$ guarantees $Fxy \in MF(C, G\vec{v})$ and $I\vec{w}x \in MF(C, G\vec{v})$, since $I\vec{w}x \in C$. Together, these facts imply

$Fxy \in MF(MF(C,G\vec{v})\ ,\ H\vec{w})$. In the second case, $I\vec{v}x \in C$ guarantees $Fxy \in MF(C,G\vec{v})$ and $I\vec{v}\vec{w}$ guarantees $I\vec{w}y \in MF(C,G\vec{v})$, since $I\vec{w}y \in C$. Together, these facts imply $Fxy \in MF(MF(C,G\vec{v})\ ,\ H\vec{w})$. The consideration is symmetric for the third and fourth cases.

Second, we show that these conditions are necessary. Suppose neither $I\vec{v}x$ nor $I\vec{v}y$ holds, then $Fxy \notin MF(C,G\vec{v})$ and therefore $Fxy \notin MF(MF(C,G\vec{v})\ ,\ H\vec{w})$. Suppose only $I\vec{v}x$ holds; the consideration is symmetric if only $I\vec{v}y$ holds and leads to the third and fourth cases. If $Fxy \in MF(MF(C,G\vec{v})\ ,\ H\vec{w})$ is to hold, then either $I\vec{w}x \in MF(C,G\vec{v})$ or $I\vec{w}y \in MF(C,G\vec{v})$ must hold. The first of these is possible only if $I\vec{w}x \in C$; this is the first case. The second of these is possible only if $I\vec{w}y \in C \wedge I\vec{v}\vec{w} \in C$; this is the second case. ∇

The previous theorem has an important corollary: the set of facts which result when a context is propagated across a sequence of goals is independent of the order in which the goals appear in the sequence.

Corollary: Commutativity of MF

$MF(MF(C,G)\ ,\ H) = MF(MF(C,H),G)$

Proof:

The characterization of the propagation of a context across two goals is symmetric in those goals. Note this relies on the property $Ixy \in C$ iff $Iyx \in C$, since the characterization contains the condition $I\vec{v}\vec{w} \in C$. ∇

Corollary: Permuting Sequences

Let $\hat{S}$ be any permutation of a sequence S of goals, then $propagate(C,S) = propagate(C,\hat{S})$.

Proof:

By commutativity of MF. ∇

As stated earlier, $post(C,\Gamma)$ should be the intersection of $propagate(C,S)$ over all possible sequences of goals S from Γ. The corollary above shows that it suffices to consider any one sequence. In the following definition, we use $nat(\Gamma)$, the sequence of goals of Γ in their "natural" left-to-right order with respect to the original clause.

Definition: post

$post(C,\Gamma) = propagate(C,nat(\Gamma))$ ∇

The correctness and completeness of *post* follows directly from the above corollary and the correctness and completeness of *propagate*.

We now develop an algorithm to compute $simplify(\Gamma,C)$, the CDG Γ simplified under the context C. Simplification consists of reducing conditions on edges and removing edges whose conditions have been reduced to true. To modify edges going out of a goal G, we compute the set of facts which hold before G is initiated; this set is called the initiation context for G.

We might define the initiation context for G to be $propagate(C,S)$, where S is a sequence of all the goals in Γ which could execute before G. This would be too pessimistic, however, since a goal right of G in the original clause can execute before G only if the goals are independent. More facts can be maintained by taking into account the independence of G and goals to its right. The following theorem shows that, in fact, goals right of G need not even be included in S because they cannot

affect facts which are *relevant* to G. A fact is relevant to G iff it appears in a condition labeling an edge going out of G. Thus, it suffices if $S = nat(\Gamma, G)$, the sequence of all goals in Γ which are left of G in their natural order.

Definition: Relevant Facts

A fact Fxy is *relevant* to G iff x or y is a variable identifier of G. ∇

Theorem Ignoring Goals to the Right

Suppose $Fxy \in C$ and Fxy is relevant to G. $Fxy \in propagate(C, nat(\Gamma, G))$ iff Fxy holds when G is initiated during execution of Γ begun in any initial state satisfying C.

Proof Outline:

Only if: Suppose Fxy holds whenever G is initiated. Since it is possible that all goals left of G finish execution before G is initiated, $Fxy \in propagate(C, nat(\Gamma, G))$ by the completeness of *propagate*.

If: Suppose $Fxy \in propagate(C, nat(\Gamma\ ,\ G))$ but Fxy does not hold when G is initiated during execution of Γ begun in some initial state satisfying C. By the correctness of *propagate*, some goal $R\vec{v}$ right of G must have executed before G and caused Fxy not to hold. Consider the facts in C that $R\vec{v}$ might have invalidated to do this. Such facts include Fxy itself and every fact necessary to maintain Fxy through the goals initiated after $R\vec{v}$ and before G. Let SS be the set of goals initiated after $R\vec{v}$ and before G. We assume SS does not contain any goals right of G; an inductive generalization of our argument covers this case. Every fact of interest must have a variable identifier w in common with G or a goal in SS. In order for $R\vec{v}$ to invalidate such a fact, $Dv_i w$ for some i must have held when $R\vec{v}$ was initiated. But then (*) was violated since $R\vec{v}$ is right of G and every goal in SS. ∇

The theorem above shows that we can define the initiation context for G to be $propagate(C, nat(\Gamma\ ,\ G))$. This suggests an efficient algorithm for computing $simplify(\Gamma\ ,\ C)$: compute $propagate(C, nat(\Gamma))$ and, along the way, reduce edges coming out of each goal as it is encountered. The correctness and completeness of *simplify* with respect to relevant facts follows directly from the above theorem and the correctness and completeness of *propagate*.

7 Summary and Current Research

In this paper, we have presented a framework for compiling clauses into EGEs. We showed how a clause can be converted into a Conditional Dependency Graph which achieves Maximal And-Parallelism, and then gradually transformed into an EGE, potentially at a loss of parallelism, using two rewrite rules. The problem of reasoning about the correctness of EGEs within this framework was solved by developing algorithms for maintaining sets of facts about test results and simplifying sub-CDGs.

A primary focus of our current research is the development of heuristics for determining where and how rules should be applied. We base such heuristics on two important insights. First, the Split Rule divides its source CDG into smaller pieces while the If Rule duplicates its source CDG. Thus, compilation should be driven by attempts to apply the Split Rule. Second, there is a loss of

parallelism associated with the Split Rule but not with the If Rule. This means that the Split Rule cannot be applied blindly; an attempt must be made to apply it in circumstances where the least amount of parallelism is lost. The If Rule should be used to set up such circumstances, but not to the extent that it makes the resulting EGE too large.

We are exploring ways of quantifying the loss of parallelism associated with applications of the Split Rule to facilitate choosing between different places to split. In the result of the Split Rule, ($CPAR\ CG_{\alpha\beta}\ \alpha\ \beta$), the goals in α and β have been collected into groups for scheduling. Thus, a goal in β can run concurrently with a goal in α only if *all* goals in β can run concurrently with *all* goals in α. Losses of parallelism occur when goals in β have to wait unnecessarily for goals in α to finish. We are also exploring ways of quantifying the usefulness of particular conditions in applications of the If Rule. Such conditions are useful to the extent that simplification of the "then" and "else" branches leads to subsequent low-cost splits. It remains to be seen whether a single sequence of transformations can produce acceptable EGEs or whether backtracking will be required.

There are several other important issues which we plan to address. First, we plan to characterize the set of EGEs which can be derived in our framework. We will show that the rules are complete for the set of all EGEs that might be "reasonably" produced by a compiler. Second, we plan to investigate ways of taking advantage of mode information about the arguments of procedures, which might be derived from global program analysis[Chang 85] or programmer annotations[Shapiro 87]. Input mode information can be added to the initial context for a clause and output mode information can be used in *MF* to maintain more facts. Third, we plan to investigate ways of handling the extra-logical features of Prolog such as *assert*, *retract*, *read*, *write*, and *cut*. The simplest way of doing this is to introduce artificial dependencies between every side-effect goal and the goals before and after it. While this produces the correct semantics for Prolog, it unnecessarily restricts the amount of parallelism that can be achieved. DeGroot has dealt with this problem by incorporating special synchronization constructs into the RAP model[DeGroot 87b]. We plan to extend our techniques to deal with these constructs. Finally, we plan to extend our framework to allow goal reordering.

References

[Carlton 88] Carlton,M. and Van Roy,P., A distributed Prolog system with and-parallelism, Dept. of EECS, Univ. of Cal. at Berkeley, Submitted to *1988 Hawaii International Conference on System Sciences*, (1988).

[Chang 85a] Chang, J.-H., High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis, Ph.D. thesis in Dept. of EECS, Univ. of Cal. at Berkeley, Report No. UCB/CSD 86/263, (1985).

[Chang 85b] Chang, J.-H., Despain, A., and DeGroot, D., AND-parallelism of logic programs based on a static dependency analysis, *Proc. Spring Compcon*, IEEE, (1985), 218-225.

[Clark 86] Clark, K.L., and Gregory, S., PARLOG: Parallel programming in logic, *ACM TOPLAS* 8,1 (1986), 1-49.

[Conery 81] Conery, J., and Kibler, D., Parallel interpretation of logic programs, *Proc. Conf. on Func. Prog. Lang. and Comp. Arch.*, ACM (1981), 163-170.

[Conery 87] Conery, J., *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, (1987).

[DeGroot 84] DeGroot, D., Restricted and-parallelism, *Proc. Int. Conf. Fifth Gen. Comp. Sys.*, ICOT, (1984), 471-478.

[DeGroot 87a] DeGroot, D., A technique for compiling execution graph expressions for restricted and-parallelism in logic programs, *Proc. Int. Supercomputing Conf.* Athens, Greece (1987).

[DeGroot 87b] DeGroot, D., Restricted and-parallelism and side-effects, *Proc. Symp. Log. Prog.*, IEEE, San Francisco, (1987).

[Hermenegildo 87] Hermenegildo, M., *A Restricted And-Parallel Execution Model and Abstract Machine for Prolog Programs*, Kluwer Academic Press, (1987).

[Shapiro 86] Shapiro, E., Concurrent Prolog: a progress report, *IEEE Computer* 19, 8 (1986), 44-58.

[Warren 83] Warren, D.H.D., An abstract Prolog instruction set, Tech. Note 309, SRI International, Oct. (1983).

[Wise 82] Wise, M.J., A parallel Prolog: the construction of a data driven model, *Proc. Conf. LISP and Func. Prog.*, ACM, (1982).

[Wise 86] Wise, M.J., *Prolog Multiprocessors*, Prentice-Hall International, (1986).