An Abstract of the Thesis of

Daniel R. Corpron for the degree of Master of Science in Computer Science presented on November 26, 1986. Title: Disjunctions in Forward-Chaining Logic Programming

Redacted for Privacy

Abstract approved:
Thomas G. Dietterich

A forward-chaining logic programming system (FORLOG) has been developed at Oregon State University. This system coupled with an assumption-based truth maintenance system (ATMS), provides an alternative to the logic programming paradigm of backward-chaining with Horn clauses. To compare FORLOG to this paradigm, we define a subset of FORLOG, called mini-FORLOG, with a restricted syntax. This syntax is derived in a straight forward manner from a Horn clause program. We prove that mini-FORLOG is procedurally isomorphic to Horn clause programming.

Logic programs derived from Horn clause programs contain implications with disjunctive consequences. Each disjunct may either lead to a solution or to a contradiction. We want FORLOG to choose only those that lead to solutions. In an attempt to remove the nondeterminism inherent in this choice, FORLOG compiles implications into several simpler ones, all void of disjunctions. Even with compilation, FORLOG often must pursue each nondeterministic choice, some of which may turn out to be contradictory. This requires FORLOG to handle discovered inconsistencies. With traditional logic, the existence of an inconsistency invalidates all deductions made. FORLOG uses the ATMS to maintain consistency of its database of facts.

One alternative in a disjunction might not terminate. It turns out that FORLOG can use contradictions discovered during computation to avoid some of these cases. This process involves resolving the disjunctions with discovered contradictions.

Disjunctions in Forward Chaining Logic Programming

by

Daniel R. Corpron

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed November 26, 1986

Commencement June 1987

Approved:

Redacted for Privacy

Assistant Professor of Computer Science in charge of major

Redacted for Privacy

Head of department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented November 26, 1986

Acknowledgements

I wish to use this opportunity to thank my advisor, Tom Dietterich, for his help and guidance. He is largely responsible for the ideas leading to FORLOG's development. I feel very fortunate to have worked with a researcher of his caliber.

Fellow student Nick Flann with his hard work and dedication has been a constant source of inspiration. He also provided many useful comments on early drafts of this thesis.

I would also like to thank my family and friends for their patience and support. I am particularly grateful to my parents, Doug and Helen Corpron who have always encouraged and aided my educational endeavors. Most of all, I am indebted to Julie Corpron, my spouse and accomplice in life. She has provided immeasurable support throughout my years of graduate study.

The bulk of the research reported here was supported by the National Science Foundation under grant DMC-8514949.

Table of Contents

List of Figures

Disjunctions in Forward-Chaining Logic Programming

Chapter 1
Introduction

Logic programming involves writing programs as sets of logical assertions. Program execution consists of applying a restricted theorem prover to deduce consequences of the assertions.

The Handbook of Artificial Intelligence (Cohen & Feigenbaum, 1982) and many texts on logic programming (Kowalski, 1979, Clocksin & Mellish, 1984, Lloyd, 1984, Sterling & Shapiro, 1986) define a logic program as a set of *Horn clauses.* This restricted definition of a logic program along with a simple interpreter provides an effective method for deducing consequences of these clauses. One particular clause in the program is designated the *goal clause.* The interpreter takes a goal driven approach, thus following a *backward-chaining* (also known as top-down) strategy.

The general form of a Horn clause is

$$\neg C_1 \lor \neg C_2 \lor \ldots \lor \neg C_{k-1} \lor C_k$$

where each C_i, $1 \leq i \leq k$, is a positive literal. A positive literal is defined as an atomic formula $p(term_1,...,term_n)$ for some predicate p. In this view, the *goal clause* is a Horn clause with all negative literals.

Semantically, a ground goal G of a program P asks whether G is deducible in P. If, however, a goal contains logic variables, they are existentially quantified. For example, the goal $p(term_1,...,term_n)$, containing the variables $x_1,x_2,...,x_m$ reads:

"Are there values for $x_1,x_2,...,x_m$ such that $p(term_1,...,term_n)$?" For convenience, existential quantification is usually omitted.

The logic behind answering the question posed by the goal clause involves using a proof procedure common in mathematics: proof by contradiction. Assume that the goal is negative, if a contradiction is derived, then the goal is true. The resolution rule (Robinson, 1965) is employed as the only rule of inference. Negating the goal is the reason for defining a goal clause as a Horn clause with no positive literals. Any given goal may have several successful computations, each resulting in a different output. Unfortunately, it may have some nonterminating computations as well. These nonterminating computations generally mean the program has no (computable) answer.

Many factors contribute to the popularity of Horn clause logic programming. As we shall see, it has a simple procedural and declarative semantics. Very efficient implementations exist, for example, Prolog (Warren, 1977), MU-Prolog (Naish, 1985a), Concurrent Prolog (Shapiro, 1983), Parlog (Clark and Gregory, 1986), and GHC (Ueda, 1985). Also, in practice it has proven to be useful for many different applications in engineering and artificial intelligence.

A project at Oregon State University has explored an alternative paradigm. The result of this research being the FORLOG logic programming system. As its name implies, FORLOG utilizes a forward-chaining approach to logic programming. It defines a logic program as a set of assertions. These assertions may take the form of almost any closed, well formed formula. The FORLOG interpreter applies *modus ponens* to deduce consequences of the assertions.

This paper examines the manner in which FORLOG uses a particular formula. This formula turns out to have particular relevance to Horn clause logic programming. The general form of this formula is

$$\forall\ x_1,x_2,...,x_n\ \ p(x_1,x_2,...,x_n)\ \supset\ D_1 \vee D_2 \vee ... \vee D_n,$$

where p is a predicate and each D_i is an existentially quantified conjunction of literals. Given an instance of p, using modus ponens, we can detach the disjunction in the consequent. By itself, $D_1 \vee D_2 \vee \ldots \vee D_n$ probably would not allow the interpreter to deduce much more. If however, FORLOG knew which members of the disjunction were satisfiable, it could use them to find further deductions and avoid the others entirely. By *compiling* the implication into several equivalent implications, FORLOG attempts just that. We shall refer to the members of a disjunction as *branches*. For example, with the disjunction $D_1 \vee D_2 \vee \ldots \vee D_n$ above, each D_i is a branch.

This paper is organized as follows. Chapter 2 provides a foundation for FORLOG by demonstrating that forward-chaining logic programming is, at the least, capable of mimicking backward-chaining with Horn clauses. Chapter 3, then discusses issues surrounding FORLOG's method for compiling implications having disjunctive consequences. Chapter 4 illustrates a problem that arises with disjunctive programming and shows a solution provided by FORLOG. The final chapter provides some concluding remarks.

Several individuals have made significant contributions to FORLOG's development. Colin Gerity designed and implemented the basic ATMS facilities (section 3.2). Nick Flann designed and implemented the basic FORLOG interpreter, including its method for handling negated literals (section 3.3). My involvment with FORLOG included the following:

- with Dr. Tom Dietterich, discovered and proved the procedural isomorphism between a subset of FORLOG and interpreters for backward-chaining with Horn clauses (chapter 2)
- implemented FORLOG's method for compiling implications with disjunctive consequences (sections 3.1 and 3.4)
- formalized the soundness of FORLOG's use of negation (section 3.3)
- modified the basic ATMS, adding disjunctive assumptions and giving it the ability to use resolution (chapter 4).

Chapter 2
Backward- vs. Forward-Chaining Logic Programming

This chapter describes abstract computation models for the two logic programming strategies. For comparison purposes, the model for FORLOG describes a computation strategy for logic programs with a restricted syntax. This syntax is a subset of the full FORLOG programming language and is derived from the completion of a Horn clause logic program. We call this model mini-FORLOG.

The final section of this chapter describes one of the important results of this thesis: Horn clause logic programming is procedurally isomorphic to mini-FORLOG.

2.1 Computation Model for Backward-Chaining with Horn Clauses

For logic programming purposes, Horn clauses are usually written as implications. This simple syntactic change means the clause

$$\neg C_1 \vee \neg C_2 \vee \ldots \vee \neg C_{n-1} \vee C_n$$

is rewritten as

$$C_n \text{ :- } C_1 \wedge C_2 \wedge \ldots \wedge C_{n-1}$$

and is read "C_n is true if C_1, C_2, ..., and C_{n-1} are true." The *head* of the clause is C_n and the conjunction $C_1 \wedge C_2 \wedge \ldots \wedge C_{n-1}$ is the *body*.

Given a Horn clause program, computation progresses by backward-chaining. At each stage we have a goal list—a list of predicates to be proven. A predicate from this list and a Horn clause are chosen such that the predicate unifies with the Horn clause's head. Next, a new goal list is composed by replacing the chosen predicate with the body of the chosen clause, and then applying the most general unifier of the clause's head and the chosen goal. When the goal list is empty, computation terminates with a successful solution to the goal. If, however, no clause's head unifies with a given subgoal, computation terminates with failure.

Any chosen predicate could unify with several different Horn clauses. Each clause might lead to a unique solution to the original goal. The choice of which clause to use is a non-deterministic one. If all possible solutions are desired, the interpreter must perform the computation several times, once for each solution. Whenever the computation reaches a point where it must make one of these non-deterministic choices, clauses that lead to results previously computed and clauses that lead to failure should be avoided, if possible.

More formally, a computation of a goal $Q=Q_0$ by a program P is a (possibly infinite) sequence of triples $\langle Q_i,G_i,C_i\rangle$. Q_i is a conjunctive goal, G_i is a goal occurring in Q_i, and C_i is a clause

$$p(t_1,t_2,...,t_n) \text{ :- } L_1 \wedge ... \wedge L_k$$

in P renamed so that it contains new variable symbols not occurring in any Q_j, $0 \leq j \leq i$. For all $i > 0$, Q_{i+1} is one of the following:

- the result of replacing G_i in Q_i by the body of C_i, and applying the substitution s_i, the most general unifier of G_i and the head of C_i
- the constant *true* if G_i is the only goal in Q_i and the body of C_i is empty
- the constant *fail*, if G_i and the head of C_i do not unify.

A trace of a computation of a logic program $\langle Q_i,G_i,C_i\rangle$ is the sequence of pairs $\langle G_i,s_i'\rangle$, where s_i' is the subset of the most general unifier s_i computed at the i^{th} reduction, restricted to variables in G_i.

Figure 2.1 illustrates an abstract interpreter for logic programs expressed as Horn clauses. This interpreter is based upon the one described by Sterling and Shapiro (1986). It solves a query G with respect to a program P. The output of the interpreter may be viewed as the bindings to logic variables in G that make G

true, or failure, if a failure has occurred during the computation. Note that the interpreter may also fail to terminate.

Input: A Horn clause logic program P, and a goal G

Output: The subset of *subs*, the composed set of unifiers, restricted to variables in G, or failure if failure has occured.

```
Algorithm:
Initialize the goal-list to be $G$, the input goal.
Initialize subs to be empty.
While the goal-list is not empty do
    1. Choose a goal $p(b_1,...,b_n)$ from the goal-list and a (renamed) clause
           $p(t_1,...,t_n)$ :- $L_1 \wedge L_2 \wedge ... \wedge L_k, k \geq 0,$
       from $P$.
    2. Unify $p(b_1,...,b_n)$ and $p(t_1,...,t_n)$ deriving the most general unifier $s$
       (exit if the unification fails).
    3. Compose $s$ and subs.
    4. Remove $p(b_1,...,b_n)$ from, and add $L_1$, $L_2$, ..., and $L_k$ to, the goal list.
    5. Apply $s$ to the goal-list and to $G$.

If the goal-list is empty output subs restricted to variables in $G$, else output
failure.
```

Figure 2.1: An abstract interpreter for Horn clause logic programs.

Step 3 in the interpreter of Figure 2.1 requires futher explanation. Composing two lists of substitutions constructs a combined set of substitutions from two other sets (such as s and *subs*). We shall use

$$S_1 = \{(x_1=t_1),...,(x_m=t_m)\}$$
$$S_2 = \{(x_1'=t_1'),...,(x_n'=t_n')\},$$

to represent the two sets to be composed. Each of the $(x_i=t_i)$ in S_1 represent the substitution of t_i for x_i. Even though it is represented as an equality, the substitution is not reflexive. That is, we would never substitute x_i for t_i. The left-hand side is always a variable, where the right-hand side is some arbitrary term. The same holds for the elements of S_2.

Composing S_1 and S_2 involves computing A ∪ B, where

$$A = \{(x=t) \mid x=x_i \text{ and } t=\text{Apply}(S_2,t_i) \text{ where } (x_i=t_i) \in S_1\}$$

B = $\{(x=t) \mid x=x_i'$ and $t=t_i'$ where $(x_i'=t_i') \in S_2$ and there is no y such that $(x_i'=y) \in S_1\}$.

The *Apply* operation in this step and in step 5 in the interpreter makes the substitutions throughout the target term (or list of terms).

2.1.1 Example: Horn Clause *delete*

To understand this abstract interpreter, examine the Horn clause program for *delete*

(2-1) $delete(x_1,[x_1|x_2],x_2)$.
(2-2) $delete(x_1,[x_2|x_3],[x_2|x_4])$:- $delete(x_1,x_3,x_4)$.

The above clauses define what it means to remove a single instance of an item from some list. The first parameter is the item to delete, the second the list to delete it from, and the third the resulting list.

Each clause handles a separate case. Clause (2-1) covers the situation where the item to delete heads the list. When this happens, the result is simply the rest of the list. The second clause (2-2) defines *delete* as the result of adding the first element of the list to the deletion of an item from the rest of the list.

Consider solving the goal $delete(c,[a,b,c,d],x)$? by the above program for *delete* using the abstract interpreter of Figure 2.1. The *goal-list* is initialized to be $delete(c,[a,b,c,d],x)$. The interpreter selects it as the goal to reduce, being the only one. Since the current goal only unifies with clause (2-1), it is chosen. The unifier of the goal and the clause's head is $\{(x_1=c), (x_2=a), (x_3=[b,c,d]), (x=[a|x_4])\}$.

The new *goal-list* becomes the instance of $delete(x_1,x_3,x_4)$—the body of chosen clause—with this unifier applied, namely $delete(c,[b,c,d],x_4)$. With the next iteration of the loop, this goal is chosen, being no other alternative. Once again the interpreter selects clause (2-1) but the variables must be renamed to avoid a name clash. The renamed clause is

$delete(x_5,[x_6|x_7],[x_6|x_8])$:- $delete(x_5,x_7,x_8)$.

The unifier of the goal and the clause's head is $\{(x_5=c), (x_6=b), (x_7=[c,d]), (x_4=[b|x_8])\}$. The new *goal-list* is $delete(c,[c,d],x_8)$. This time, the interpreter may choose clause (2-1), whose variables must be renamed as well

$$delete(x_9,[x_9|x_{10}],x_{10}).$$

The final unifier is $\{(x_9=c,\ x_8=[d])\}$. This time, the new *goal-list* is empty and the computation terminates.

To determine the result of the computation, we apply the relevant part of the unifiers calculated during the computation. The first unification instantiated x to $[a|x_4]$. The variable x_4 was involved in the substitution to $(x_4=[b|x_8])$ in the second unification, and x_8 later became fixed to $[d]$. Putting these together, x has the value $[a,b,d]$.

$\langle delete(c,[a,b,c,d],x),\ (x=[a|x_4])\rangle$
  $\langle delete(c,[b,c,d],x_4),\ (x_4=[b|x_8])\rangle$
    $\langle delete(c,[c,d],x_8),\ (x_8=[d])\rangle$
     true
     Output: $(x=[a,b,d])$.

Figure 2.2: Horn clause trace

The computation can be represented by a trace. The trace of the *delete* computation described above is presented in Figure 2.2. To make traces clearer, goals are indented one depth more than their parent.

2.2 Completion of Horn Clause Logic Programs.

Clark (1978) originally introduced the completion of a logic program in order to formalize the use of the negation-as-failure inference rule in Prolog. Here, we are not interested in negation-as-failure. Instead we wish to derive a FORLOG program that is equivalent to a pure Horn clause program (i.e. one without negations or extra-logical predicates). We will show that this program, when given to a constrained FORLOG interpreter (called mini-FORLOG), gives us the same behavior as the interpreter described in Figure 2.1.

The basic idea behind the completion is to find all the known ways of proving each literal and then assert that these are the only ways. Let P be a pure Horn clause program. The procedure involves the following. For each predicate symbol p appearing in program P there are two cases to consider. Either (1) p appears as the head of some clause or (2) it does not.

In case (1), gather up all clauses involving p in their head. Let one of the clauses be

$$p(t_1,t_2,...,t_n) \;\text{:-}\; L_1 \wedge L_2 \wedge ... \wedge L_m$$

where each t_i is a term involving the logic variables $x_1,...,x_a$ and each L_j is a positive literal. Let $y_1,...,y_b$ be all variables appearing in the clause's body but not appearing in its head. Let $u_1,...,u_n$ be unique variables to the entire clause. With these conventions, we can rewrite this clause to be

$$\forall\, u_1,...,u_n \;\; p(u_1,...,u_n) \;\text{:-}\; \exists\, x_1,...,x_a,y_1,...,y_b \;\; (u_1=t_1) \wedge ... \wedge (u_n=t_n) \wedge L_1 \wedge L_2 \wedge ... \wedge L_m.$$

This syntactic reformulation simply removed the terms $t_1,...,t_n$, from the clause's head and moved them to its body to form equalities. Notice that for an n-place predicate we create n equalities. The universal variables $u_1,...,u_n$ were introduced to communicate across the implication. Clark calls this the *general form* of a Horn clause.

Rewriting all clauses involving p makes their heads identical. Their bodies, however, may be quite different from one another. Name each body of the k general forms obtained in this way D_1, D_2, ..., D_k. Using E_i to represent the equality $u_i=t_i$, each D_j has the form

$$\exists\, x_1,...,x_a,y_1,...,y_b \;\; E_1 \wedge E_2 \wedge ... \wedge E_n \wedge L_1 \wedge L_2 \wedge ... \wedge L_m.$$

With the clause's heads identical, the final step involves combining each general form of p to obtain the formula

$$\forall\, u_1,...,u_n \;\; p(u_1,...,u_n) \equiv D_1 \vee D_2 \vee ... \vee D_k.$$

This gives the completion for the Horn clauses of p.

In case (2), where p is a predicate symbol that does not appear as the head of any rule, we add the formula

$$\forall\ u_1,\ldots,u_n\ \neg p(u_1,\ldots,u_n).$$

The completion of an entire logic program P, *comp*(P), is obtained by combining the separate completions of all predicates in P. The equalities produced employ a special theory. This theory of equality requires that two terms be equal if and only if they are unifiable. This can be captured by the following axioms:

- $c \neq c'$ for any distinct constants c and c'
- $f(x_1,\ldots,x_n) \neq g(y_1,\ldots,y_n)$ for distinct functors f and g
- $f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n) \supset (x_1=y_1) \wedge \ldots \wedge (x_n=y_n)$
- $f(x_1,\ldots,x_n) \neq c$
- $t(x) \neq x$ where $t(x)$ is any term in which x is free

2.2.1 Example: Completing *delete*

The delete program discussed earlier provides a simple example of computing the completion of a Horn clause program. Once again we have the clauses

(2-3) $delete(x_1,[x_1|x_2],x_2).$
(2-4) $delete(x_1,[x_2|x_3],[x_2|x_4])$:- $delete(x_1,x_3,x_4).$

To complete this program, we begin by converting each clause to general form. Clause (2-3) has the general form

$$\forall\ u_1,u_2,u_3\ \ delete(u_1,u_2,u_3)\ \text{:-}\ \exists\ x_1,x_2\ \ (u_1=x_1) \wedge (u_2=[x_1|x_2]) \wedge (u_3=x_2).$$

and clause (2-4) has the general form

$$\forall\ u_1,u_2,u_3\ \ delete(u_1,u_2,u_3)\ \text{:-}\ \exists\ x_1,x_2,x_3,x_4\ (u_1=x_1) \wedge (u_2=[x_2|x_3]) \wedge (u_3=[x_2|x_4]) \wedge delete(x_1,x_3,x_4).$$

After computing the general forms, we collect them together into a single formula to obtain the completion

$$\begin{aligned}\forall\ u_1,u_2,u_3\ \ delete(u_1,u_2,u_3) \equiv\ & [\exists\ x_1,x_2\ (u_1=x_1) \wedge (u_2=[x_1|x_2]) \wedge (u_3=x_2)]\\ & \vee\\ & [\exists\ x_1,x_2,x_3,x_4\ (u_1=x_1) \wedge (u_2=[x_2|x_3]) \wedge\\ & \quad (u_3=[x_2|x_4]) \wedge delete(x_1,x_3,x_4)].\end{aligned}$$

2.3 Computation Model for Forward-Chaining with the Completion

The completion of a Horn clause program provides a method for converting a Horn clause program to what are called mini-FORLOG programs.

DEFINITION. Given a Horn clause program *P*, the mini-FORLOG program, *for*(*P*), is derived from *comp*(*P*) by replacing all equivalences with forward implications.

In other words, start with a pure Horn clause program, compute its completion, and then replace each $\equiv$ with a $\supset$. For example, from the completion of *delete* we derive the mini-FORLOG program

$$\begin{aligned}\forall\, u_1,u_2,u_3\ \ delete(u_1,u_2,u_3) \supset\ & [\exists\, x_1,x_2\ (u_1{=}x_1) \wedge (u_2{=}[x_1|x_2]) \wedge (u_3{=}x_2)] \\ & \vee \\ & [\exists\, x_1,x_2,x_3,x_4\ (u_1{=}x_1) \wedge (u_2{=}[x_2|x_3]) \wedge \\ & \quad (u_3{=}[x_2|x_4]) \wedge delete(x_1,x_3,x_4)].\end{aligned}$$

Any mini-FORLOG program can be given to a restricted FORLOG interpreter, itself called mini-FORLOG. The only implications accepted by this interpreter are ones with single literals as antecedents, such as those in *for*(*P*).

As mentioned previously, FORLOG (and thus mini-FORLOG) takes a forward-chaining approach to logic programming. Mini-FORLOG computes the consequents of an assertion list and a mini-FORLOG logic program. This assertion list is a conjunction of literals and is analogous to the goal list of Horn clause programming. Instead of posing a question, however, the literals in the assertion list make demands. Semantically, an assertion $p(term_1,...,term_n)$, containing the *Skolem constants* $s_1,s_2,...,s_k$ reads: "Find restrictions on $s_1,s_2,...,s_k$ that must be true given that $p(term_1,...,term_n)$." Mini-FORLOG uses Skolem constants in the same way that Horn clause programming uses logic variables.

Within the forward-chaining view, the computation model for mini-FORLOG chooses a literal from the assertion list. If this literal matches the antecedent of an implication, a branch of the disjunction in the implication's consequent is chosen and added to the assertion list replacing the chosen predicate. This choice of which branch to pursue is a non-deterministic one.

As with Horn clauses, we pursue one result at a time. The computation model for mini-FORLOG derives single equality restrictions for each Skolem constant in the original assertion list. There could, however, be a set of restrictions for the individual Skolem constants. For completeness all values need to be derived, which means that every branch in the disjunctive consequent should be pursued. To do this, we can execute the model once for each answer. Whenever the model must make a choice between which branch to pursue, ones that lead to previously computed results and ones that lead to failure, should be avoided.

More formally, a computation of the goal assertion $S=S_0$ by program F is a (possibly infinite) sequence of triples $\langle S_i, A_i, B_i \rangle$. S_i is a (conjunctive) goal assertion, A_i is an assertion $p(c_1,...,c_n)$ in S_i and B_i is one of the D_j, appearing in the rule

$$\forall\ u_1,...,u_n\ \ p(u_1,...,u_n) \supset D_1 \vee D_2 \vee ... \vee D_m$$

which is itself in F. Each c_j is a term possibly containing Skolem constants. Recall from computing the completion of p that each D_j, and thus B_i, is a conjunction of n equalities $E_1,...,E_n$ and k literals $L_1,...,L_k$. Rename with unique Skolem constants the existential variables appearing in B_i, and replace each u_j with c_j, $0 \leq j \leq n$. This step derives new equalities $E_1',...,E_n'$ and new literals $L_1',...,L_k'$. For all $i > 0$, S_{i+1} is the result of one of the following:

- replacing A_i by the literals $L_1',...,L_k'$ in S_i, and applying to S_i the most general equalities e_i derived from $E_1',...,E_n'$ (more on e_i below)
- the constant *true* if A_i is the only goal assertion and there were no literals in B_i (ie. $k=0$)
- the constant *fail* if the equalities are contradictory
- the constant *fail* if we instead had the rule of the form

$$\forall\ u_1,...,u_n\ \neg p(u_1,...,u_n).$$

Deriving the most general equalities from $E_1',\dots,E_n'$ mirrors unification. Because of rewriting and substitution, each E_j' has the form $c_j=t_j'$. The equality axioms discussed previously mean that simplification may be performed by unifying each c_j with t_j'. To compute e_i, the most general equalities, compose results of these single unifications.

As with the model for Horn clause programming, a trace of a computation of a mini-FORLOG logic program $\langle S_i,A_i,B_i\rangle$ is the sequence of pairs $\langle A_i,e_i'\rangle$, where e_i' is the subset of the most general unifier e_i computed at the ith iteration, restricted to variables in A_i.

Input: A mini-FORLOG program F, and a start assertion S

Output: The subset of *equals*, the composed list of equalities, restricted to Skolem constants in S, or failure if failure has occurred.

Algorithm:

```
Initialize the assertion-list to be $S$, the start assertion.
Initialize equals to be empty.
While the assertion-list is not empty do
    1. Choose an assertion $p(c_1,\dots,c_n)$ from the assertion-list and $B$ a
       (skolemized) $D_j$ with each of the $c_i$'s replacing $u_i$'s. $D_j$ is a branch
       from the implication with $p$ as an antecedent
           $\forall\ u_1,\dots,u_n\ \ p(u_1,\dots,u_n) \supset D_1 \vee D_2 \vee \dots \vee D_j \vee \dots \vee D_m$.
       Therefore, $B$ has the following form:
           $E_1 \wedge E_2 \wedge \dots \wedge E_n \wedge L_1 \wedge L_2 \wedge \dots \wedge L_k$.
    2. Simplify the equalities, $E_1,E_2, \dots,$ and $E_n$, in $B$, deriving the most
       general equalities $e$ (exit if a contradiction arises).
    3. Compose $e$ and equals.
    4. Remove $p(c_1,\dots,c_n)$ from, and add $L_1$, $L_2$, ..., and $L_k$ to, the
       assertion-list.
    5. Apply $e$ to the assertion-list and to $S$.

If the assertion-list is empty output equals restricted to Skolem constants in
$S$, else output failure.
```

Figure 2.3: An abstract interpreter for mini-FORLOG.

Figure 2.3, describes an abstract interpreter for FORLOG. It processed an assertion S with respect to a program F. The output of the interpreter may be viewed as the derived equalities involving the Skolem constants in S, or failure, if a failure has occurred during the computation. As with the interpreter of Figure 2.1, this interpreter may fail to terminate.

2.3.1 Example: mini-FORLOG *delete*

To understand this strategy, examine the mini-FORLOG program for deleting as it appears in section 2.3.

Consider the deductions made as a consequence of the assertion $delete(c,[a,b,c,d],sk)$ using the abstract interpreter of Figure 2.3. The *assertion-list* is initialized to be $delete(c,[a,b,c,d],sk)$. It is chosen as the assertion to pursue, being the only one. The Skolemized disjunction from the program is

$$(c=sk_1) \wedge ([a,b,c,d]=[sk_2|sk_3]) \wedge (sk=[sk_2|sk_4]) \wedge delete(sk_1,sk_3,sk_4)$$

where the universal variables have been replaced. Simplifying the equalities gives us $\{(sk_1=c), (sk_2=a), (sk_3=[b,c,d]), (sk=[a|sk_4])\}$. The new *assertion-list* is the instance of $delete(x_1,x_3,x_4)$ with these equalities applied, namely $delete(c,[b,c,d],sk_4)$. Because there is no other choice again, this goal assertion is chosen in the next iteration of the loop. The same disjunction from *delete* is chosen but with different skolemized variables. The skolemized disjunction is

$$(c=sk_5) \wedge ([b,c,d]=[sk_6|sk_7]) \wedge (sk_4=[sk_6|sk_8]) \wedge delete(sk_5,sk_7,sk_8).$$

where, again, the universal variables have been replaced. Simplifying equalities this time give us $\{(sk_5=c), (sk_6=b), (sk_7=[c,d]), (sk_4=[b|sk_8])\}$. The new *assertion-list* is $delete(c,[c,d],sk_8)$. This time the skolemized disjunction is

$$(c=x_9) \wedge ([c,d]=[x_9|x_{10}]) \wedge (sk_8=x_{10}).$$

Again we have replaced the universal variables. The final simplification of equalities gives us $\{(x_9=c), (sk_8=[d])\}$. This time, the new *assertion-list* is empty and the computation terminates.

To determine the result of the computation, we apply the relevant part of the simplified equalities calculated during the computation. The first simplification equated *sk* to [*a*|sk_4], the second equated sk_4 to [*b*|sk_8], and sk_8 finally became fixed to [*d*]. Putting these together by using the law of substitution of equals, *sk* has the value [*a*,*b*,*d*].

```
<delete(c,[a,b,c,d],sk), (sk=[a|sk4])>
    <delete(c,[b,c,d],sk4), (sk4=[b|sk8])>
        <delete(c,[c,d],sk8), (sk8=[d])>
         true
         Output: (sk=[a,b,d]).
```

Figure 2.4: Mini-FORLOG trace

The computation can be represented by a trace. The trace of the delete computation described above is presented in Figure 2.4.

2.4 Isomorphism Between the Two Strategies

For brevity of this argument, we will call the logic programming strategy of backward-chaining with Horn clauses BACKLOG. Notice that neither interpreter provided a policy for adding and removing goals from the goal list. Prolog implements a depth-first version of BACKLOG. It chooses the leftmost goal instead of an arbitrary one, and replaces the non-deterministic choice of a clause by sequential search for a unifiable clause and it backtracks when failure occurs. Mini-FORLOG could have the exact same strategy.

Looking at the two interpreters and the delete example provided for each, it should be clear that BACKLOG and mini-FORLOG are computationally equivalent. The only major difference concerns the use of unification in Prolog and the use of equality in mini-FORLOG. To show this equivalence we shall prove that the two methods are *procedually isomorphic.*

DEFINITION. Two procedures A and B are procedurally isomorphic if there are three functions f_I, f_O, and f_S each of which is one-to-one and onto, such that

$$f_I(I_A) = I_B$$
$$f_O(O_A) = O_B$$
$$f_S(S_A) = S_B$$

where f_I takes any input given to A and returns the corresponding input given to B, f_O takes any output produced by A and maps it to the corresponding output from B, and f_S takes any internal state of A and maps it to the corresponding internal state of B.

THEOREM. BACKLOG and mini-FORLOG are procedurally isomorphic.
Proof. *P*, a Horn clause program, and *G*, a goal, represent the input to BACKLOG. Define f_I to map any input pair (P,G) to the input pair $(for(P),Sk(G))$, where *for* is the mapping from Horn clause programs to mini-FORLOG programs defined above. *Sk* is a one-to-one, onto mapping from logic variables to Skolem constants. When applied to a literal or a list of literals, *Sk* replaces all occurrences of each logic variable with the corresponding Skolem constant. When applied to substitutions, it similarly replaces <u>all</u> occurrences of logic variables (on either side of the equals sign) with appropriate Skolem constants. Further, define f_O and f_S to be *Sk* as well. To show the isomorphism, we must demonstrate the correspondence provided by f_O and f_S. Also, since BACKLOG and mini-FORLOG are nondeterministic, we must show that for every sequence of nondeterministic decisions in BACKLOG, there exists an isomorphic sequence of decisions in mini-FORLOG and vice-versa.

First we shall prove that f_S does in fact take any internal state of BACKLOG and map it to a corresponding internal state of mini-FORLOG. The state for BACKLOG is represented by the variables *goal-list* and *subs*. The state for mini-FORLOG is represented by the variables *assertion-list* and *equals*. We will use $goal\text{-}list_i$, $subs_i$, $assertion\text{-}list_i$, and $equals_i$ to indicate the values of these variables at the end of iteration *i*. Therefore, we wish to demonstrate that for all $i > 0$, $assertion\text{-}list_i{=}Sk(goal\text{-}list_i)$ and $equals_i{=}Sk(subs_i)$. We will use a proof by induction on the number of iterations in the respective loops. With no iterations, since the start assertion S=$Sk(G)$ by the definition of f_I, after initialization *assertion-list*$_0$=Sk(*goal-list*$_0$). Initializing *subs* and *equals* to be empty gives us *equals*$_0$=Sk(*subs*$_0$) trivially.

Now assume that for some m

assertion-list$_m$=Sk(*goal-list*$_m$)
equals$_m$=Sk(*subs*$_m$).

Now we must prove that after iteration m+1, that

assertion-list$_{m+1}$=Sk(*goal-list*$_{m+1}$)
equals$_{m+1}$=Sk(*subs*$_{m+1}$).

The five steps within the loops (see Figure 2.1 and Figure 2.3) maintain the equivalences.

1a. We must show that the goal literal chosen by BACKLOG corresponds to the assertion literal chosen by mini-FORLOG.

Since *assertion-list*$_m$=Sk(*goal-list*$_m$), the chosen goal in BACKLOG $p(b_1,...,b_n)$ $\in$ *goal-list*$_m$, corresponds to some assertion $p(c_1,...,c_n)$ $\in$ *assertion-list*$_m$. That is, $p(c_1,...,c_n)$=$Sk(p(b_1,...,b_n))$. Therefore, have mini-FORLOG choose the assertion $p(c_1,...,c_n)$.

1b. For this step, we want to demonstrate the correspondence between the body of BACKLOG's chosen clause and a branch of a disjunction from a mini-FORLOG's implication.

Let $p(t_1,...,t_n)$:- $L_1 \wedge ... \wedge L_k$ be the clause chosen by BACKLOG in this step. Recall that this clause would have the general form

$$\forall\ u_1,...,u_n\ \ p(u_1,...,u_n)\ \ \text{:-}\ \ \exists\ \overline{X}, \overline{Y}\ \ (u_1{=}t_1) \wedge ... \wedge (u_n{=}t_n) \wedge L_1 \wedge ... \wedge L_k.$$

where $\overline{X}$ are logic variables appearing in the $t_1,...,t_n$ and $\overline{Y}$ are logic variables appearing in the body but not the head (i.e., $\overline{X}$ and $\overline{Y}$ are disjoint). Because of the mapping *for*, mini-FORLOG has exactly one rule that defines p and it must have the right hand side of the general form appearing as a branch in its consequent. This, of course is the branch chosen by mini-FORLOG. This mini-FORLOG step then replaces the u_i's with the c_i's found in $p(c_1,...,c_n)$, the chosen assertion (step 1a), and uses Sk to Skolemize the branch. The result being a conjunction of equalities and literals

$$(c_1{=}t_1') \wedge ... \wedge (c_n{=}t_n') \wedge L_1' \wedge ... \wedge L_k',$$

where t_i' and L_i' are Skolemized versions of t_i and L_i respectively.

Steps 1a and 1b involve making nondeterministic choices. We have shown that for the nondeterministic choice of a goal and clause in BACKLOG there is a corresponding choice of an assertion and branch of a disjunction. The reverse holds as well.

2. Here we want to show that the most general unifier computed by BACKLOG corresponds to mini-FORLOG's simplification of equalities in the branch selected in the previous step.

   Clark (1978, p. 306) proved the following lemma. Under the restricted theory of equality (section 2.2), $p(b_1,...,b_n)$ can be unified with $p(t_1,...,t_n)$ with most general unifier

   $$s = \{(x_1=e_1),...,(x_k=e_k)\},$$

   where each $(x_i=e_i)$ represents the substitution of e_i for x_i, if and only if the conjunction of equalities

   $$(b_1=t_1) \wedge ... \wedge (b_n=t_n)$$

   can be simplified to yield the set of equalities

   $$s' = \{(x_1=e_1),...,(x_k=e_k)\}.$$

   Syntactically s and s' are identical, but semantically they differ.

   Applying the Sk mapping to s' and to the conjunction of equalities above certainly will not affect the equivalency. Therefore, we have

   $$(c_1=t_1') \wedge ... \wedge (c_n=t_n')$$

   equivalent to

   $$\mathrm{Sk}(s') = \{(sk_1=e_1'),...,(sk_k=e_k')\},$$

   where x_i was mapped to the Skolem constant sk_i, and c_i, t_i', and e_i' are Skolemized versions of b_i, t_i and e_i respectively. This equivalency defines the most general equalities sought by mini-FORLOG, we therefore have

   $$e = \mathrm{Sk}(s) = \{(sk_1=e_1'),...,(sk_k=e_k')\}.$$

   Each $(sk_i=e_i')$ is an equality.

3. Here we wish to demonstrate that composing s and $subs_i$ corresponds to composing e and $equals_i$.

   We know that $equals_m=Sk(subs_m)$ and $e=Sk(s)$. Since substitutions and equalities have the same syntactic form, BACKLOG and mini-FORLOG can use the same Compose procedure. In addition, because of the result in step 2 above, we can treat the equalities just like substitutions. That is, each $(sk_i=e_i') \in e$ can represent the substitution of e_i' for sk_i. The left hand side of an equality in e is always a Skolem constant and the right hand side is some term. From

   $$subs_{m+1} = \mathrm{Compose}(s,subs_m)$$
   $$equals_{m+1} = \mathrm{Compose}(e,equals_m)$$

   we have the following:

   $$\begin{aligned} Sk(subs_{m+1}) &= Sk(\mathrm{Compose}(s,subs_m)) \\ &= \mathrm{Compose}(Sk(s),Sk(subs_m)) \\ &= \mathrm{Compose}(e,equals_m) \\ &= equals_{m+1} \end{aligned}$$

4. Now we want to show that removing $p(b_1,...,b_n)$ from, and adding $L_1, ..., L_k$ to, *goal-list*$_m$ in BACKLOG and removing $p(c_1,...,c_n)$ from, and adding $Sk(L_1,...,L_k)$ to, *assertion-list*$_m$ in mini-FORLOG maintains the correspondence. That is, from

   $$g = goal\text{-}list_m - p(b_1,...,b_n) + (L_1,...,L_k)$$
   $$a = assertion\text{-}list_m - p(c_1,...,c_n) + Sk(L_1,...,L_k)$$

   we have the following:

   $$\begin{aligned} Sk(g) &= Sk(goal\text{-}list_m - p(b_1,...,b_n) + (L_1,...,L_k)) \\ &= Sk(goal\text{-}list_m) - Sk(p(b_1,...,b_n)) + Sk(L_1,...,L_k) \\ &= assertion\text{-}list_m - p(c_1,...,c_n) + Sk(L_1,...,L_k) \\ &= a \end{aligned}$$

5. Finally we have the correspondence between computing *goal-list*$_{m+1}$ in BACKLOG and computing *assertion-list*$_{m+1}$ in mini-FORLOG.

   We know that $e=Sk(s)$ and $Sk(g)=a$. Therefore, from

   $$goal\text{-}list_{m+1} = \text{Apply}(s,g)$$
   $$assertion\text{-}list_{m+1} = \text{Apply}(e,a)$$

   we have the following:

   $$\begin{aligned} Sk(goal\text{-}list_{m+1}) &= Sk(\text{Apply}(s,g)) \\ &= \text{Apply}(Sk(s),Sk(g)) \\ &= \text{Apply}(e,a) \\ &= assertion\text{-}list_{m+1} \end{aligned}$$

Therefore, by induction, the states maintain their correspondence. Because of this, when the *goal-list*$_i$ is empty (for some i), *assertion-list*$_i$ is empty as well, and $equals_i=Sk(subs_i)$. Further, since the start assertion for mini-FORLOG was $Sk(G)$, where G was BACKLOG's original goal, Sk provides the mapping from the output of BACKLOG and the output of mini-FORLOG. With this, we now have proven that the two strategies are procedurally isomorphic ∎

This somewhat surprising result means that backward-chaining and forward-chaining are alternative methods for searching through different formulations of the same problem space.

Chapter 3
Programming with Disjunctions

Recall that the mini-FORLOG interpreter described previously did not specify how to select which branch of the disjunction to pursue. We discovered that mini-FORLOG can mirror Prolog's depth-first computation strategy. Full FORLOG provides a more flexible method for dealing with this nondeterministic choice. When it encounters an implication with a disjunctive consequent, FORLOG *compiles* it into logically sound implications. These implications try to identify which branches lead to solutions. When no identification can be made, each branch is pursued by assuming that it is the "correct" one.

This implementation of nondeterminism requires FORLOG to track assumptions. These assumptions are fundamentally different from assertions. Any given assumption could be contradictory alone or in conjunction with other assumptions made by FORLOG. To maintain a consistent database of assertions and assumptions, FORLOG employs a truth maintenance system (TMS). The TMS also provides support for deciding the validity of negated literals.

3.1 Compiling Disjunctions

When FORLOG encounters an implication with a disjunctive consequent, it compiles the implication into several simpler ones, none of which contain a disjunction. This operation, performed once at compile time, derives implications of the form

$$\forall\ x_1,\dots,x_n\ A_1 \wedge A_2 \wedge \dots \wedge A_m \supset C$$

where each A_i is some literal or equality (possibly negative) and C is an existentially quantified conjunction of equalities and literals.

Notice that the implication $p \supset q \vee r$ could be rewritten as $p \wedge \neg q \supset r$. This is exactly what FORLOG does, except it enumerates all implications. For an example, examine the following implication:

$$\forall\, x\ p(x) \supset D_1 \vee D_2 \vee \ldots \vee D_n.$$

The number of parameters in predicate p is irrelevant for this discussion, so a single one will suffice for illustrative purposes. FORLOG compiles this implication into the implications

$$\begin{array}{l} \forall\, x\ [p(x) \wedge \neg D_2 \wedge \neg D_3 \wedge \ldots \wedge \neg D_n] \supset D_1 \\ \forall\, x\ [p(x) \wedge \neg D_1 \wedge \neg D_3 \wedge \ldots \wedge \neg D_n] \supset D_2 \\ \quad\quad\cdot \\ \quad\quad\cdot \\ \quad\quad\cdot \\ \forall\, x\ [p(x) \wedge \neg D_1 \wedge \neg D_2 \wedge \ldots \wedge \neg D_{n-1}] \supset D_n. \end{array}$$

The above implications look to be in the desired form, but they, however, do not complete the process. Recall that each branch of the disjunction is a conjunction of existentially quantified literals and equalities (actually any disjunction could contain an imbedded disjunction, which just means that this method must be applied recursively). During the compilation process, equalities are treated exactly like literals. Let the conjunction

$$\text{(3-1)} \quad \exists\, y_i\ L_{i,1} \wedge L_{i,2} \wedge \ldots \wedge L_{i,m}$$

represent D_i, where each $L_{i,j}$ is ground or contains the universal variable x and/or the existential variable y_i. Again, the number of parameters is irrelevant. Negating a branch turns it into a universally quantified disjunction of negated literals (deMorgan's law). If D_i contains m literals as in (3-1), $\neg D_i$ becomes

$$\forall\, y_i\ \neg L_{i,1} \vee \neg L_{i,2} \vee \ldots \vee \neg L_{i,m}.$$

With each negated branch expanded with its literals, the implications above are not particularly useful. If, however, the antecedents were in disjunctive normal form, each implication could be divided into several simpler implications (e.g. $p \vee q \supset r$ can be rewritten as the two implications $p \supset r$ and $q \supset r$).

With $n=3$, we will use the implication

$$\forall\ x\ [p(x) \wedge \neg D_2 \wedge \neg D_3] \supset D_1$$

to illustrate the process of putting an antecedent into disjunctive normal form. Expanding the negated branches $\neg D_2$ and $\neg D_3$, by putting the literals in the antecedent gives us

$$\forall\ x y_1 y_2\ [p(x) \wedge (\neg L_{2,1} \vee \neg L_{2,2}) \wedge (\neg L_{3,1} \vee \neg L_{3,2})] \supset D_1.$$

The right-most conjunction can be distributed to produce

$$\begin{aligned}\forall\ x y_1 y_2\ [p(x) \wedge ((\neg L_{2,1} \wedge \neg L_{3,1}) \vee (\neg L_{2,1} \wedge \neg L_{3,2}) \vee \\ (\neg L_{2,2} \wedge \neg L_{3,1}) \vee (\neg L_{2,2} \wedge \neg L_{3,2}))] \supset D_1\end{aligned}$$

which amounts to taking the cross product between each branch in the antecedent. Distributing $p(x)$ amongst the literals, we get the antecedent in disjunctive normal form (not shown here). From this we derive the four implications

$$\begin{aligned}&\forall\ x y_1 y_2\ [p(x) \wedge \neg L_{2,1} \wedge \neg L_{3,1}] \supset D_1\\ &\forall\ x y_1 y_2\ [p(x) \wedge \neg L_{2,1} \wedge \neg L_{3,2}] \supset D_1\\ &\forall\ x y_1 y_2\ [p(x) \wedge \neg L_{2,2} \wedge \neg L_{3,1}] \supset D_1\\ &\forall\ x y_1 y_2\ [p(x) \wedge \neg L_{2,2} \wedge \neg L_{3,2}] \supset D_1.\end{aligned}$$

This process is repeated for the other two implications with D_2 and D_3 as consequents, deriving a total of twelve implications (we will not list them here, but the process is identical).

Even though the example showed the branches D_2 and D_3 with two literals each, they don't necessarily have to have the same number of literals. In general, we must take the cross product of $n-1$ varied length vectors to produce at most k^{n-1} new vectors where k is the length of the branch containing the largest number of literals. Each of the new vectors contains $n-1$ literals which are conjoined together. This must be done for each implied branch, that is n times. This may seem rather expensive (it is exponential), but it is only performed once for an assertion and n is usually small.

3.1.1 Example: Compiling *delete*

The FORLOG *delete* program discussed earlier contained a disjunction in its consequent

$$\forall\ u_1,u_2,u_3\quad delete(u_1,u_2,u_3)\ \supset\ \exists\ x_1,x_2\ \ (u_1=x_1) \wedge (u_2=[x_1|x_2]) \wedge (u_3=x_2)$$
$$\vee$$
$$\exists\ x_1,x_2,x_3,x_4\ (u_1=x_1) \wedge (u_2=[x_2|x_3]) \wedge (u_3=[x_2|x_4]) \wedge delete(x_1,x_3,x_4).$$

Compiling this rule produces the following implications:

$$\forall\ u_1,u_2,u_3\quad br_1(u_1,u_2,u_3)\ \supset\ \exists\ x_1,x_2\ \ (u_1=x_1) \wedge (u_2=[x_1|x_2]) \wedge (u_3=x_2)$$
$$\forall\ u_1,u_2,u_3\quad br_2(u_1,u_2,u_3)\ \supset\ \exists\ x_1,x_2,x_3,x_4\ (u_1=x_1) \wedge (u_2=[x_2|x_3]) \wedge (u_3=[x_2|x_4]) \wedge delete(x_1,x_3,x_4)$$

$$\forall\ u_1,u_2,u_3,x_1\quad delete(u_1,u_2,u_3) \wedge \neg(u_1=x_1)\ \supset\ br_2(u_1,u_2,u_3)$$
$$\forall\ u_1,u_2,u_3,x_1,x_2\quad delete(u_1,u_2,u_3) \wedge \neg(u_2=[x_1|x_2])\ \supset\ br_2(u_1,u_2,u_3)$$
$$\forall\ u_1,u_2,u_3,x_2\quad delete(u_1,u_2,u_3) \wedge \neg(u_3=x_2)\ \supset\ br_2(u_1,u_2,u_3)$$
$$\forall\ u_1,u_2,u_3,x_1\quad delete(u_1,u_2,u_3) \wedge \neg(u_1=x_1)\ \supset\ br_1(u_1,u_2,u_3)$$
$$\forall\ u_1,u_2,u_3,x_2,x_3\quad delete(u_1,u_2,u_3) \wedge \neg(u_2=[x_2|x_3])\ \supset\ br_1(u_1,u_2,u_3)$$
$$\forall\ u_1,u_2,u_3,x_2,x_4\quad delete(u_1,u_2,u_3) \wedge \neg(u_3=[x_2|x_4])\ \supset\ br_1(u_1,u_2,u_3)$$
$$\forall\ u_1,u_2,u_3,x_1,x_3,x_4\quad delete(u_1,u_2,u_3) \wedge \neg delete(x_1,x_3,x_4)\ \supset\ br_1(u_1,u_2,u_3)$$

$$\forall\ u_1,u_2,u_3\quad delete(u_1,u_2,u_3) \wedge enumerate\ \supset\ br_1(u_1,u_2,u_3) \vee br_2(u_1,u_2,u_3).$$

The first two implications serve to assert the original literals involved in each branch when a particular branch is ruled in. The next seven implications attempt to decide which branch is valid. The final rule enumerates the branches if and only if no progress was gained with the previous implications. The *enumerate* predicate gives FORLOG control over when this happens. Enumeration *assumes* that each branch is viable. This may turn out to be a false assumption; FORLOG must then handle these derived contradictions.

Notice that some additional syntax has been added to the method described. Each branch of the disjunction is in fact associated with a unique name (i.e. br_1 and br_2). This was introduced to avoid repeating the entire branch formula in the second set of implications, and to simplify the enumeration process.

3.2 Handling Contradictions: Using a Truth Maintenance System

The abstract mini-FORLOG interpreter did not have to handle contradictions because the choice of which disjunction to pursue was a nondeterministic one. The interpreter simply chose the "right" disjunction to follow. In reality, FORLOG must enumerate branches of a disjunction to achieve this nondeterministic behavior. Often, as a result of this enumeration, it derives a contradiction. This inconsistency may be discovered immediately while simplifying equalities or derived later from some deduction involving a literal in a branch.

In classical, monotonic logic, the presence of an inconsistency renders all deductions meaningless—any assertion may be proven (or disproven). If, however, we employ nonmonotonic logic and retract the assertions affected by the contradiction, other assertions unaffected by the contradiction remain valid.

Enumerating the branches of a disjunction requires FORLOG to assume that each branch is valid, otherwise it would not have to enumerate. This may in fact be a false assumption. Often pursuing a particular branch leads to a contradiction. For example, with the implications

$$p \supset q \vee r, \quad r \supset \perp$$

where $\perp$ designates a contradiction, FORLOG would compile the first one into the several implications:

$$\begin{aligned}&branch_1 \supset q\\ &branch_2 \supset r\\ &p \wedge \neg q \supset r\\ &p \wedge \neg r \supset q\\ &p \wedge enumerate \supset branch_1 \vee branch_2.\end{aligned}$$

Asserting the predicate p, with the reformulation, does not allow FORLOG to derive anything further. Therefore, it enumerates by assuming both $branch_1$ and $branch_2$. From $branch_1$ FORLOG deduces q and from $branch_2$ it deduces r. Remember that $r \supset \perp$. Since FORLOG deduces r, it also deduces $\perp$. Assuming the validity of $branch_2$ led to a contradiction. To remove the inconsistency, we could remove everything affected by this assumption (i.e. the predicate r).

Unfortunately, there are other ways to derive contradictions. I borrow an example from deKleer (1986a). Examine the following assertions:

$$\begin{array}{ll} a, & c, \\ a \supset b, & c \supset d, \end{array}$$

$$a \wedge c \supset e, \qquad b \wedge d \supset \perp.$$

Figure 3.1 describes the set of assertions graphically. This set of assertions is inconsistent because it allows the derivation of $\perp$. Unfortunately, everything is affected by the contradiction, it cannot be removed short of retracting an assertion, such as $c \supset d$ or $b \wedge d \supset \perp$.

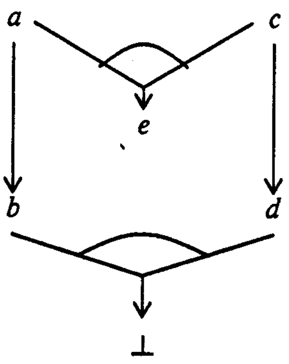

Figure 3.1: Graph representing the assertions.

In general, it is costly to determine assertions affected by a contradiction. This is exacerbated by the problem of selecting which of the affected assertions to retract. Fortunately, there are reasonable ways to overcome these fundamental difficulties. FORLOG employs an assumption-based truth maintenance system (ATMS) (deKleer, 1986a, b, c) to maintain the consistency of its database of assertions and to achieve nonmonotonicity in the face of derived contradictions.

Now consider the alternative formulation for the second example:

$$\begin{array}{lll} :MA/A, & :MC/C, & \\ A \supset b, & C \supset d, & A \wedge C \supset e, \end{array}$$

where M is Reiter's (1980) default reasoning operator. In this theory, $:MA/A$ translates to "in the absence of any information to the contrary, assume A." If A somehow led to a contradiction, we have the contrary information. By convention, capital letters represent default assumptions. We already used this assumption idea when enumerating the branches of a disjunction. Now, any assertion may be implied by an assumption. Since it consistent to assume both A and C, we may derive b, d, and e. The set $\{A,b,C,d,e\}$ is called an *extension* of the theory.

Adding the implication $b \wedge d \supset \perp$ has a different effect than it did in the original formulation. As before, this introduces a contradiction, but this contradiction affects only e, nothing else. We cannot assume both A and C simultaneously. We may assume A and derive b or assume C and derive d, but not both. In formal terms, the theory now has two mutually exclusive extensions: $\{A,b\}$ and $\{C,d\}$. The contradiction split the original extension in two.

The ATMS implements the behavior suggested by this example. It maintains with each assertion, the default assumptions used to derive it. The tuple

$$\langle fact, \{S_1,...,S_n\}\rangle$$

states that *fact* is supported by the assumption sets $S_1,...,S_n$. Logically, each S_i forms a separate conjunction that implies the assertion. The set of sets $\{S_1,...,S_n\}$ is called a label. If, for any reason, one of the S_i's contains assumptions that cannot hold simultaneously, that S_i is removed from *fact*'s label.

Looking at the final state of the example, the following tuples describe all known assertions:

$$\langle b, \{\{A\}\}\rangle, \quad \langle d, \{\{C\}\}\rangle, \quad \langle e, \{\ \}\rangle.$$

Notice that e has no default assumption set (i.e. an empty label), which means that e is false. FORLOG has the important feature that while an assertion is false it cannot be used in any deduction. In fact, the ATMS makes sure that any assertion justified solely by other assertions that are false is itself false.

It is also possible for an assertion to be supported by the null assumption, making the assertion a premise. For example, the assertion

$$\langle f, \{\{\ \}\}\rangle$$

indicates that f is a premise and must be true and not simply assumed true.

Using ATMS terminology, deKleer calls default assumptions that cannot hold simultaneously *nogoods.* The ATMS maintains a list of all nogoods derived or asserted. In the example, adding the final axiom causes the ATMS to derive the nogood $\{A,C\}$. This nogood simply asserts $\neg(A \wedge C)$ or alternatively $(\neg A \vee \neg C)$. As mentioned above, the ATMS updates the labels of assertions by removing contexts that are supersets of any nogood. For example, e started out as the assertion $\langle e, \{\{A,C\}\}\rangle$ but when $\{A,C\}$ was found to be nogood, it was updated to be $\langle e, \{\ \}\rangle$.

Assertions inherit labels from the assertions used to derive them. With the implication

(3-1) $p \wedge q \supset r$

and the assertions

$$\langle p,\{\{F\}\}\rangle,\ \langle q,\{\{G\}\}\rangle,$$

FORLOG derives

(3-2) $\langle r,\{\{F,G\}\}\rangle$.

Logically, from $p \wedge q \supset r$, $F \supset p$, and $G \supset q$ we can derive $F \wedge G \supset r$, which is exactly what the final tuple represents.

Suppose that, by some other means, FORLOG deduces the assertions

(3-3) $\langle p,\{\{M\}\}\rangle,\ \langle q,\{\{N\}\}\rangle$.

FORLOG, because of the ATMS, knows that r has previously been derived. These new assertions simply provide additional support for believing r. Rather than applying the rule (3-1) again, the ATMS propagates labels from p and q to r. It does this by recording not only assumptions but justifications as well. In the example, the ATMS knew that p and q were used to derive r. Therefore, it updates r's label to reflect the new support provided by the assertions in (3-3). This updating changes (3-2) to be

$$\langle r,\{\{F,G\},\{M,N\}\}\rangle.$$

In summary, an ATMS acts as a sophisticated database system for recording not only assertions, but also their labels (underlying assumptions). It provides some benefits in addition to those provided by a conventional truth maintenance system. First, it allows the system to store multiple, mutually inconsistent views of a single database efficiently and correctly. These different views may be explored in any manner, depth-first or breadth-first. Another benefit of an ATMS is that it provides support for intelligent backtracking by determining which assumptions contribute to a contradiction.

3.3 Handling Negations

Compiling an implication with a disjunctive consequent introduces negations to FORLOG. Negative literals have historically caused plenty of problems for logic programming. Our implementation relies upon negations, so we must have a sound and effective method for using them.

Prolog employs the unsound method of negation-as-failure to implement negations of literals. This method closes the database of facts by assuming that all the true facts are known. Any facts not present are false. FORLOG employs an alternative method by supporting several kinds of negated literals.

Except for equality literals, the user must inform FORLOG about a literal and its type. The different types available are:

- Test literals. The user provides a boolean Lisp function to call when the parameters of a test literal are bound.
- Functional literals. These are still predicates not functions. The user gives one or more Lisp functions that establish a functional dependency between the parameters of the given literal.
- Functional dependency literals. The user asserts that some parameters of the literal are functionally determined by other parameters.

Any literal not defined to be one of the above is known as a simple literal. Each type of literal employs a different but sound method to determine the truth value of its negation. We will examine each one individually.

3.3.1 Negating equality literals

Any given branch of a disjunction often contains several equalities. As we saw with the compilation of the program for *delete*, these equalities end up negated in antecedents of some implications.

It turns out that some of these may be discarded. For example, the compiled implication

$$\forall\ u_1,u_2,u_3,x_1\quad delete(u_1,u_2,u_3) \land \neg(u_1=x_1) \supset br_2(u_1,u_2,u_3)$$

could never detach the branch br_2. When the variable u_1 gets bound by a particular instantiation of the *delete* predicate, it will always unify with the universal variable x_1. Because of the equality theory employed, the equality $(u_1=x_1)$ will always be true, therefore the inequality $\neg(u_1=x_1)$ will always be false. FORLOG avoids creating implications containing negations of this type.

Examine another implication generated by the compilation of *delete*:

$$\forall\ u_1,u_2,u_3,x_1,x_2\quad delete(u_1,u_2,u_3) \land \neg(u_2=[x_1|x_2]) \supset br_2(u_1,u_2,u_3).$$

In this case, whenever u_2 gets bound to a list by a particular instantiation of *delete*, the equality $(u_2=[x_1|x_2])$ can be satisfied, thus making the inequality $\neg(u_2=[x_1|x_2])$ false. So, whenever the bound value for u_2 unifies with the list $[x_1|x_2]$, the inequality is false, otherwise the inequality is true.

In general FORLOG uses the success or failure of unification to evaluate an inequality. If the left side of the equality unifies with the right, then the inequality is false. If the unification fails, then the inequality is true.

3.3.2 Negating test literals

With FORLOG, a literal having an attached boolean Lisp function can be defined. An instance of a test literal (positive or negative) must be bound before it is evaluated. To evaluate a negative test literal, the bound parameters are passed to the attached Lisp function. If the function returns T, the negation is false. If, on the other hand, the function returns NIL, the negation is true.

For example, suppose we had the implication

$$\forall\ x,y\ p(x,y) \wedge \neg greater(x,y) \supset q(x,y)$$

where p and q are any predicates and *greater* is attached to the Interlisp-D function *IGREATERP*. With the assertion

$$\langle p(3,10),\ \{\{A\}\}\rangle$$

the universal variables, x and y, in the implication get bound. This permits the literal $\neg greater(3,10)$ to be evaluated. FORLOG runs *IGREATERP*(3,10). Since 3 is not greater than 10, the lisp function returns false, meaning that $\neg greater(3,10)$ is true. Therefore, using modus ponens, FORLOG deduces

$$\langle q(3,10),\ \{\{A\}\}\rangle.$$

3.3.3 Negating functional literals

Not all attached functions in FORLOG must be boolean. Literals can have attached lisp functions of any type. These attached functions define functional dependencies among a literal's parameters. As with test literals, the parameters for a negated functional literal must be bound before it can be evaluated.

For the functional literal $r(x,y_1,y_2,...,y_n)$, let x be the parameter functionally determined by the parameters $y_1,...,y_n$ as defined by f, an n-place Lisp function attached to r. Examine the implication

$$\forall\ y_1,y_2,...,y_n\ p(y_1,y_2,...,y_n) \supset \exists\ x\ r(x,y_1,y_2,...,y_n).$$

Suppose we had the assertion

$$\langle p(c_1,c_2,...,c_n),\{\{B\}\}\rangle$$

where the $c_1,c_2,...,c_n$ are ground terms. FORLOG would use the attached function $f(c_1,c_2,...,c_n)$ to compute a value for the existential variable x and assert

$$\langle r(d,c_2,...,c_n),\{\{B\}\}\rangle$$

where $d=f(c_1,c_2,...,c_n)$.

To determine the truth value of a negated functional literal, $\neg r(x,y_1,y_2,...,y_n)$, FORLOG tests the equality between x and the value returned by $f(y_1,y_2,...,y_n)$. If the two are equal, the negated literal is false, otherwise it is true.

For example, suppose we had the implication

$$\forall\ x,y,z\ p(x,y,z) \land \neg plus(x,y,z) \supset q(x,y,z)$$

where p and q are any predicates and *plus* has the Interlisp-D function *IPLUS* defining a functional dependency between x, y and z (i.e. $x+y=z$). With the assertion

$$\langle p(5,7,2),\ \{\{B\}\}\rangle,$$

the universal variables in the implication get bound. The literal $\neg plus(5,7,2)$ can be evaluated. FORLOG runs *IPLUS*(5,7) returning 12. Since this value not equal to 2, $\neg plus(5,7,2)$ is true. Therefore, using modus ponens, FORLOG deduces

$$\langle q(5,7,2),\ \{\{B\}\}\rangle.$$

3.3.4 Negating functional dependency literals

Functional dependencies among the parameters of a literal without any explicit attached function can be declared. Here again, the parameters must be bound before a literal of this type is evaluated.

For a literal h, let $x_1,x_2,...,x_m$ be the parameters functionally determined by the parameters $y_1,y_2,...,y_n$. FORLOG enforces the following axiom for such a literal:

$$\forall\ x_1,...,x_m,z_1,...,z_m,y_1,...,y_n\ [h(x_1,...,x_m,y_1,...,y_n) \land h(z_1,...,z_m,y_1,...,y_n)] \supset (x_1=z_1) \land ... \land (x_m=z_m)$$

In this case, negations are true with respect to the contexts in which assertions of h are made. Examine the assertion of the positive literal

$$\langle h(x_1,...,x_m,y_1,...,y_n),\ \{\{C\}\}\rangle$$

and the assertion

$$\langle h(z_1,...,z_m,y_1,...,y_n),\ \{\{D\}\}\rangle$$

where $y_1,...,y_n$ are the exact same (bound) parameters as above, but at least one $z_i \neq x_i$. FORLOG uses the defined functional dependency to deduce that the context $\{C,D\}$ is nogood. Which also means that in context $\{C\}$ $\neg h(z_1,...,z_m,y_1,...,y_n)$ is true, and in context $\{D\}$ $\neg h(x_1,...,x_m,y_1,...,y_n)$ is true.

For example, examine the implication

$$\forall\ x,y\quad \neg father(x,y) \supset q(x,y)$$

where q is any predicate, and the literal *father* states that x is the father of y. Since everyone has only one father, y functionally determines x. But, any one father can have several children, so x does not functionally determine y. With the assertion

$$\langle father(Doug,Dan),\ \{\{F\}\}\rangle$$

nothing else gets deduced. Adding the additional assertion

$$\langle father(Steve,Dan),\ \{\{G\}\}\rangle$$

causes FORLOG to deduce the nogood $\{F,G\}$ and the assertions

$$\langle q(Steve,Dan),\ \{\{F\}\}\rangle, \qquad \langle q(Doug,Dan),\ \{\{G\}\}\rangle.$$

Now, we have two valid extensions of the database of assertions: $\{father(Doug,Dan), q(Steve,Dan)\}$ under context $\{F\}$ and $\{father(Steve,Dan), q(Doug,Dan)\}$ under context $\{G\}$.

Asserting

$$\langle father(Doug,Karl),\ \{\{H\}\}\rangle$$

does not cause any extra deductions to be made, because it fails to violate the functional dependency (i.e. *Doug* can have several children).

3.3.5 Negating simple literals

With simple literals, no attached functions exist nor are there any dependencies among parameters. Without these, we cannot evaluate the truth of a negated predicate, unless it is explicitly derived or asserted as a fact. In general, there is no solution to this problem other than resorting to Prolog's method of negation-as-failure.

As an example, examine the implication

(3-4) $\forall\ x,y\ \neg brother(x,y) \supset q(x,y)$

where q is any predicate and *brother* asserts that x is the brother of y. Since anyone could have several brothers, we have no functional dependency between the two parameters. With the assertion

$\langle brother(Dan,Nick),\ \{\{I\}\}\rangle$

nothing else gets deduced. Adding the assertion

$\langle brother(Dan,Karl),\ \{\{J\}\}\rangle$

is perfectly consistent with known assertions. Rule (3-4) would never detach its consequent unless a specific $\neg$*brother* assertion was made.

3.4 Enumerating a Disjunction

When FORLOG has to fall into enumerating the branches of a disjunction, it employs the ATMS to maintain them in separate contexts.

Given the implication

$\forall\ x\ p(x) \supset q(x) \vee r(x),$

FORLOG compiles it into the following:

$$
\begin{aligned}
&\forall\ x\ p(x) \wedge branch_1(x) \supset q(x)\\
&\forall\ x\ p(x) \wedge branch_2(x) \supset r(x)\\
&\forall\ x\ p(x) \wedge \neg r(x) \supset branch_1(x)\\
&\forall\ x\ p(x) \wedge \neg q(x) \supset branch_2(x)\\
&\forall\ x\ p(x) \wedge \mathit{enumerate} \supset branch_1(x) \vee branch_2(x).
\end{aligned}
$$

Using the ATMS notation, suppose we made the assertion $\langle p(a),\ \{\{A\}\}\rangle$. FORLOG has no way of ruling in either branch, since it knows nothing about $r(a)$ or $q(a)$. So it employs enumeration. The special system predicate, *enumerate*, causes FORLOG to introduce assumptions for each branch to keep them disjunctive. So we get the assertions

$$\langle branch_1(a),\ \{\{A,B_1\}\}\rangle, \qquad \langle branch_2(a),\ \{\{A,B_2\}\}\rangle$$

where the assumptions B_1 and B_2 are the ones introduced for $branch_1(a)$ and $branch_2(a)$ respectively. Now, FORLOG can apply modus ponens with the first two compiled implications, deriving

$$\langle q(a),\ \{\{A,B_1\}\}\rangle, \qquad \langle r(a),\ \{\{A,B_2\}\}\rangle.$$

This gives us the single extension $\{p(a),\ q(a),\ r(a)\}$ under the context $\{A,\ B_1,\ B_2\}$.

Imagine adding the implication

$$\forall\ x\ q(x) \land r(x) \ \supset\ \perp.$$

This introduces a contradiction into our database. Now all three assumptions cannot hold simultaneously (i.e. $\{A,B_1,B_2\}$ is nogood). The implication splits the original extension into two mutually exclusive ones: $\{p(a),\ q(a)\}$ under the context $\{A,\ B_1\}$ and $\{p(a),\ r(a)\}$ under the context $\{A,\ B_2\}$.

Chapter 4
Infinite Computations

Certain problems, when given to Prolog, never terminate. For example, the program that declares the reflexive property of a sibling relationship

$$sibling(x,y) \text{ :- } sibling(y,x).$$

would never terminate when given any goal (e.g. *sibling(Dan,Karl)*).

FORLOG does not have this particular problem. Any instantiation of an implication is only used once. The FORLOG equivalent to the above program is:

(4-1) $\forall\ x,y\ sibling(x,y) \supset sibling(y,x).$

Given the assertion $\langle sibling(Dan,Karl),\{\{A\}\}\rangle$, FORLOG would deduce $\langle sibling(Karl,Dan),\{\{A\}\}\rangle$. This last assertion would cause FORLOG to use the rule once more to deduce again the first assertion. But since this assertion was already used once with rule (4-1), it is not used again. FORLOG caches every assertion in the ATMS to avoid infinite computation such as this.

Unfortunately, other programs that cause problems for Prolog cause problems for FORLOG as well. This stems mainly from FORLOG's eagerness to derive new consequences of known facts. Resolution, a function performed by the ATMS upon known assumptions, provides a method for terminating some infinite computations.

4.1 Examining the Problem Through an Example

Naish (1985b) used a program defining the permutation relation on lists to illustrate the need to add flexible control rules to Prolog. This program

$$perm([\,],[\,]).$$
$$perm([x_1|x_2],[x_3|x_4]) \text{ :- } perm(x_5,x_4) \wedge delete(x_3,[x_1|x_2],x_5).$$

needs a different ordering of its subgoals depending upon which argument is bound. The *delete* predicate is the same one discussed previously. If *perm* is called with the second argument a variable, the execution of *delete* should proceed ahead of *perm*. If, on the other hand, the first argument is a variable, *perm* should proceed ahead of *delete*. Nontermination is the consequence of not following this strategy.

Given the forward, mini-FORLOG version of this program

$$\begin{aligned}\forall\, u_1,u_2\ perm(u_1,u_2) \supset\ & (u_1=[\,]) \wedge (u_2=[\,]) \\ & \vee \\ & \exists\, x_1,x_2,x_3,x_4,x_5\ (u_1=[x_1|x_2]) \wedge (u_2=[x_3|x_4]) \wedge \\ & \quad perm(x_5,x_4) \wedge delete(x_3,[x_1|x_2],x_5),\end{aligned}$$

FORLOG would compile it into the following implications:

$$\begin{aligned}&\forall\, u_1,u_2\ p\text{-}br1(u_1,u_2) \supset (u_1=[\,]) \wedge (u_2=[\,])\\ &\forall\, u_1,u_2\ p\text{-}br2(u_1,u_2) \supset \exists\, x_1,x_2,x_3,x_4,x_5\ (u_1=[x_1|x_2]) \wedge (u_2=[x_3|x_4]) \wedge\\ &\qquad perm(x_5,x_4) \wedge delete(x_3,[x_1|x_2],x_5)\end{aligned}$$

$$\begin{aligned}&\forall\, u_1,u_2\ perm(u_1,u_2) \wedge \neg(u_1=[\,]) \supset p\text{-}br2(u_1,u_2)\\ &\forall\, u_1,u_2\ perm(u_1,u_2) \wedge \neg(u_2=[\,]) \supset p\text{-}br2(u_1,u_2)\\ &\forall\, u_1,u_2,x_1,x_2\ perm(u_1,u_2) \wedge \neg(u_1=[x_1|x_2]) \supset p\text{-}br1(u_1,u_2)\\ &\forall\, u_1,u_2,x_3,x_4\ perm(u_1,u_2) \wedge \neg(u_2=[x_3|x_4]) \supset p\text{-}br1(u_1,u_2)\end{aligned}$$

$$\forall\, u_1,u_2\ perm(u_1,u_2) \wedge p\text{-}enumerate \supset p\text{-}br1(u_1,u_2) \vee p\text{-}br2(u_1,u_2)$$

The compiled version of the FORLOG delete program of the previous section, adds the following (slightly rewritten) implications:

$$\begin{aligned}&\forall\, u_1,u_2,u_3\ \ d\text{-}br1(u_1,u_2,u_3) \supset \exists\, x_1,x_2\ \ (u_1=x_1) \wedge (u_2=[x_1|x_2]) \wedge (u_3=x_2)\\ &\forall\, u_1,u_2,u_3\ \ d\text{-}br2(u_1,u_2,u_3) \supset \exists\, x_1,x_2,x_3,x_4\ (u_1=x_1) \wedge (u_2=[x_2|x_3]) \wedge\\ &\qquad (u_3=[x_2|x_4]) \wedge\ delete(x_1,x_3,x_4)\end{aligned}$$

$$\begin{aligned}&\forall\, u_1,u_2,u_3,x_1,x_2\ \ delete(u_1,u_2,u_3) \wedge \neg(u_2=[x_1|x_2]) \supset d\text{-}br2(u_1,u_2,u_3)\\ &\forall\, u_1,u_2,u_3,x_2,x_3\ \ delete(u_1,u_2,u_3) \wedge \neg(u_2=[x_2|x_3]) \supset d\text{-}br1(u_1,u_2,u_3)\\ &\forall\, u_1,u_2,u_3,x_2,x_4\ \ delete(u_1,u_2,u_3) \wedge \neg(u_3=[x_2|x_4]) \supset d\text{-}br1(u_1,u_2,u_3)\\ &\forall\, u_1,u_2,u_3\ \ delete(u_1,u_2,u_3) \wedge d\text{-}enumerate \supset d\text{-}br1(u_1,u_2,u_3) \vee\\ &\qquad d\text{-}br2(u_1,u_2,u_3)\end{aligned}$$

When discussing negated literals, we discovered that some could never be true. Notice that the implications that contained such negated literals have not been included in the ones above.

Examine the deductions produced as a result of making the assertion

$$\langle perm([1],y),\ \{\{\}\}\rangle$$

with Skolem constant y in the null context (see Figure 4.1). Recall that the null assumption set means that the assertion is true in all contexts; it is a premise.

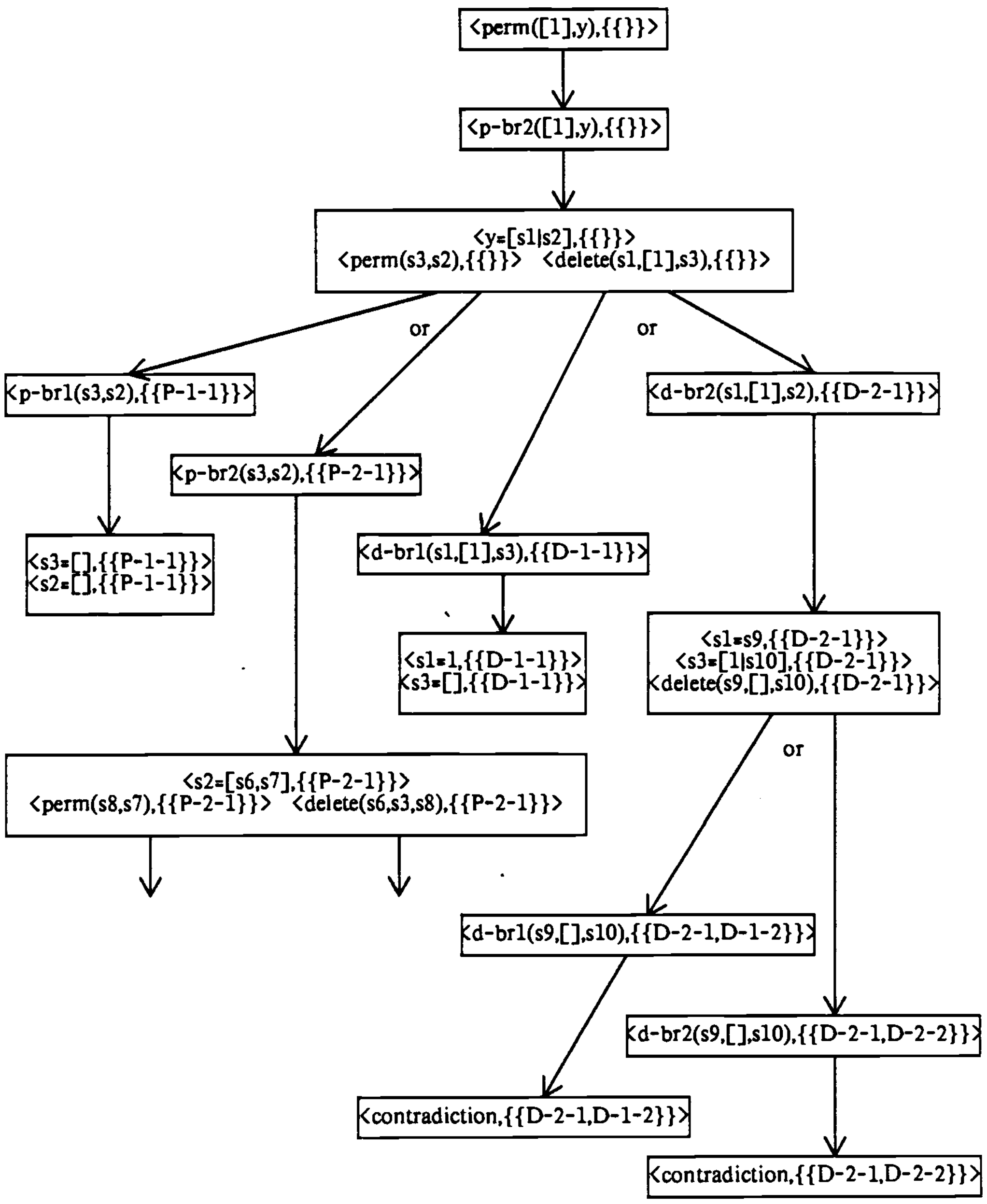

Figure 4.1: Permuting the list [1].

The initial assertion satisfies the antecedent of the implication

$$\forall\ u_1,u_2\ perm(u_1,u_2) \wedge \neg(u_1=[\,]) \supset p\text{-}br2(u_1,u_2),$$

since the inequality $\neg([1]=[\,])$ is true. FORLOG deduces that the second branch of the permute rule can be detached and asserts

$$\langle p\text{-}br_2([1],y),\{\{\}\}\rangle$$

which detaches the actual branch, adding the assertions

$\langle y=[s1|s2],\{\{\}\}\rangle$, $\langle perm(s3,s2),\{\{\}\}\rangle$, $\langle delete(s1,[1],s3),\{\{\}\}\rangle$.

Neither the *perm* nor the *delete* assertion causes a particular branch of its respective implications to detach. They both must enumerate. FORLOG does this by producing the branch assertions

$\langle p\text{-}br1(s3,s2),\{\{P\text{-}1\text{-}1\}\}\rangle$,	$\langle p\text{-}br2(s3,s2),\{\{P\text{-}2\text{-}1\}\}\rangle$,
$\langle d\text{-}br1(s1,[1],s3),\{\{D\text{-}1\text{-}1\}\}\rangle$,	$\langle d\text{-}br2(s1,[1],s3),\{\{D\text{-}2\text{-}1\}\}\rangle$,

each introducing its own assumption. This detaches the branches under those assumptions, asserting:

$\langle s3=[],\{\{P\text{-}1\text{-}1\}\}\rangle$	$\langle s3=[s4|s5],\{\{P\text{-}2\text{-}1\}\}\rangle$
$\langle s2=[],\{\{P\text{-}1\text{-}1\}\}\rangle$	$\langle s2=[s6|s7],\{\{P\text{-}2\text{-}1\}\}\rangle$
	$\langle perm(s8,s7),\{\{P\text{-}2\text{-}1\}\}\rangle$
	$\langle delete(s6,s3,s8),\{\{P\text{-}2\text{-}1\}\}\rangle$
$\langle s1=1,\{\{D\text{-}1\text{-}1\}\}\rangle$	$\langle s1=s9,\{\{D\text{-}2\text{-}1\}\}\rangle$
$\langle s3=[],\{\{D\text{-}1\text{-}1\}\}\rangle$	$\langle s3=[1|s10],\{\{D\text{-}2\text{-}1\}\}\rangle$,
	$\langle delete(s9,[],s10),\{\{D\text{-}2\text{-}1\}\}\rangle$.

FORLOG examines every equality assertion to enforce consistency. It discovers that not all of the assumptions introduced can hold simultaneously. That is to say, it derives some nogoods. For example, the equalities involving Skolem constant *s3* assert $s3=[]$ under assumption *P-1-1* and $s3=[s4|s5]$ (i.e. *s3* must be a list) under assumption *P-2-1*. These assumptions are inconsistent with each other. So the context $\{P\text{-}1\text{-}1,P\text{-}2\text{-}1\}$ is nogood. Through this type of reasoning, FORLOG discovers the following four nogoods:

$\{P\text{-}1\text{-}1,P\text{-}2\text{-}1\}$, $\{P\text{-}2\text{-}1,D\text{-}1\text{-}1\}$, $\{P\text{-}1\text{-}1,D\text{-}2\text{-}1\}$, $\{D\text{-}1\text{-}1,D\text{-}2\text{-}1\}$.

Inconsistencies, and thus nogoods, can be derived through other means as well. For example, pursuing the consequents of the assertion

$\langle delete(s9,[],s10),\{\{D\text{-}2\text{-}1\}\}\rangle$

derives a contradiction. With this assertion FORLOG cannot decide to pursue any particular delete branch. It enumerates by assuming the validity of both branches, producing the assertions

$\langle d\text{-}br1(s9,[],s10),\{\{D\text{-}2\text{-}1,D\text{-}1\text{-}2\}\}\rangle$,
$\langle d\text{-}br2(s9,[],s10),\{\{D\text{-}2\text{-}1,D\text{-}2\text{-}2\}\}\rangle$.

each introducing an additional assumption. Looking back at the implications for the branches, each consequent requires the second argument to be a list. In both branch assertions, the second argument is [] which is contradictory. Therefore,

{*D-2-1*,*D-1-2*}, {*D-2-1*,*D-2-2*}

are additional nogood contexts.

Notice further that an answer has been found for the initial Skolem constant *y*. To find out what it is, we can apply the relevant part of the equalities asserted. Right away *y* was equated to [*s1*|*s2*]. When FORLOG assumed *p-br1* under *P-1-1*, it equated *s2* to []. Further, when it assumed *d-br1* under *D-1-1* it equated *s1* to 1. Putting these together, *y* has the value [1] under context {*P-1-1*, *D-1-1*}, the permutation of the list [1] is the list [1].

As it stands, however, FORLOG would continue deriving consequences of the assertions believed under assuption *P-2-1*. In particular, the assertions

<*perm*(*s8*,*s7*),{{*P-2-1*}}>, <*delete*(*s6*,[*s4*|*s5*],*s8*),{{*P-2-1*}}>

are completely unconstrained. An attempt to pursue either would lead to infinite computation.

4.2 Using Resolution to Cut off Deductions

Recall that FORLOG will not use false assertions in any deductions. An assertion becomes false when all its supporting assumption sets become inconsistent (i.e. it has an empty label). An individual assumption set is inconsistent if it is a superset of a nogood. To enforce this, whenever FORLOG discovers a new nogood, the ATMS removes supersets of this nogood from every assertion's label that contains the nogood. With the example just presented, if the assumption set {*P-2-1*} were found to be nogood, the ATMS would remove it from the assertions

<*perm*(*s8*,*s7*),{{*P-2-1*}}>, <*delete*(*s6*,[*s4*|*s5*],*s8*),{{*P-2-1*}}>

among others, changing them to the following:

<*perm*(*s8*,*s7*),{}>, <*delete*(*s6*,[*s4*|*s5*],*s8*),{}>.

These assertions, which caused the infinite computation, would not be used for any further deductions since they are false.

The ATMS employs resolution to derive additional nogoods not discovered by FORLOG itself. Resolution works only with logical formulas in clause form (disjunctions of positive and negative literals). For example, with the nogoods discovered through equality reasoning, we have the following clauses:

$$(\neg P\text{-}1\text{-}1 \vee \neg P\text{-}2\text{-}1),\ (\neg P\text{-}2\text{-}1 \vee \neg D\text{-}1\text{-}1),\ (\neg P\text{-}1\text{-}1 \vee \neg D\text{-}2\text{-}1),$$
$$(\neg D\text{-}1\text{-}1 \vee \neg D\text{-}2\text{-}1),\ (\neg D\text{-}2\text{-}1 \vee \neg D\text{-}1\text{-}2),\ (\neg D\text{-}2\text{-}1 \vee \neg D\text{-}2\text{-}2).$$

Because FORLOG also asserts to the ATMS that assumptions introduced with enumerated branches are disjunctive, it has stored the following:

$$(P\text{-}1\text{-}1 \vee P\text{-}2\text{-}1),\ (D\text{-}1\text{-}1 \vee D\text{-}2\text{-}1),\ (D\text{-}1\text{-}2 \vee D\text{-}2\text{-}2).$$

These disjunctive assumptions tell the ATMS that one or more of the assumptions in the disjunction must be true.

Now we have a set of propositions in clause form. Resolving $(D\text{-}1\text{-}2 \vee D\text{-}2\text{-}2)$ with $(\neg D\text{-}2\text{-}1 \vee \neg D\text{-}1\text{-}2)$ we get the clause $(\neg D\text{-}2\text{-}1 \vee D\text{-}2\text{-}2)$. Use this result to resolve with $(\neg D\text{-}2\text{-}1 \vee \neg D\text{-}2\text{-}2)$ deriving $\neg D\text{-}2\text{-}1$. The clause $(D\text{-}1\text{-}1 \vee D\text{-}2\text{-}1)$ resolves with this new single nogood deriving $D\text{-}1\text{-}1$. This, then resolves with $(\neg P\text{-}2\text{-}1 \vee \neg D\text{-}1\text{-}1)$ to derive $\neg P\text{-}2\text{-}1$. Therefore, the assumption set $\{P\text{-}2\text{-}1\}$ is nogood, which is exactly what we wanted.

After applying resolution, the computation tree for *perm*([1],*y*) may be simplified by removing the false nodes. Only one answer remains (Figure 4.2).

4.2.1 Updating nogoods through resolution

As mentioned, the ATMS performs resolution at FORLOG's request. When FORLOG enumerates branches of a disjunction, it informs the ATMS of the newly introduced assumptions. The ATMS stores these disjunctive assumptions and also the discovered nogoods.

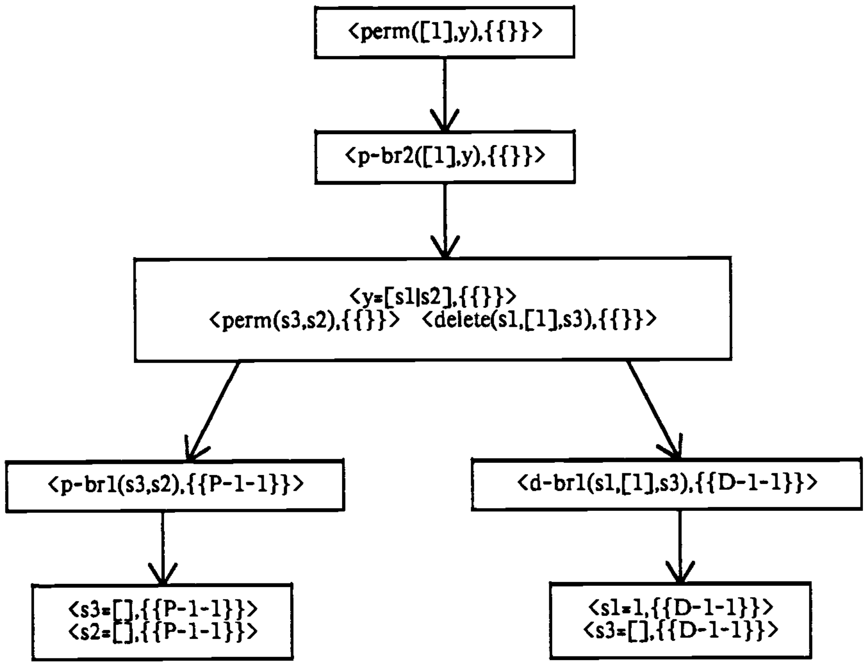

Figure 4.2: Trimmed version of permuting the list [1].

The actual resolution method employed is called negative hyperresolution. Given a single positive clause (a disjunctive set of assumptions) and a set of negative clauses (nogoods) each of which contains a single distinct atom of the positive clause, hyperresolution produces a single negative resolvent:

$$\begin{array}{l} A_1 \vee A_2 \vee \ldots \vee A_n \\ \neg A_1 \vee N_1 \\ \neg A_2 \vee N_2 \\ \vdots \\ \neg A_n \vee N_n \\ \hline N_1 \vee N_2 \ldots \vee N_n \end{array}$$

Each ($\neg A_i \vee N_i$) is a known nogood (N_i represents the remaining part of the nogood so it is a disjunction of negated assumptions). In addition, for any $j \neq i$, $A_j \notin N_i$. That is to say that to use a nogood in the hyperresolution it can contain only one negated assumption from the positive clause. The combined negative clauses $N_1 \vee N_2 \ldots \vee N_n$, constitute a new nogood.

For example, in the previous section we could have used hyperresolution:

$$\frac{\begin{array}{l} D\text{-}1\text{-}2 \lor D\text{-}2\text{-}2 \\ \neg D\text{-}2\text{-}1 \lor \neg D\text{-}1\text{-}2 \\ \neg D\text{-}2\text{-}1 \lor \neg D\text{-}2\text{-}2 \end{array}}{\neg D\text{-}2\text{-}1}$$

Single nogoods like the one derived above help to simplify other nogoods and disjunctions alike. In general, nogoods that are supersets of any other nogood can be forgotten. The ATMS keeps track of only the simplest nogoods. Notice that

$$\neg P \land (\neg P \lor \neg Q) \equiv \neg P$$

by the law of absorption. This means that when $\neg D\text{-}2\text{-}1$ was derived through hyperresolution, the ATMS can (and does) remove all nogoods that contain it.

Single nogoods also help to simplify cached disjunctions. The assumption that was discovered to be invalid can be removed from any disjunction in which it participates. The inverse is true for single disjuncts, such as the assumption *D-1-1* derived in the previous section. These can be viewed as assumptions that are true (not just assumed true), therefore any nogood that contains the single nogood can be simplified. The ATMS removes assumptions from any nogood that are simultaneously single disjunctions.

Whenever FORLOG deduces an additional nogood, that nogood, combined with existing nogoods, might hyperresolve with an existing disjunction. Likewise, whenever FORLOG asserts a new disjunction of assumptions, it may hyperresolve with existing nogoods. This means that for completeness, this resolution strategy must be attempted whenever FORLOG discovers new nogoods or new disjunctions.

4.2.2 Updating labels through resolution

Not only does the ATMS discover additional nogoods through resolution, but it updates labels (on assertions) as well. Recall that assumption sets act as implications. For example, the assertion

$\langle father(Doug,Dan),\ \{\{A,C\},\{B,C\}\}\rangle$,

represents the following:

$$A \land C \supset father(Doug,Dan)$$
$$B \land C \supset father(Doug,Dan).$$

These implications can be rewritten as clauses (formulas containing only disjunctions), such as:

$$\neg A \lor \neg C \lor father(Doug,Dan)$$
$$\neg B \lor \neg C \lor father(Doug,Dan).$$

Suppose we had the disjunction $A \lor B$. We can use resolution to derive the clause

$$\neg C \lor father(Doug,Dan),$$

which is a simplified version of the original assertion. Therefore, that assertion's label could be simplified to obtain

$\langle father(Doug,Dan),\ \{\{C\}\}\rangle$.

As in the previous section, hyperresolution provides the specific method. To resolve a particular assertion's label, each context (set of assumptions) in the label can be treated as a negative clause. A disjunction that includes an assumption in the assertion's label provides the positive clause. In this case, however, the resulting negative clause is not a new nogood, it is a new context supporting the original assertion. Often this new context is a subset of one of the original contexts in the label, which acts to simplify the label as in the example above.

We want labels to be as simple as possible. Conceptually, to maintain these simple labels, resolution needs to be performed whenever a label is changed or a new disjunction or new nogood discovered. Because of its expense, in actual practice we almost never take advantage of this feature.

Chapter 5
Conclusions

This paper has described a logic programming language, FORLOG, and illustrated a few key points in its design. This chapter summarizes these points, discusses further work that needs to be done, and points out some of the contributions of this thesis.

5.1 Summary

FORLOG, since it uses a forward-chaining computation model, represents an alternative paradigm for logic programming. In fact, the traditional method of backward chaining with Horn clauses can be implemented with a computation model for logic programs that is a subset of FORLOG. These programs are derived in a straight forward manner from the original Horn clause program.

Disjunctions appear in the derivation of FORLOG programs from Horn clause programs. In general, disjunctions represent nondeterministic choices. The valid branches of the disjunction should be asserted and the invalid ones should be discovered as soon as possible. Through careful compilation of the formula containing a disjunction, the valid branches can be determined efficiently at run-time. However, compiling introduces negative literals.

FORLOG handles negative literals in a "sound" manner depending upon their types. These types include, test, functional, functional dependency, and simple. Test literals call attached boolean Lisp functions when their arguments are bound that determine their truth value. Both functional and functional dependency literals require FORLOG to employ functional dependency theory to decide truth value. Functional literals differ from functional dependency literals in that they have attached Lisp functions defining the functional dependencies among their parameters. Any other literal is categorized as a simple literal. At present, FORLOG has no means to evaluate the validity of negated simple literals.

One common method for dealing with negations of simple literals is to use negation-as-failure. This method closes down the database of facts. Any fact not in the database is false. FORLOG could achieve this behavior but not easily.

Unfortunately, cases arise where compilation does not remove the nondeterministic choice. When FORLOG cannot decide which branches are valid, it falls into generate-and-test behavior by assuming that all of them are valid. FORLOG takes advantage of deKleer's ATMS to keep track of all assumptions made.

5.2 Future Research

Logic programming has thus far benefitted greatly from efficient implementations of the backward chaining methodology. It is not clear that such efficiency gains are possible with FORLOG. For this system to be acceptable for anything other than research, progress must be made in this area. Parallelism might be the key here.

Another area of interest is the issue of control. Open questions remain concerning the order in which branches of a disjunction should be enumerated. Often enumerating the "right" branch will remove the need to enumerate some or all of the other branches. The options include some kind of run-time control versus compile-time control.

On the practical side, FORLOG needs further testing to see how it performs with larger, more realistic problems. One research group is currently exploring applications in machine learning and mechanical design (Dietterich & Ullman, 1986).

5.3 Contributions of the Thesis

This work on disjunctions in forward-chaining logic programming contributes to the logic programming field by presenting a method for compiling away nondeterminism. It presents logically sound methods for handling negative literals. In addition, it illustrates how the ATMS can be integrated into a logic programming system.

FORLOG certainly will not replace systems like Prolog, but it does represent another step toward having specifications as programs.

Chapter 6

Bibliography

Clark, K.L. 1978. Negation as Failure. In *Logic and Databases*, H. Gallaire and J. Minker Eds. Plenum Press, New York, pp. 293-322.

Clark, K.L., and Gregory, S., 1986. PARLOG: Parallel Programming in Logic, *ACM Transactions on Programming Languages and Systems*, 8 (1) 1-49.

Clocksin, W.F., and Mellish, C.S., 1984. *Programming in Prolog*. Springer-Verlag, Berlin.

Cohen, P., and Feigenbaum E., 1982. *The Handbook of Artificial Intelligence*, Volume 3. Kaufman, Los Altos, Calif., pp. 120-123.

deKleer, J. 1986a. An Assumption-based TMS. *Artificial Intelligence*, 28 (2) 127-162.

deKleer, J. 1986b. Extending the ATMS. *Artificial Intelligence*, 28 (2) 163-196.

deKleer, J. 1986c. Problem Solving with the ATMS. *Artificial Intelligence*, 28 (2) 197-224.

Dietterich, T.G., and Ullman, D.G., 1986. FORLOG: A Logic-based Architecture for Design. Technical Report 86-30-8, Oregon State University.

Kowalski, R. 1979. *Logic for Problem Solving*. American Elsevier, New York.

Lloyd, J.W. 1984. *Foundations of Logic Programming*. Springer-Verlag, Berlin.

Naish, L. 1985a. Negation and Control in Prolog, Technical Report 85/12, University of Melbourne, Australia.

Naish, L. 1985b. Prolog Control Rules. *Proceedings of the Ninth IJCAI*, 720-722.

Reiter, R. 1980. A Logic for Default Reasoning. *Artificial Intelligence*, 13 81-132.

Robinson, J.A. 1965. A Machine-Oriented Logic Based on the Resolution Principle, *J. ACM*, 12 (1) 25-48.

Shapiro, E. 1983. A Subset of Concurrent Prolog and its Interpreter, Technical Report TR-003, ICOT-Institute for New Generation Computer Technology, Tokyo, Japan.

Sterling, L., and Shapiro, E., 1986. *The Art of Prolog*. MIT Press, Cambridge, Mass.

Ueda, K. 1985. Guarded Horn Clauses, ICOT Technical Report 103, ICOT, Tokyo Japan.

Warren, D.H.D. 1977. Implementing Prolog - Compiling Logic Programs 1 and 2, DAI Research Reports 39 and 40, University of Edinburgh.