# CACHING AND NON-HORN INFERENCE IN MODEL ELIMINATION THEOREM PROVERS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Donald F. Geddis
June 1995

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Michael R. Genesereth
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Nils J. Nilsson

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Narinder P. Singh

Approved for the University Committee on Graduate Studies:

# Abstract

Caching in an inference procedure holds the promise of replacing exponential search with constant-time lookup, at a cost of slightly-increased overhead for each node expansion. Caching will be useful if subgoals are repeated often enough during proofs.

In experiments on solving queries using a backward chainer on Horn theories, caching appears to be very helpful on average. When trying to extend this success to first-order theories, however, intuition suggests that subgoal caches are no longer useful. The cause is that complete first-order backward chaining requires goal-goal resolutions in addition to resolutions with the database, and this introduces a context-sensitivity into the proofs for a subgoal. A cache is only feasible if the solutions are independent of context, so that they may be copied from one part of the space to another.

It is shown here that a full exploration of a subgoal in one context actually provides complete information about the solutions to the same subgoal in all other contexts of the proof. In a straightforward way, individual solutions from one context may be copied over directly. More importantly, non-Horn failure caching is also feasible, so no additional solutions in the new context (that might affect the query) are possible and therefore there is no need to re-explore the space in the new context. Thus most Horn clause caching schemes may be used with minimal changes in a non-Horn setting.

In addition, a new Horn clause caching scheme is proposed: postponement caching. This new scheme involves exploring the inference space as a graph instead of as a tree, so that a given literal will only occur once in the proof space. Despite the previous extension of failure caching to non-Horn theories, postponement caching is incomplete in the non-Horn case. A counterexample is presented, and possible enhancements to reclaim completeness are investigated.

# Acknowledgements

My research experience as a graduate student at Stanford has been shaped by my advisors over the years:

- Nils Nilsson sparked my interest in artificial intelligence as an undergraduate at Stanford. He taught two AI classes I took then and was a very accessible professor, even inviting me to observe his newly formed research group. That cemented my interest in pursuing graduate study. Nils was my primary advisor for the first few years.

- Mike Genesereth was my final graduate advisor, and assisted both financially and intellectually. He was especially helpful in the final stages, providing detailed feedback and suggestions to help me pass beyond that dreaded "all but thesis" stage.

In addition, I would like to thank my orals committee, who besides my reading committee of Mike Genesereth, Nils Nilsson, and Narinder Singh, included Jeff Ullman and John McCarthy, as well as Paul Teicholz as the chair.

Substantial contributions to the content of this thesis were made by

- Narinder Singh, who worked extensively with me to cooperatively prove the results in section 4.4.2. In one memorable weekend, Narinder and I independently proved Lemmas 15 and 16 and Theorem 18.

- Scott Roy, who took pity on the confusion I exhibited trying to understand the behavior of postponement caching on the pigeonhole problem. He finally couldn't stand my complaining any more and with a few days of hard thought managed to produce the counterexample in table 6.1. That insight lifted the intellectual fog I was in at the time.

- David Sturgill, who with his advisor Alberto Segre, corresponded with me via email for some time and explored some of the conjectures I proposed. In particular, David came up with the example in table 4.6.

Mark Stickel gave me references to much of the related work on non-Horn caching, and gave comments on a draft of this thesis. He also kindly provided me with a pre-publication copy of a paper of his own [Sti94].

Jeff Ullman provided references and much discussion about the technique of magic sets, which led to the comparison of this work with the bottom-up approaches in sections 5.4.3 and 6.3.

Much valuable feedback came over the years from the members of the various research groups I have been a member of: Nils Nilsson's Principia group, Matt Ginsberg's Principia group, and

Mike Genesereth's MUGS group. Many thanks especially to the members of the MUGS group, who had to endure what must have appeared to be my constant talks on the same subject of caching in theorem proving. Particular thanks are due to Narinder Singh and Scott Roy, who were always willing to listen to my latest problems and potential solutions, and constantly offered valuable advice and feedback.

Pat Witting provided constructive comments on various drafts of this thesis. In fact Pat sparked the actual writing down of words by means of a challenge bet he proposed between the two of us. I recall winning that bet, but somehow I never seemed to be able to collect...

Finally, I would like to thank the people who kept me sane during my years as a graduate student: my family, the jujitsu crowd, the volleyball folks, and of course Laura.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

A cache is a device for re-using previous work. In the course of solving some problem, subproblems are solved and the solutions stored in the cache. If those subproblems later recur, the solution can be retrieved from the cache rather than solved again.

Caches are well known in computer science. In computer hardware architecture, memory caches store the contents of some general memory address in a small but very high speed piece of RAM. In computer systems, virtual memory modules in operating systems store a subset of the pages of memory that are on hard disk. In software systems, world wide web caches store the HTML pages referred to by particular URLs.

This thesis is about caching in first-order inference. An inference engine derives facts which are logically implied by a knowledge base. In the process of checking whether some particular query follows from a given database, many intermediate facts are considered. A cache allows the results of the inference effort on such intermediate facts to be remembered and then re-used when the same facts occur elsewhere during the inference effort.

## 1.1 Subgoal Caching

Caching attempts to replace further search by simple lookup. This has the potential benefit of occasionally reducing the exponential cost of search with the constant (or sub-linear) cost of lookup, in those cases when portions of the search are repeated.

This benefit is not without some cost. The need to examine the cache at each node expansion in the inference space adds some (usually small) overhead at every step. Also, the benefits are only realized if the problems being solved have numerous repeated subproblems. Nonetheless it is often the case that adding a cache to an inference procedure can dramatically improve its performance on problems of interest.

Repeated subproblems can occur for many reasons. One of the common causes is the use of database rules that have recursive definitions. Recursion does not guarantee that subgoals will be repeated, but it makes repetition quite likely. For example, Fibonacci numbers are a sequence of integers, where each subsequent number is the sum of the previous two:[1]

[1]Capitalized words and letters are constant terms, and lowercase words and letters are variables.

$$Fib(0) = 1$$
$$Fib(1) = 1$$
$$Fib(n) = Fib(n-1) + Fib(n-2)$$

The search space for solving such a problem grows exponentially with the Fibonacci number being computed. A backward chaining proof that the fifth Fibonacci number equals 8 is shown in figure 1.1. Note the subgoal repetition: besides all the instances of subgoals which can be proved in one step, there are three instances of the subgoal $Fib(2) = 2$, and two instances of the subgoal $Fib(3) = 3$. A caching proof of the same query is shown in figure 1.2. In this case, caching simulates the well-known method of dynamic programming [AHU87], which can turn an exponential space into a linear one. This complexity advantage is occurring without the need for bottom-up inference; all of these searches remain top-down, focused on the goal.

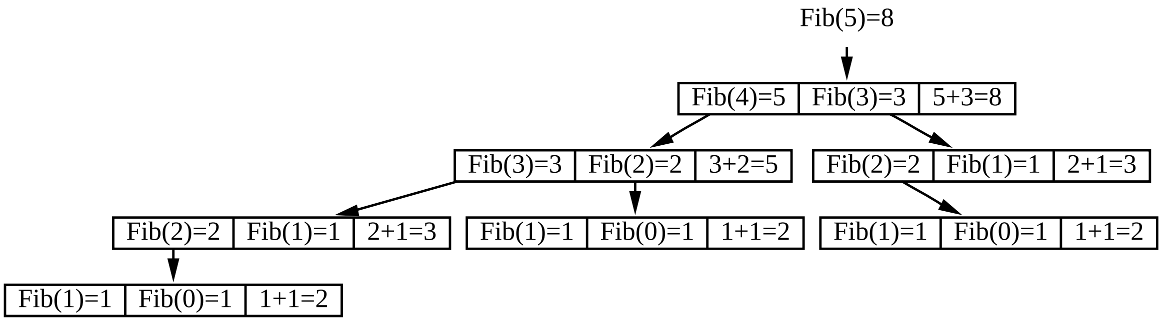


Figure 1.1: Proof showing $Fib(5) = 8$, with repetition

Even if the recursive subgoal does not repeat exactly, the context in which it appears often forces repetition. Consider the factorial function,[2] which defines a sequence of integers such that the $n^{\text{th}}$ number is $n$ times the previous one:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

A backward chaining proof that $4! = 24$ is shown in figure 1.3. Proofs like this can be arbitrarily deep, simply by requesting the factorial of ever larger values.

The direct proof of 4! does not exhibit any subgoal repetition. Imagine, though, that the context was a conjunctive subgoal of the form

Find an integer $n$ such that $n! = 6$

This proof (shown in figure 1.4) proceeds by selecting an integer and then checking whether $n! = 6$. Each subsequent effort to solve the second conjunct results in a repetition of a previous factorial subgoal.

If instead solved subgoals are cached, then a repeated subgoal need not be solved again. The proof space for the same problem with caching enabled is shown in figure 1.5. Aside from the first computation of 0! which is looked up from the database, all the other instances of the $n!$ subgoal are solved in one step by looking up the $(n-1)!$ solution in the success cache.

[2]The $n^{\text{th}}$ factorial number is typically written as $n!$.

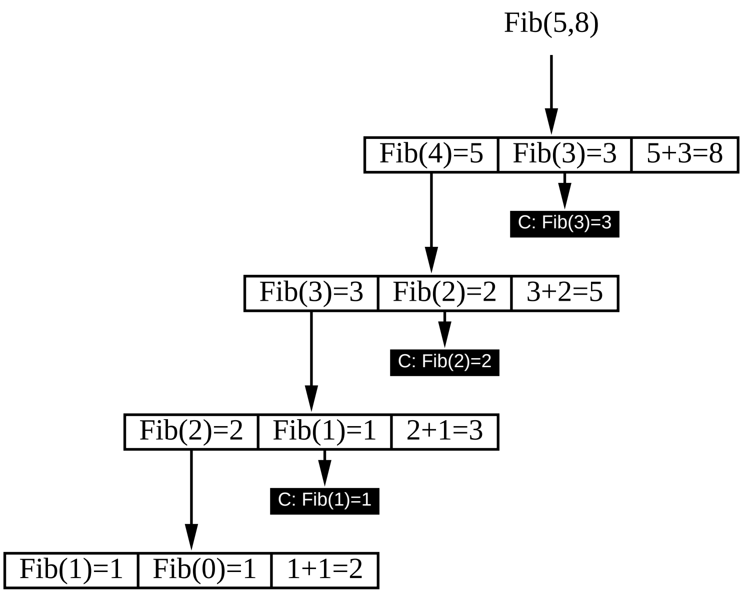


Figure 1.2: Proof showing $Fib(5) = 8$, with caching

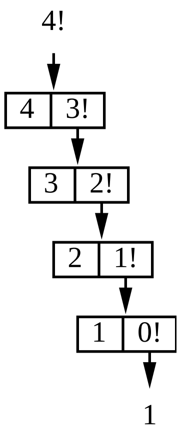


Figure 1.3: Proof of $4! = 24$

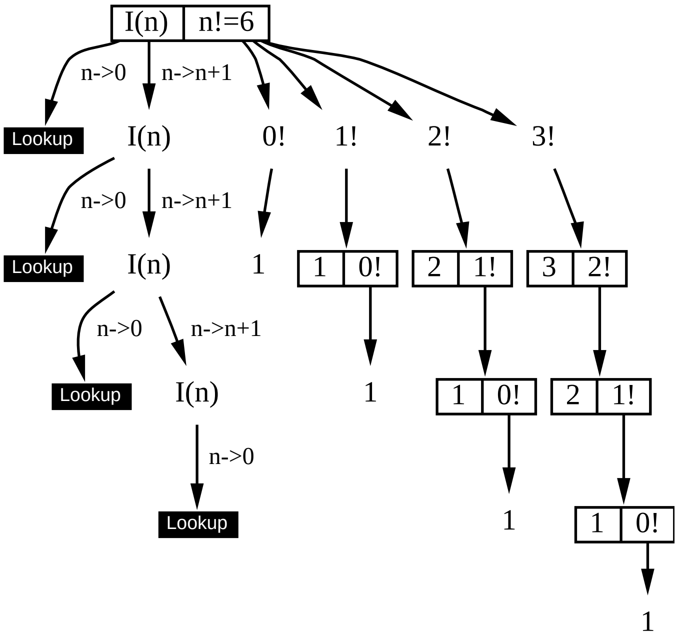


Figure 1.4: Proof of $n! = 6$ with repetition

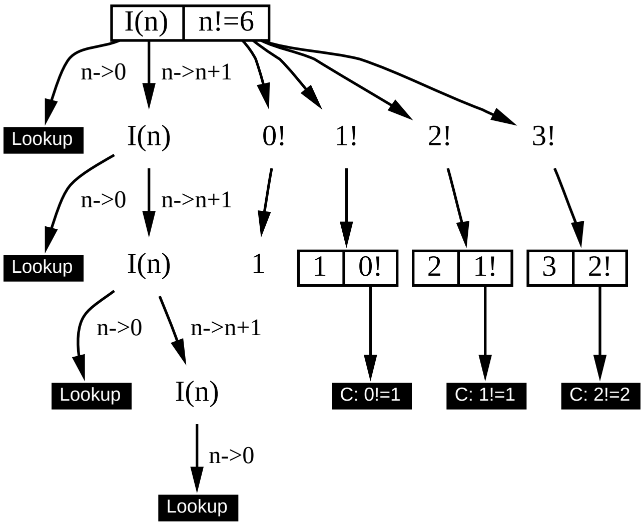


Figure 1.5: Proof of $n! = 6$ using caching

In addition to the structure of the problem, the search strategy used in inference also can cause repetition. For example, iterative deepening [Kor85] is a commonly used search strategy. Iterative schemes guarantee that subgoals will be repeated during a proof effort, as a subgoal that is solved during one level of the iteration will need be solved again when the next iteration occurs. (The addition of depth cutoffs makes caching more complicated, but repetition is still quite common in these cases.)

Not only can successful proofs be cached, but the fact that a given subgoal failed to be proved can also be cached. Such failures are generally even more expensive than successful subgoals, and thus the potential savings is greater. This is because any single proof is sufficient to establish a true subgoal, but in order to know that a subgoal is false (or, rather, that it doesn't follow from the theory), it is necessary to completely explore the space below it. Repeating such failures for the same subgoal during the course of a proof is often a source of much inefficiency.

The notion of failure caching has one last impact, which is on more generalized subgoal caches. Assume that, during the course of some proof, we had come across some subgoal `P(x)`. Rather than finding a full proof of the subgoal (that `P(x)` is true for all objects `x`), the more usual case is that the subgoal is proved for some instances, say for `x`$\rightarrow$`1` and `x`$\rightarrow$`2`. If this same subgoal is encountered later in the proof, we would like to use the results of the first effort in lieu of rederiving those solutions. The ability to cache failures allows us to conclude that not only are the instances `1` and `2` valid in this new location, but also that there are no other possible solutions. There is thus no need to explore the space below the second `P(x)`, as all information about what might be found there is already known by the subgoal cache.

## 1.2 Thesis Contributions

In this thesis, we explore the topic of caching in non-Horn theorem proving (in particular, by augmenting the model elimination inference algorithm [Lov78, section 3.6]). The previous two examples (of Fibonacci numbers and the factorial function) can both be defined using only PROLOG-like Horn sentences. In such a sentence, the database rule is composed of a head which is a single positive literal, and a body which is a conjunctive set of positive literals. Full first-order logic, however, is more expressive than this. Non-Horn theories allow one to write disjunctions (*e.g.* that either `A` or `B` is true, without saying exactly which one is true) and negations (*e.g.* that `C` is false if some conditions are true), neither of which is expressible in the more limited Horn rules.

The extensions of the basic inference algorithms, from algorithms that are complete on Horn subsets of logic to those which work on full first-order logic, are well known and have been extensively studied. Extending the caching schemes to the non-Horn case, however, has appeared to be infeasible. It seems unfortunate to lose the benefits of caching shown in the previous examples, just because a more expressive language is required for some application.

What prevents the standard Horn clause caching schemes from functioning in first-order inference? A caching scheme is built on the notion that solutions to a subgoal in one part of the inference space can be used when that subgoal appears in another part of the space. Non-Horn inference, however, requires the addition of a reduction operation, which means that a subgoal can succeed if it unifies with (a negated version of) an ancestor goal. This introduces a context-sensitivity into the solutions for a subgoal. A given subgoal might have a proof in one location because of a fortuitous set of ancestor goals; when the subgoal appears again in some different part of the space, most likely the set of ancestor goals will be different, and thus the previously-discovered proof will no longer apply.

The primary contribution of this thesis is the surprising discovery that, despite the apparent context-sensitivity of non-Horn inference, the standard Horn clause caching schemes can be added to a non-Horn inference engine basically unchanged, and the resulting algorithm will still be complete. More specifically, we present a proof that, if a subgoal has a solution in any location (which can affect the query), then it will have some solution in every location in which it occurs.

This is surprising because it would seem that a subgoal proof in one location might depend on some particular ancestors of that subgoal. How does a different instance of the subgoal, which occurs with a different set of ancestors, also have a solution? The unexpected proof in the new location is essentially a copy of the original proof, except that each reduction in the original proof (which used one of the missing ancestors) is replaced with a new subproof. By using contrapositives of database rules, new subproofs are constructed which essentially duplicate the reasoning from the goal to the original ancestor, but in reverse. This establishes a subproof of the child subgoal which was originally solved by reduction; since the child must be a negation of the ancestor, this reverse reasoning is sufficient to prove the child in the new location.

Thus we know that if a subgoal can ever be used to prove the query, it will have a solution everywhere in the inference space. This means that if we encounter a subgoal and it fails to have a solution where we first encounter it, then no occurrence of that subgoal can ever help prove the query, and hence we can prune all future such occurrences. But this is just the standard Horn clause failure caching scheme!

Failure caching is the key; it essentially establishes that exploring the space for a subgoal in one location reveals all the information derivable from exploring all the occurrences of the same subgoal everywhere in the inference space. If there would have been a particular solution at one of subgoals, then there would also be some related proof at the first subgoal encountered. This justifies all the typical caching schemes: if a subgoal is completely explored at some location, it need never be solved again.

A secondary contribution is the development of a new caching scheme, postponement caching. This scheme basically searches the inference space as a graph rather than a tree. A given subgoal only occurs once in the space; any further instances are slaved to the original occurrence, so that the effort of proving a subgoal need only happen once. In contrast to the typical caching schemes, postponement caching allows the re-use of partial proving efforts. The cached information for a subgoal includes the solutions discovered so far, as well as a continuation in the event that more solutions are needed from any one of the slaved instances of the subgoal.

In the case of Horn inference, postponement caching is an algorithm that echos previous developments: specifically, magic templates in deductive databases, and memoing in logic programming. Non-Horn postponement caching, unfortunately, is not complete, but the cause of incompleteness is explored and suggestions for possibly repairing the algorithm are considered.

An outline of the thesis is as follows: Chapter 2 gives a brief overview of formal logic and the model elimination algorithm. This differs from the standard presentation only in the formatting of chains for model elimination, as described in section 2.4. Chapter 3 describes typical subgoal caching schemes that have been used in Horn clause theorem provers. Chapter 4 describes the apparent difficulty of failure caching in the non-Horn case, and presents the solution that non-Horn failure caching is still complete. Chapter 5 presents the postponement caching algorithm; finally, chapter 6 explains how postponement caching fails in the non-Horn case, and suggests future work to improve it in that situation.

This thesis is also available online as a technical report [Ged95].

# Chapter 2

# Inference

## 2.1 Introduction

Automated reasoning for first-order logic is somewhat of an odd field. Because of the well-known semi-decidable nature of logical inference, any system which attempts to determine whether a query follows from a set of sentences is in trouble from the outset. Such a system may be sound, in that any answer it gives (either that the query is implied by the database, or that it is not) is correct. The system may be complete, in that if either the query or its negation follows from the database, then the system can eventually determine that. But for any sound and complete inference system, there will always exist some query and some database for which the system never returns an answer at all.

Given this fundamental limitation, research in automated reasoning has progressed by expanding the number of queries and databases for which correct answers may be returned. The first systems proceeded by model checking: for every object mentioned by the theory, they would try to plug all combinations of such objects into every database sentence, using a propositional satisfiability engine to determine if the system had found a counterexample to the theory. In many problems of interest, however, the number of entities in the universe of discourse grows so quickly that a generate-and-test checker of this kind is quickly overwhelmed.

A huge leap was made by Robinson [Rob65], who realized that by propagating constraints in the form of variable binding lists, infinite sets of domain objects could be checked in a single step. His unification concept, the key to this summarizing computation, has formed the basis for further automated provers.

Robinson's resolution, while astonishingly more efficient than previous techniques, still had poor complexity for some small interesting problems. The number of resolvants grew very quickly, and so work proceeded on restrictions of resolution (*i.e.* removing certain resolvants which were possible in the original formulation) which were still complete. One of these was refined by Loveland [Lov78] into a very streamlined backward-chaining algorithm called model elimination. The model elimination procedure focussed on individual literals rather than the clauses of resolution. A very efficient PROLOG-like implementation by Stickel [Sti89] led to the current enormous popularity of this form of automated inference.

All of these formulations are still bound by the theoretical restriction: there are some queries for which they will run forever. Nonetheless, the class of questions that are answerable has been growing steadily.

## 2.2 First Order Predicate Calculus

This section contains a brief background of mathematical logic. For more details see any introductory text on logic or artificial intelligence (for example, Nilsson's [Nil80, chapter 4]).

### 2.2.1 Syntax

A *symbol* is a sequence of alphabetic characters. Each symbol is assigned to exactly one of the following classes: *predicate*, *variable*, *function*, *object*. Lowercase symbols are used for variables, and capitalized symbols for predicate, function, and constant symbols.[1]

A *simple term* is either a constant symbol or a variable symbol. A *term* is either a simple term, or else a constant function of some arity applied to the appropriate number of terms.

An *atomic formula* is a predicate of some arity applied to the appropriate number of terms. A *literal* is either an atomic formula, or the negation of an atomic formula (indicated by a prefixed "`Not`").

A *formula* is either a literal, or a negated formula, or else two formulas joined by one of the connectives `And`, `Or`, or `Is Implied By`. (Occasionally `Implies` is used as the dual of `Is Implied By`.) Sometimes the symbols in table 2.1 are used instead of the English variants.

| | |
|---|---|
| `Not` | $\neg$ |
| `And` | $\wedge$ |
| `Or` | $\vee$ |
| `Is Implied By` | $\Leftarrow$ |
| `Implies` | $\Rightarrow$ |
| `For All` | $\forall$ |
| `There Exists` | $\exists$ |

Table 2.1: Logical symbols

*Propositional calculus* is a subset of predicate calculus with no variables, functions, or objects, and all relations have an arity of exactly zero.

A *sentence* is a formula where every variable is enclosed in the scope of either a universal quantifier ("`For All`") or an existential quantifier ("`There Exists`"). These also have symbolic variants, as shown in table 2.1.

(This predicate calculus is called *first order* because variables are quantified only over terms, not over predicate or function symbols. More general quantification leads to higher order logics. We will be concerned only with first-order logic in this thesis.)

In this thesis, logical expressions are written in `typewriter` font. Terms, atomic formulas, and quantifiers are written in prefix form with optional parentheses indicating the scope of application for ambiguous constructs.

[1] This is the opposite of the standard PROLOG notation, which uses uppercase for variables and lowercase for the others.

### 2.2.2 Semantics

An *interpretation* assigns a correspondence between the elements of the language and the relations, entities, and functions in the domain of discourse. Each constant symbol refers to an object in the domain. Each function symbol refers to a many-to-one mapping from objects in the domain to other objects in the domain. Each predicate symbol refers to a set of tuples of objects from the domain.

Once an interpretation has been defined, we can say that an atomic formula is true if the interpretation of the terms is a tuple in the interpretation of the predicate. A formula connected by `And` is a *conjunction*, and is true if both subformulas are true, otherwise it is false. A formula connected by `Or` is a *disjunction*, and is true if either (or both) subformulas are true, and false otherwise. A formula prefixed by `Not` is true if and only if the subformula is false. A formula connected by `Is Implied By` is true if the first subformula is true, or if the second subformula is false, and is false otherwise.

A formula under the scope of a universal quantifier is true if the subformula is true for *all* assignments of the variable to entities in the domain. A formula under the scope of an existential quantifier is true if the subformula is true for *at least one* assignment of the variable to an entity in the domain.

A set of premise sentences *logically implies* a goal sentence if, in every interpretation which assigns each of the premise sentences true, then the goal sentence is also assigned true. Logical implication is *semidecidable*. This means that, if a goal sentence follows from a set of premise sentences, there is an automated procedure which can determine that fact. In addition, any such procedure will not terminate for some combination of goal and premise sentences.

A given first order proof procedure is *complete* if, when given a set of premise axioms and a goal sentence, will eventually return `true` whenever the axioms logically imply the goal. A proof procedure is *sound* if it never returns `true` for goals which are not logical implications of the premise axioms.

## 2.3 Clausal Form

A *clause* is a disjunction of literals, where all variables are implicitly universally quantified. Any sentence in first order predicate calculus can be converted to a logically equivalent sentence in clausal form[2]. The conversion procedure is described in most automated reasoning texts; see, for example, the descriptions by Loveland [Lov78, section 1.5], Nilsson [Nil80, section 4.2.1], or Genesereth and Nilsson [GN87, section 4.1]. The steps involved are:

1. Eliminate implication symbols
2. Reduce the scope of negation symbols
3. Standardize variables apart
4. Eliminate existential quantifiers (by Skolemization)
5. Convert to prenex form

[2]This is sometimes referred to as Skolem conjunctive form.

6. Put in conjunctive normal form
7. Eliminate universal quantifiers
8. Eliminate `And`
9. Rename variables (standardize apart again)

An example from Nilsson [Nil80] is

$$\forall x\ (\ P(x) \Rightarrow (\ \forall y\ (\ P(y) \Rightarrow P(F(x,y))\ ) \wedge \neg\forall y\ (\ Q(x,y) \Rightarrow P(y)\ )))$$

After each of the procedure's steps, the corresponding result is

1. $\forall x\ (\ \neg P(x) \vee (\ \forall y\ (\ \neg P(y) \vee P(F(x,y))\ ) \wedge \neg\forall y\ (\ \neg Q(x,y) \vee P(y)\ )))$

2. $\forall x\ (\ \neg P(x) \vee (\ \forall y\ (\ \neg P(y) \vee P(F(x,y))\ ) \wedge \exists y\ (\ Q(x,y) \wedge \neg P(y)\ )))$

3. $\forall x\ (\ \neg P(x) \vee (\ \forall y\ (\ \neg P(y) \vee P(F(x,y))\ ) \wedge \exists w\ (\ Q(x,w) \wedge \neg P(w)\ )))$

4. $\forall x\ (\ \neg P(x) \vee (\ \forall y\ (\ \neg P(y) \vee P(F(x,y))\ ) \wedge (\ Q(x,G(x)) \wedge \neg P(G(x))\ )))$

5. $\forall x\ \forall y\ (\ \neg P(x) \vee ((\ \neg P(y) \vee P(F(x,y))\ ) \wedge (\ Q(x,G(x)) \wedge \neg P(G(x))\ )))$

6. $\forall x\ \forall y\ ((\ \neg P(x) \vee \neg P(y) \vee P(F(x,y))\ ) \wedge (\ \neg P(x) \vee Q(x,G(x))\ ) \wedge (\ \neg P(x) \vee \neg P(G(x))\ ))$

7. $(\ \neg P(x) \vee \neg P(y) \vee P(F(x,y))\ ) \wedge (\ \neg P(x) \vee Q(x,G(x))\ ) \wedge (\ \neg P(x) \vee \neg P(G(x))\ )$

8. $\neg P(x) \vee \neg P(y) \vee P(F(x,y))$
   $\neg P(x) \vee Q(x,G(x))$
   $\neg P(x) \vee \neg P(G(x))$

9. $\neg P(x_1) \vee \neg P(y) \vee P(F(x_1,y))$
   $\neg P(x_2) \vee Q(x_2,G(x_2))$
   $\neg P(x_3) \vee \neg P(G(x_3))$

## 2.4 Model Elimination

The ideas and examples in this thesis are presented as augmentations to a complete form of the problem reduction framework [Lov78, chapter 6] that Loveland calls the MESON procedure. A theorem prover using the framework searches an and/or tree of literals, where each subsequent level is created by backward chaining on some parent literal, or else by a resolution between a literal and one of its ancestor literals.

Loveland has shown [Lov78, section 6.2] that such a framework is equivalent to weak model elimination [Lov78, section 3.6]. As weak ME is often more convenient to prove properties about, some of the formal results in this thesis will be about augmentations of the weak ME procedure.

Weak model elimination behaves very much like a version of PROLOG with a few modifications. PTTP [Sti89], is an implementation of this complete version of PROLOG where

- The *occurs check* is added to the unification routine, so that the unifications are sound.
- Successfully resolved ancestors remain on the chain (albeit specially marked), so that they may participate in future reduction operations.

A complete description of model elimination, proofs of completeness, and several variants are given by Loveland [Lov78]. The presentation here differs from the standard one in two ways:

1. Chains are written with the most recent literals on the left and the oldest to the right, and right justified in the text. This reinforces the correspondence with the stack data structure that they represent.
2. Literals in the chains are written as the complement of the usual notation. This allows an easier mapping between the chains and the MESON-style proof graphs.

(The formal description below is adapted from Astrachan's description [Ast92].)

### 2.4.1 Inference Rules

The ME procedure is a set of three inference rules that operate on a structure called a *chain*. A chain is an ordered list of literals, representing a set of goals to be proved. Each literal may be positive or negative, and each may be either an *A-literal* or a *B-literal*. A-literals represent ancestor goals of all the literals that are more recent. They are shown as bracketed when listing a particular chain. B-literals are subgoals that have yet to be proven.

Chains are stack-like structures, and thus are manipulated by transforming the literal at the top of the stack. In this document will shall write a chain as

`R(A), Q(y), P(x)`

where `R(A)` is the most recent literal, and `P(x)` is the oldest. Thus `R(A)` will be the literal most affected by any transformation of the chain, and the literal in this position will be referred to as the *top* literal in the chain.

A chain may be transformed by one the following three inference rules:

1. *Extension.* A backward-chaining step using the top literal and some clause from the database.

2. *Reduction.* A *reductio ad absurdum* conclusion, found by resolving the top literal with some earlier A-literal.

3. *Contraction.* The removal of top A-literals.

The *extension* operation is the same as the Prolog inference operation (using sound unification), except that it retains the unified literal as an A-literal. (As is the usual case, we assume that all chains and clauses are renamed apart so that they are variable disjoint as necessary.)

As an example, given the chain

```
Q(F(x),y), [R(y,z)], P(x,z)
```

and the database clause

```
Q(F(A),C) ∨ R(C,B)
```

which is equivalent to the backward chaining rule

```
Q(F(A),C) ⇐ ¬R(C,B)
```

the extension of the chain using the clause (which involves unifying the first literal in the clause with the top literal in the chain) is the chain

```
¬R(C,B), [Q(F(A),C)], [R(C,z)], P(A,z)
```

The unifying binding list of the two literals is { `x→A, y→C` }, and this binding is applied to the entire chain.

In general we have

**Definition 1 (Extension)** *Given chain $C_1$ of the form $l_1C_0$ with top B-literal $l_1$, and input clause $C_2$ with literal $l_2$ such that $l_1$ and $l_2$ are unifiable with most general unifier (mgu) $\theta$, the ME <u>extension</u> operation of $C_1$ with $C_2$ on $l_2$ yields the chain $\{\neg(C_2 - l_2)[l_1]C_0\}\theta$ where $[l_1]$ is an A-literal and the literals in $(C_2 - l_2)$ may be reordered.*

The *reduction* operation implements a form of reasoning by contradiction: if $\neg P$ and $Q$ imply $P$, then we can reason by cases (of whether $P$ holds or not) to conclude that $Q$ by itself implies $P$. Either (1) $P$ is true (and thus $Q \Rightarrow P$ is true), or (2) $\neg P$ is true, in which case since $\neg P \wedge Q \Rightarrow P$, we can conclude that just $Q \Rightarrow P$. In either case we have established that $Q \Rightarrow P$.

Reductions allow a top B-literal to be removed if it unifies with the complement of an A-literal. Using the resulting chain from the previous example,

```
¬R(C,B), [Q(F(A),C)], [R(C,z)], P(A,z)
```

the (only possible) reduction operation yields the chain

```
[Q(F(A),C)], [R(C,B)], P(A,B)
```

via the binding list { `z→B` } .

In general,

**Definition 2 (Reduction)** *Given chain $C_1$ of the form $l_1C_0$ with top B-literal $l_1$ and A-literal $l_2$ of opposite sign to $l_1$ such that the atoms of $l_1$ and $l_2$ are unifiable with mgu $\theta$, the ME reduction operation yields chain $C_0\theta$.*

The *contraction* operation removes trailing A-literals, and is typically performed after every extension or reduction operation. In the example above, the chain

```
[Q(F(A),C)], [R(C,B)], P(A,B)
```

is contracted to

```
P(A,B)
```

When an A-literal is removed by contraction it has been proved by the ME procedure.[3]

In general

**Definition 3 (Contraction)** *Given chain $C_1$ of the form $l_1C_0$ with top A-literal $l_1$, the ME contraction operation yields chain $C_0$.*

### 2.4.2 Deduction in Horn and non-Horn theories

Model elimination is a refutation procedure. A deduction is a sequence of chains, beginning with the goal, where each chain except the first is the result of applying one of the inference operations to the preceding chain.

**Definition 4 (Deduction)** *A weak model elimination (weak ME) deduction of chain $K$ from a set $S$ of clauses is a sequence $K_1, \ldots, K_n$ of chains such that*

1. *$K_n$ is $K$*
2. *for all $i > 1$, $K_i$ is a chain derived from its parent chain $K_{i-1}$ by either extension (using a clause from $S^c$, the set of all contrapositives of each clause in $S$) or reduction, followed by as many applications of contraction as possible.*

**Definition 5 (Refutation)** *A weak ME refutation is a weak ME deduction of the empty chain from $S$.*

**Definition 6 (Proof)** *A weak ME proof of a conjunctive goal $G$ is a weak ME refutation where $K_1$ is a chain composed of the literals in $G$.*

**Definition 7 (Horn clause)** *A Horn clause is a clause with at most one positive literal.*

[3]Actually, the conclusions one can make by the contraction of an A-literal are somewhat more subtle. A removed literal is "proven" only for the context of the current chain. This thesis in fact is about exactly what can be inferred about a literal when it is removed in such an operation, under a variety of conditions.

If $\Delta$ is a set of Horn clauses and $\Delta$ implies $\phi$, then there is exists a weak ME proof of $\phi$ from the set $\Delta$, so any complete search of the space of possible deductions will discover the proof. (In the presence of recursive predicates or functional expressions the inference space may be infinite. This just illustrates the semidecidable nature of inference.) Hence weak ME is complete for Horn clause theories. In fact, the reduction operation is not necessary for completeness in Horn theories. Weak ME is also sound.

For completeness when using ME with non-Horn theories (or queries) the algorithm must be altered as follows:

1. The negated goal must be available for extension operations just like the rest of the database. It is typically inserted into the database for the duration of the proof. This allows one to conclude `P or ¬P` from the empty theory, for example.[4]

2. All database clauses (and the aforementioned negated goal) must be available for extension in the form of each possible contrapositive rule, which is not necessary in the Horn case.

This version of model elimination, using a minimal set of inference rules and without special reasoning for equality, is sound and complete for full first-order inference.

## 2.5 Refinements

In this section we briefly mention some common enhancements to the model elimination inference procedure, and how those enhancements interact with caching strategies.

### 2.5.1 Iterative Deepening

Iterative deepening [Kor85] is a common search strategy, where a space is searched with a maximum bound (say, the length of the path from the root to the subgoal). If no answer is found, the depth bound is increased and the search is restarted. Such a strategy combines the best of breadth-first and depth-first search: It will find the shortest proof first (like BFS), but only using space linear in the depth of the search (like DFS). In addition, since the fringe of such a space is generally larger than all the space above, repeatedly searching the top parts of the space tends not to be significant and thus the time complexity is essentially the same as BFS.

It is valuable to note that careful choice must be made of what to bound. In PTTP [Sti88], Stickel iterates on the number of subgoals used during the proof, with a few minor modifications. This kind of a bound interacts with many other aspects of the theorem proving search, such as intelligent backtracking.[5] Stickel himself notes[6] that

> In PROLOG, when solving the goals
>
> `P(x) and Q(y) and R(x)`

[4]Actually, if the goal is purely conjunctive, and if it is propositional, then model elimination is complete even if the negated goal is not added to the theory.

[5]See section 2.5.4 for more details about backjumping, a specific type of intelligent backtracking.

[6]The quoted text has been edited slightly.

if the goal `R` fails, `P` can be directly backtracked to because the computation of `Q` and the bindings for `y` it creates are irrelevant to the failure of `R`. However, in PTTP, the computation of `Q` can affect the success or failure of `R`, by using subgoals that reduce the "depth" bound available for solving `R`. The goal `R` might be made to succeed with alternative solutions of `Q` that require fewer subgoals in their solution.

Rather than attempt to deal with the complexities arising from a bound with such a character, the more common definition of a subgoal's depth will be used. It will be assumed that the iterated bound is on the length of the path between the root node and the current subgoal.

For simple success caching, no cache adornment is required. It is possible that such an addition to an inference algorithm will result in somewhat different search spaces for a given depth cutoff, although no incompleteness will result. For example, imagine that a subgoal `P` was initially proven with some remaining depth bound, and then later on, somewhere deeper in the proof space, another subgoal `P` occurred. Without caching, this second, deeper instance might fail (given the remaining depth bound). On the other hand, it is certainly the case that `P` follows from the database (since we have a proof of it), so it is valid to replace the search of the subspace below `P` with the cached success. It is possible that for a given problem this will result in a larger explored space for a given depth bound. For example, this deeper `P` might be part of a large and difficult conjunction that eventually fails; in the original case, the first conjunct `P` would fail given the depth bound, so the rest of the conjunction wouldn't even be explored. With caching, we may allow `P` to succeed, only to have the conjunction fail after a perhaps large amount of searching. With the same depth limit, the failure of `P` will be cheaper than the failure of a sibling of `P`, if the sibling is bushy.

The search space may shrink substantially too: perhaps this newly succeeding `P` results in a simple proof of the query at the given depth bound. In that case, the non-caching algorithm may be forced to explore a vast amount of the remaining space for a proof, whereas the caching version succeeds quickly. In any case, if for some reason it is crucial that the depth-limited search spaces of the caching and non-caching provers remain identical, then the cache can always be extended to also store the depth limit used for a given proof.

Cache entries for simple failure caching (*e.g.* "`P(B)` has no solution") need to be adorned with the depth cutoff in effect at the time. A failure cache entry is only successfully retrieved if the future subgoal has a cutoff less than or equal to the cutoff stored in the cache. Otherwise the search space must be re-explored (this time to a deeper depth).

### 2.5.2 Identical Ancestor Pruning

Identical ancestor pruning (IAP) is a powerful pruning heuristic in a model elimination search. Imagine, in the course of expanding an ME proof space for a particular goal `P`, that one were to encounter that same goal `P` again. One of two situations must hold:

1. There are no proofs of `P` from this database (because it doesn't logically follow).

2. Whether or not there is a proof using this second occurrence of `P`, there must be another proof of the original `P` not using it. Also, the different proof occurs at a shallower depth.

This is true because the second occurrence must eventually be proven somehow, so this recursion must bottom out. And then, by whatever proof this second occurrence succeeds, an analogous proof path must exist below the first occurrence of `P`. In either case, it is justifiable to prune the space below the second occurrence of `P`.

As an example, consider the database in table 2.2. A successful proof of the goal `G` is shown in figure 2.1. The goal `G` resolves with the first database rule to produce the conjunctive subgoals `P and C`. Working on the `P` subgoal first, it resolves with the second rule to produce `A`. There are two ways of expanding `A`; using the first of them results in the subgoal `C`. This resolves with the fifth rule to produce subgoal `P`.

| | | | |
|---|---|---|---|
| 1. | `G` | $\Leftarrow$ | `P and C` |
| 2. | `P` | $\Leftarrow$ | `A` |
| 3. | `A` | $\Leftarrow$ | `C` |
| 4. | `A` | $\Leftarrow$ | `L` |
| 5. | `C` | $\Leftarrow$ | `P` |
| 6. | `L` | | |

Table 2.2: Identical Ancestor Pruning database

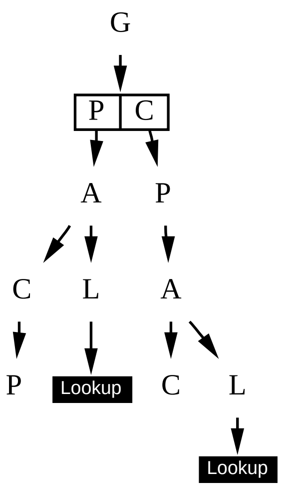


Figure 2.1: Successful use of IAP

Since this subgoal `P` has an identical `P` as an ancestor, IAP forces pruning of this branch of the search. There actually is a proof of the lower `P` subgoal (namely, the very proof we will soon discover for the first `P` subgoal), but we know by IAP that there will be a shorter proof of the ancestor `P` somewhere else. Backtracking to `C`, we also fail here. Backtracking again to `A`, there is another child. Resolving with rule 4 produces the subgoal `L`. This succeeds by lookup of sentence 6.

Now we return to the `P and C` conjunction, and begin working on the second conjunct, `C`. This resolves with rule 5 to produce `P`, then rule 3 (like before) to produce `C`. This branch also stops due to IAP. Backtracking to `A` and then resolving with rule 4 produces `L`, that again succeeds by lookup with rule 6. And thus the overall goal `G` is proven.

Unfortunately, IAP interacts poorly with failure caching. Just as ME reductions terminate a path successfully, but only in a limited context, so too does IAP terminate unsuccessfully, but again only in a limited context. Caching, on the other hand, attempts to reuse the results of a search in one part of the space by copying the results from a subgoal that appears in a different part of the space. Context-dependencies confuse such attempts.

Consider this same example, but with caching at the same time. This time, the second attempt to prove `C` uses the cached (failed) result from the first attempt, resulting in a failure to prove this overall goal. The complete (failed) search space is shown in figure 2.2, and the corresponding search of chains is in table 2.3.

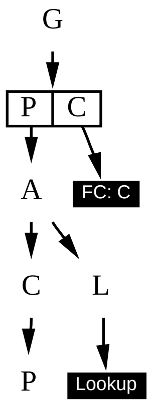


Figure 2.2: Incompleteness of failure caching with IAP

| No. | Chain | Failure cache |
|---|---|---|
| 1. | `G` | {} |
| 2. | `P C [G]` | {} |
| 3. | `A [P] C [G]` | {} |
| 4. | `C [A] [P] C [G]` | {} |
| 5. | `P [C] [A] [P] C [G]` | {} |
| 6. | `L [A] [P] C [G]` | {`P, C`} |
| 7. | `C [G]` | {`P, C`} |

Table 2.3: Incompleteness of failure caching with IAP

Another view is shown in table 2.3. The middle column is the sequence of chains searched, and the right column is the state of the failure cache. The search starts with the goal `G` as chain 1. A sequence of extension operations results in chain 5, which then fails due to IAP. The search then backs up to chain 3 (thus adding `P` and `C` to the failure cache), and so chain 6 is an extension of chain 3. The last chain also fails, because the top literal is in the current failure cache. When backtracking from this last failure, no further alternatives are found and so the search for a proof terminates (incorrectly) with failure.

This obvious problem can be fixed by propagating results with more information. With just failure caching, a subgoal could return to its parent the knowledge of whether or not it is still valid to put the parent in the failure cache. In a more complex inference system which might also include success caches, a three-valued result is necessary: success, failure, and IAP-pruned. This would allow IAP-caused unsuccessful terminations of a subgoal to avoid being placed in the failure cache, while at the same time permitting proved subgoals to be put in the success cache and completely explored failing subgoals to be added to the failure cache.

### 2.5.3 Pure Literal Elimination

A pure literal is one which has no complement anywhere in the database.[7] Such literals can never be removed by refutation resolution, and thus clauses with pure literals can never participate in the proof of any query.

Consider the database in table 2.4 with the goal `G`. Imagine that the subgoal `H` were very hard to prove; for example that it was equivalent to solving Fermat's Last Theorem. Notice that the subgoal `I` is impossible to prove, as it is not mentioned in the database anywhere except the first rule. After receiving the query, it is possible to recognize that the first rule is irrelevant, and thus avoid the computationally expensive attempt to prove the subgoal `H`. The proof space shown in figure 2.3 does not include any use of the first rule in the database.

| | | |
|---|---|---|
| `G` | $\Leftarrow$ | `H and I` |
| `H` | $\Leftarrow$ | `Fermat` |
| `Fermat` | $\Leftarrow$ | `...` |
| `G` | $\Leftarrow$ | `P` |
| `P` | | |

Table 2.4: Pure Literal Elimination database

### 2.5.4 Backjumping

When solving a conjunctive subgoal, simple theorem provers search depth-first through the space of solutions. The first conjunct is solved, the resultant bindings plugged in to the remaining conjuncts, and then the process iterates. When some intermediate instantiated subgoal has no solutions at all, then some form of backtracking must take place, to return to some previous choice point and

[7] Since complete inference for non-Horn database requires adding the negated query to the database, in order for a literal to be pure there must be no matching literal in the query either.

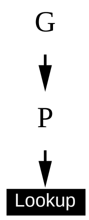


Figure 2.3: Proof with pure literals removed

attempt a different choice. In the typical search algorithms this is implemented with chronological backtracking, returning to the most recently made choice in the event of a failure.

This search can be unnecessarily expensive, however, if the root cause of a failure at some conjunct is a poor choice at a very early conjunct. The same failure at the downstream conjunct will be discovered over and over again.

Problems very similar to this have been addressed in the constraint satisfaction literature. Their problems are different in at least two important ways:

1. The domains of the variables are finite, and known explicitly in advance.

2. In addition to the query, there is a set of constraints with nice properties, *e.g.* variable consistency can be checked against them with a very low complexity algorithm.

Solving a conjunction in inference doesn't share these properties. Nonetheless, it is often the case that the insights behind various constraint satisfaction algorithms can yield analogous algorithms in theorem proving. Some of the examples in this thesis utilize a form of backjumping,[8] where the search backtracks past all (easily-computed) irrelevant conjuncts until locating one that actually impacts the detected failure. In essence, it is possible to discover short proofs that large portions of the search space will not have solutions either, given one particular failed solution.

The computation for determining "relevant" upstream conjuncts can be open-ended. In the simple form of backjumping used in this thesis, the explanations for a conjunct's failure don't include the value of the bindings themselves, but instead are merely the list of conjuncts which established some binding affecting the failed conjunct. (The binding values could be useful to avoid generating a new answer with the same failing bindings.)

### Example

An example can help illustrate backjumping.[9] Consider the query `G(w)` for the database in table 2.5. The proof space is shown in figure 2.4.

[8] Other candidate algorithms include GSAT, min-conflicts, dependency-directed backtracking, and dynamic backtracking [Gin93a].

[9] The example presented only requires database lookup to solve. Note that this is only for clarity of explanation; the extension to inference merely requires solving the individual conjuncts as subgoals rather than looking up their solutions in a database.

```
G(w)    ⇐  A(w) and B(j) and C(k) and D(x) and
           E(j,k) and F(w,x)
A(1)
A(6)
B(2)
C(3)
C(5)
D(4)
E(2,5)
F(6,4)
```

Table 2.5: Illustration of backjumping

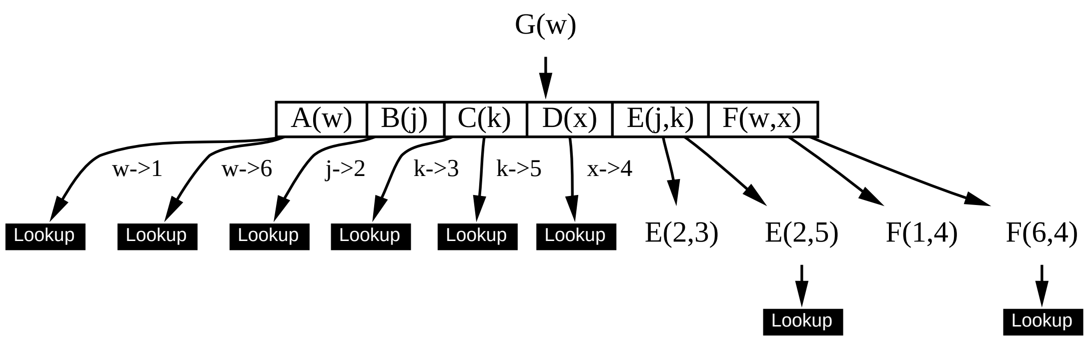


Figure 2.4: Illustration of backjumping

We begin by solving the `A(w)` conjunct, binding `w` to 1. Then `B(j)`, with `j`→2, then `C(k)`, with `k`→3, and then `D(x)` with `x`→4. At this point we attempt to solve `E(2,3)`, and fail, necessitating a backtrack. The explanations for the failure are `B` and `C`, the two previous conjuncts which bound variables appearing in `E`. We select the most recent one, `C`, and backjump to there, recording the remaining part of the explanation (conjunct `B`), in the set of explanations for `C`.

Continuing in the search, we call the `C(k)` inferential generator again, yielding the new binding `k`→5. Then `D` (having restarted, since we backed up over it) binds `x` to 4 again, and this time `E(2,5)` succeeds with no additional bindings.

Now `F(1,4)` fails. Computing the explanations (*i.e.* the conjuncts which caused `w` to be bound to 1 and `x` to 4), we get the set of conjuncts {`A,D`}. We back up to the most recent, namely `D`, and add conjunct `A` to `D`'s explanation set. Calling the generator on `D` yields no new answers, so we must back up again. No previous conjunct bound a variable in `D(x)`, but from the failure at `F` we had already put conjunct `A` on `D`'s explanation set, so we backjump to `A` and search for a new answer.

The generator for `A(w)` returns `w`→6 this time. The search from then proceeds much as before, including the rediscovery[10] of the failure to find a solution for `E(2,3)` and the backjump to `C` at that point. When we arrive at conjunct `F` the second time, we attempt to solve `F(6,4)`, and this succeeds. Thus the query is proved, with the bindings `w`→6, `j`→2, `k`→5, and `x`→4.

### 2.5.5 Goal Displacement

Loveland [Lov78, section 6.1] describes a problem reduction format which is similar to weak ME, and a "device of convenience" that can be added called *displacement*. If a goal $G$ is identical to a sibling of an ancestor[11] of `G`, then the goal under consideration may be said to succeed in the given context. This is valid because, in order for the overall proof to succeed through this line, the goal `G` must be established from the ancestor sibling position, and any such confirmation would be valid in both places.

Such a displacement results in a completion at the original location, which must be treated as carefully as completions by means of reduction. Nothing has been established about the immediate parents of the original `G`. They are not actually successful; they are mere allowed to proceed as though they were. Hence care must be taken in any caching scheme.

Displacement is permitted only if the ancestor sibling has not yet been expanded. This is necessary for the soundness of the procedure, for otherwise two branches might be accepted by each displacing a goal over to the other branch.

A use of displacement to correctly prove a top goal is shown in figure 2.5. This is a proof of `A` from the database in table 2.6. The subgoal `B` is completes successfully in the left branch by displacement, because it has a sibling ancestor in the conjunction `C and B`.

Displacement can be effective in the propositional case. For first-order inference, Loveland [Lov78, section 6.2] writes

> [Goal displacement] demands even more care in [first-order] use than for the propositional case. As before, its usefulness is simply in delaying pursuit of a goal. It does

[10] The attempt to avoid re-doing such work is the inspiration behind the dynamic backtracking algorithm [Gin93a].

[11] Displacement can be generalized to search for an identical subgoal which is the descendent of a sibling of an ancestor, instead of only the sibling itself (as long as the ancestor has only a single possible expansion on the path to the descendent).

```
A  ⇐  C and B
B  ⇐  D
C  ⇐  B
D
```

Table 2.6: Goal displacement

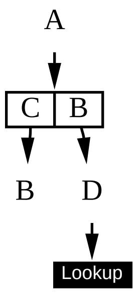


Figure 2.5: Goal displacement

collect identical goals together to expend only one effort in the attempt to establish the goal statement; otherwise a lemma mechanism is needed to accomplish essentially the same thing. However, with free variables present an instantiation may occur, which risks incorrectly restricting variables common to other goals so the other goals no longer can be established. We recommend avoiding any application of [goal displacement] where instantiation is involved.

# Chapter 3

# Horn-Clause Caching

Even in the limited context of theorem proving, the word "caching" often has wildly differing meanings, not all of which are clearly identified. In this chapter we describe some techniques (of increasing complexity), and assign names to them for the purpose of further discussion.

Caching is the basic idea of remembering the results of previous work, so as to decrease the cost of answering future queries. The computational advantage comes because a possibly exponential search is replaced by some simple lookup. This only results in an advantage if subgoals are repeated often enough, if the searches are hard enough, and if the overhead of checking the cache at every node expansion is small enough, that there is some actual gain in CPU time.

A cache entry may be either *partial* or *complete*. Partial entries provide true information about a node, but perhaps not all the information deducible from the knowledge base. This means that if the cached information is not sufficient to solve the problem, normal theorem proving must be started to try to solve the new subgoal. Such cache entries are often referred to as *lemmas*. They are similar to macro-operators in planning systems. Since they simply increase the breadth of the inference space, it is possible to observe adverse search effects where more nodes need to be explored to solve a problem with lemmas than without them.

Just the possible presence of adverse search effects, of course, doesn't mean such algorithms have negative utility. Storing only carefully chosen lemmas (*e.g.* unit clauses) can often substantially speed up problem solving. Nonetheless, much work in caching focusses on complete caching, where all necessary information about a node is stored in the cache. This simplifies the utility calculation as the caching scheme can only make the search space smaller, by pruning various subtrees that are summarized in the cache. The next sections describe a variety of complete caching schemes, which are the only type we will consider in this thesis.

## 3.1 Simple Success Caching

This is the most basic form of caching. If a literal is proven, then it is recorded in a table. If ever encountered as a subgoal again, the search effort that went into proving it originally can be replaced by simple lookup.

For example, if in one part of the space an attempt is made to prove `P(x)`, and after some inference the result "true for `x` bound to `A`" is discovered, then the literal `P(A)` will be entered into the cache. A subsequent need to prove a similar subgoal (say, `P(z)`) will result in the same inference process being repeated, as the needed subgoal is not in the cache. However, if sometime later a new `P(A)` subgoal is encountered, then a cache lookup can yield the answer "true" without the need for further search.

## 3.2 Simple Failure Caching

If a subgoal is completely explored, and no proofs are found, then the subgoal may be entered into a table recording failures. If the same subgoal is encountered later, then the failure can be noticed immediately via lookup rather than necessitating searching that complete subspace again.

Note, however, that special attention must be paid if some pruning techniques are used in conjunction with failure caching. For example, if identical ancestor pruning is enabled, then often subspaces will not be completely explored. It is not correct to place a subgoal in a failure cache if the search of the subspace below was pruned with IAP, as shown by the example in section 2.5.2.

## 3.3 Answer Caching

Answer caching is simply combining both simple success and simple failure caching.

## 3.4 Generalized Answer Caching

The matching function, for determining whether a new subgoal is already present in the cache, can be generalized to one-way unification without permitting adverse search effects. If `P(x)` is stored in the cache as successfully proven, and a new subgoal `P(A)` appears, then we can immediately conclude that this new `P(A)` also follows from the database. This is because some instance of the proof for `P(x)` will be a valid proof for `P(A)`.

The failure cache may be similarly generalized: if `Q(y)` has been cached such that the subspace below has been completely explored and no solutions were discovered, then we can conclude the failure of a new `Q(B)` subgoal without further search.

Consider the database in table 3.1. The proof space is shown in figure 3.1. The space was explored using a depth-first search strategy.

| | | |
|---|---|---|
| G(x) | $\Leftarrow$ | S(x) and T |
| S(x) | $\Leftarrow$ | F(y) |
| F(y) | $\Leftarrow$ | H |
| S(x) | $\Leftarrow$ | P |
| P | | |
| T | $\Leftarrow$ | F(1) |
| T | $\Leftarrow$ | S(2) |

Table 3.1: Generalized answer caching

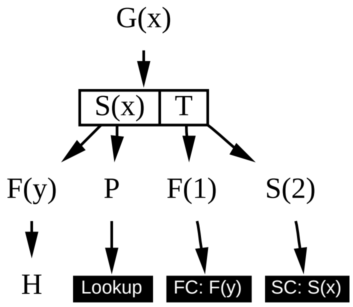


Figure 3.1: Answer Caching

After the left side of the tree (underneath the subgoal `S(x)`) had been explored, `F(y)` was in the failure cache and `S(x)` was in the success cache. (Also, of course, the failure cache contained `H` and the success cache contained `P`.)

When now exploring the space below the `T` conjunct, the first subgoal is `F(1)`. This fails immediately, because it is an instance of a literal in the failure cache (`F(y)`). The next subgoal is `S(2)`. This succeeds immediately, because it is an instance of a literal in the success cache (`S(x)`). And thus the overall goal has been proven.

Segre has used generalized answer caching in some reported experiments [SS93].

## 3.5 Subgoal Caching

A somewhat more sophisticated scheme is to record the actual generalized subgoals along with the set of known instances. When attempting to prove a subgoal, say `P(x)`, a cache entry is made. During the course of the proof, some set of answers will be returned, asserting that the subgoal is true for various bindings of the variable `x`, say `x`→`1` and `x`→`2`. If later some similar subgoal `P(y)` is encountered, then the cached answers can be transformed to yield the solutions `y`→`1` and `y`→`2` in the new context. Assuming that the original `P(x)` had been completely explored, it is also possible to conclude that there are no additional bindings for `y` that are solutions to `P(y)`.

Stickel [AS91] has used subgoal caching.

## 3.6 Generalized Subgoal Caching

Just like answer caching, it is possible to generalize the matching criteria for subgoal caching. Assume `P(x,y)` is a cached subgoal, and the solutions `P(1,10)`, `P(1,11)`, `P(2,20)`, and `P(2,21)` are found. Later, a different subgoal `P(1,z)` is encountered. Even though `P(1,z)` is not found in the subgoal cache, a more general version (`P(x,y)`) can be found. Thus the space below `P(1,z)` need not be explored, and the transformed answers `z`→`10` and `z`→`11` can be copied from the cache. Similarly, if a subgoal `P(3,w)` is encountered, that subgoal can be concluded to fail without the need to search the space below.

## 3.7 Lemmas

A *lemma* is a derived sentence (either a model elimination chain or a resolution resolvent) which is stored, and then used during a proof as though it were one of the given clauses. This is obviously related to caching, as it involves saving and then reusing previous work. By the stated form, in fact, it is more general, as lemmas may be whole clauses (disjunctions of literals), whereas the caches described in this chapter all store single literals.

The primary difference, however, is that lemmas are by nature a partial form of caching. (In fact, sometimes the term "lemma" is used to refer to any form of partial caching.) They increase the branching factor, by providing an additional database sentence to resolve on, but without eliminating the need to consider all the original sentences.

Thus it is possible to observe adverse search effects, where more nodes need to be explored to solve a problem with lemmas than without them. This is, of course, only loosely related to the utility of using lemmas, which is really the question of interest. There are two main differences:

1. The analysis of interest is really the average case performance, not the worst case performance.
2. The size of the search space is only an approximation to the time cost of the algorithm: the per-node expansion overhead is also an important factor.

As it turns out, storing only carefully chosen lemmas (*e.g.* unit clauses) can often substantially speed up problem solving. We do not consider partial caching in this thesis; more information on the impact of lemmaizing, and a collection of references, is available in Astrachan's thesis [Ast92, chapter 7].

## 3.8 Utility Analysis

Caching seems like an intuitively appealing augmentation of an inference system. It has the potential for recognizing redundancy in problem solving, and saving arbitrary amounts of time and space thereby. The types of caching we consider in this thesis all are purely pruning mechanisms: if a literal is found in the cache, then the search that would have taken place below it is aborted in favor of using the results from the cache.

This benefit is not without some cost. There is the space cost of storing the cache. There is also the overhead cost in time per node expansion, as a new subgoal is checked to see if it is already present in the cache. The real question of interest is whether the addition of a cache mechanism improves the utility of the inference system. Do the benefits outweigh the costs?

Unfortunately, it is difficult to say much in general about the utility of caching. For one thing, it is the average case which matters, but (like elsewhere in computer science) often only worst case analysis (or none at all) is available. In inference, coming up with useful problem distributions is problematic, making empirical data suspect. There exists a large collection [SSY94] of theorem proving problems, but to a large extent these are problems that have been generated to illustrate particular inference systems. It is difficult to say in what sense they represent objectively important problems.

There is a good amount of work in the operating systems literature on cache algorithms, for example page replacement policies in memory systems. One significant difference is the amount of items that can be stored: Memory systems have a polynomial number of pages, but inference explores an exponentially sized space. Gathering statistical information about an item in the case of inference is essentially as expensive as just storing the literal in the cache![1] Of course, an algorithm won't really have time to explore an exponential amount of the inference space, but that doesn't mean it is acceptable to use space linear in the amount of time spent. As Richard Korf writes [Kor92] on the subject of blind search techniques,

> Since best-first search stores all generated nodes in the Open or Closed lists, its space complexity is the same as its time complexity, which is typically exponential. Given the ratio of memory to processing speed on current computers, in practice best-first search exhausts the available memory on most machines in a matter of minutes, halting the algorithm.

It is thus far more effective to use algorithms whose space needs grow slowly with time spent. This is born out in practice, where inference algorithms with unbounded caches typically show negative utility over the same algorithms with no caching at all.

As a first step, the space needs of the underlying blind search mechanism need to be accounted for. Iterative deepening [Kor85] is a good solution to this problem: Its space requirements are only linear in the depth of the tree searched (the same as depth-first search), its time is only slightly worse than breadth-first search, and (like breadth-first search) it finds the shallowest solution in the space.

It would be nice if the caching module of an inference algorithm had similar characteristics. With good indexing, the per node overhead time costs can be made logarithmic in the size of the cache, although even this isn't good enough if the cache grows without bound. Fortunately, there is a simple alternative: the size cache can be limited to an arbitrary fixed size. Segre [SS93] has reported experimental evidence (on a large number of problems from the TPTP library [SSY94]) that bounded-sized caches show almost the same effectiveness (in terms of ability to prune the space searched) as infinite-sized caches, but with a well-bounded overhead on both total space used and per node time cost.

With a cache of bounded size, the question naturally arises of what to do when that bound is reached. The replacement policies from operating systems are applicable here, such as first-in-first-out (FIFO), least-recently used (LRU), and least-frequently used (LFU), or even just random replacement. Segre tested these strategies (and others) using cache bounds ranging from 10 to 1000 elements. (Each element could store either a success or a failure entry.) Interestingly enough, the cache replacement strategy appeared to have a fairly minor effect on the overall utility of the algorithm. LRU had a slight performance edge, but that effect was swamped by the influence of the cache size, which had a maximal utility at around 100 elements. Segre writes

> A very small cache bears much of the overhead costs yet yields little of the beneficial search effects. As the size grows larger, the beneficial effects of caching become evident but are eventually overwhelmed by increasing cache overhead.

It is dangerous to generalize experimental evidence from one domain to others, as the results may depend on the domain in some critical and as yet not understood way. Nonetheless, the approach of a bounded size cache seems promising.

[1] It may be possible to abstract literals in order to gather statistics about a much smaller abstract set. Arthur Keller [personal communication] has suggested recording data about literals of the form `P(x)` when the actual literals that are seen are `P(A)`, `P(B)`, etc. It is not clear how this could be turned into a useful cache system, however.

Plaisted [Pla94] also ran extensive experiments with various kinds of theorem provers and caching strategies. Some general observations were:

> The backward chaining strategies are goal-sensitive, but are mostly inefficient. Forward chaining strategies, though efficient for Horn clauses, are not goal-sensitive. All of the strategies that are goal sensitive have exponential duplication, except for the simplified problem reduction format with caching, the modified problem reduction format with caching, and clause linking with backward support. MESON and model elimination with caching and unit lemmas have this property, but the versions that are efficient on Horn clauses are not complete for general first-order clauses.

Backward-chaining provers with subgoal caching were clearly superior to either forward-chaining provers, or to backward-chaining provers without caching.

# Chapter 4

# Non-Horn Failure Caching

In the course of attempting a proof, imagine that we completely explore the space below some subgoal, and fail to find a proof of that subgoal. We thus place the subgoal in our growing cache of failed subgoals. Later, in the same proof effort, we encounter the subgoal again.

Ideally, we would like our inference algorithm to skip exploring the space below this second occurrence of the subgoal, using the intuitive justification that a previous attempt to solve the subgoal failed.

As it turns out, few researchers have considered using caching strategies in full first-order theorem proving. Most work on adding caching to inference procedures involve augmenting a Horn clause theorem prover.

What goes wrong in the non-Horn case? It appears to be a fundamental conflict between the caching ideal of context independence, and the inference reality that the complete version of model elimination for non-Horn theories requires the *reduction* operation, which is a context-sensitive operation as it involves resolving a goal with an ancestor goal.

For example, Astrachan and Stickel [AS91, Section 4.4] write

> In non-Horn problems A-literals can contribute to the solution of a goal via the reduction operation. Thus for non-Horn problems a goal cannot be considered in isolation, but must be considered in the context of the A-literals in the chain. These A-literals constitute an environment in which attempts to solve a goal are made. Since a goal template is intended to provide information about possible solutions for a given goal, and solutions are based on this environment, a template must somehow convey information regarding the A-literals that constitute the particular environment of a potentially cached goal.

Such an observation leads to the obvious idea that the cache itself be a more complex structure, containing not only the literal subgoal and its solutions (if any), but also the set of ancestor subgoals (*i.e.* A-literals) up the chain at this particular location in the proof space. This additional information would allow sound conclusions to be drawn about the presence of a literal in the cache, when encountering that same literal elsewhere in the proof space.

Indeed, this modification to caching is sound, but now the question of utility become paramount. Astrachan and Stickel continue:

> Examination of "chain dumps" for several non-Horn problems indicates that cache hits would be very rare and this method does not appear viable for non-Horn problems

> (Plaisted [Pla90] noted this as a potential problem with caching using model elimination). Consider a snapshot of the search tree for a particular theorem. For Horn problems, nodes in the search tree (unexpanded and-nodes that correspond to subgoals) can be considered for caching independently of the position at which they occur. In the naïve approach we have outlined for non-Horn problems, it is the root-to-node path that is considered for caching (where each node other than the last constitutes an A-literal). It is, perhaps, not surprising that such paths are not often candidates for cache retrieval.
>
> ...
>
> Our intuition leads us to believe that caching may still not be viable for any large class of non-Horn problems.

Stickel repeats this assessment in a later publication [Sti94]:

> Caching will surely be more complicated and less effective for the full model elimination procedure than for the Prolog subset on which it has been successfully tested. In the full procedure, solutions to a goal no longer depend on the goal formula alone, but also on its ancestor goals. Even if goals recur frequently, they may rarely recur with a set of ancestor goals that can be found in the cache.

## 4.1 An Example of The Problem

Before delving into the formal results, let's explore the intuition behind the reasoning. Imagine a situation where using a cache that ignores ancestors causes you to miss a reduction proof, for example in the database in table 4.1.[1]

| | | |
|---|---|---|
| G | $\Leftarrow$ | C or R |
| C | $\Leftarrow$ | $\neg$R |
| R | $\Leftarrow$ | C |

Table 4.1: Intuition: Missing proof

When trying to prove (without caching) the goal `G` using a Model Elimination backward-chaining theorem prover, the proof process might go as follows (the final space is in figure 4.1):

1. Resolve the goal with rule 1 to yield the disjunctive subgoals `C` and `R`.
2. Resolve the subgoal `C` with rule 2 to get $\neg$`R`.
3. Presumably failing to find any way to continue on this branch, back up to the second child of `G`, and resolve the subgoal `R` with rule 3 to get `C`.

[1]This database is propositional purely for purposes of clarity. The results in this thesis apply to general first-order logic, with full quantification.

4. This subgoal `C` (the second instance in the space) resolves with rule 2 exactly as it did before, yielding ¬`R`.

5. Using the *reduction* operation, the subgoal ¬`R` resolves successfully with its ancestor `R`, and the proof is complete.

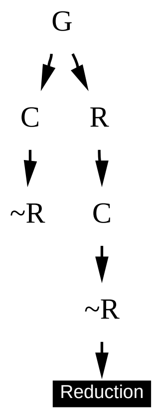


Figure 4.1: Without caching

The concern, naturally enough, is that with caching enabled, the first (failed) attempt to prove `C` will be saved. Then, when `C` is examined again as a child of `R`, the cached failure will be copied to the new instance of the subgoal, and the reduction proof of figure 4.1 will be missed. This failed proof space is shown in figure 4.2, where the dashed arrow from `R` to `C` indicates that some search has been replaced with lookup, and the solutions to this child of `R` are copied from the first time subgoal `C` was attempted.

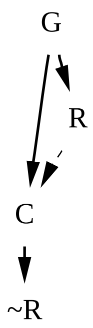


Figure 4.2: Failure with caching

## 4.2 The Intuitive Solution

It appears that we must take sets of ancestors into account when making a cache hit, in order to avoid this problem. Upon closer examination, though, this proposed counterexample fails, because in fact a proof of the goal is found.

For completeness of inference, we need to use all of the contrapositives[2] of each database rule. In particular, the first rule in the database can also be expressed as

$\neg$C and $\neg$R $\Leftarrow$ $\neg$G

We actually only need the second half of this contrapositive rule

$\neg$R $\Leftarrow$ $\neg$G

which makes the real proof space have a branch like figure 4.3, with or without caching. We find a proof using the original `C` branch, without ever needing to explore the `R` child of the top-level goal.

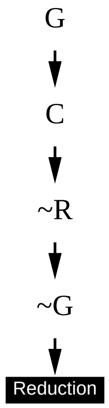


Figure 4.3: With contrapositives

As it turns out, such alternate proofs will always exist. A sketch of the proof construction is: Take the reduction proof which should have succeeded, and transform it into a new proof that will be found in any other context. The transformation takes the path from the root to the highest subgoal involved in the reduction, negates every subgoal on that path, turns the path upside-down, and reattaches it to the space below the original exploration of the cached subgoal. Conjunctive siblings are carried along in the same place as they were in the original proof, and they are proven the same way.

## 4.3 Failure Cache Semantics

The situation becomes a little more tricky when one tries to formalize it. The simple caching algorithm is clear:

[2]If $P$ implies $Q$, then it also must be the case that $\neg Q$ implies $\neg P$.

1. Search with normal model elimination
2. If you completely explore the space below some goal (in any context at all), and fail to find a proof, then add the goal to the failure cache.
3. If you later encounter a goal, and it is already in the failure cache, then prune the search below that goal (treat it as unsuccessful).

The point of this chapter is that this naïve algorithm is complete. The question, though, is what justifies the pruning of the search space at the second encounter of the goal?

Rather than talking about "proofs" of subgoals, we need to consider success, that can be achieved either through a deductive proof, or via a context-dependent reduction operation. We will call such a successful discarding of a subgoal a "completion."

What would justify our pruning strategy? We know how subgoals enter the failure cache: in some context, they are completely explored, and no completion is found. It would be nice if we could have semantics about what it means for a subgoal to be in the cache. When we come across a subgoal, and it is found in the failure cache, we then prune (with failure) the space below the new subgoal. This algorithm would be (logically) complete if we knew that no completion for the subgoal could be found in the pruned subspace. So a proposed theorem is: if there are no completions of a subgoal in one context, then there are no completions in any other context. The logical contrapositive of this claim is the following proposed theorem:

**Failed Theorem 8** *Whenever there is a completion of some subgoal in one context, there also exists a completion of the same subgoal in any other context of a given proof effort.*

Unfortunately, this proposed theorem is false as is shown by the counterexample in figure 4.4, which is the full proof space[3] for the goal `G` from the database in table 4.2.

| | | |
|---|---|---|
| `G` | $\Leftarrow$ | `R and C` |
| `R` | $\Leftarrow$ | `C` |
| `C` | $\Leftarrow$ | $\neg$`R` |

Table 4.2: Wrong: Completions here imply completions there

Note that, while there is a completion of `C` in the context on the left (using a reduction between `R` and $\neg$`R`), there is no completion of the second occurrence of `C` in the context on the right. Of course, `C` would not have been put in the failure cache in this example, but we can easily fix that. Simply exploring the `R and C` conjunction in the reverse order allows the failed search space for `C` to come first, as shown in figure 4.5. Of course, since the first conjunct now fails, we never explore the second conjunct,[4] and thus never notice the second subgoal `C` that would be in the failure cache

[3]The search strategy used was depth-first exploration with identical ancestor pruning.

[4]It is conceivable, however, that some best-first search strategy or else a multi-threaded prover might explore both conjuncts simultaneously, and therefore might actually encounter this phenomenon.

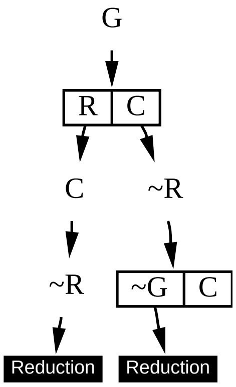


Figure 4.4: Wrong: Completions here imply completions there

by then, but actually has a completion in the second context. In any case, though, the query `G` does not follow from this database, and so we haven't lost any overall solutions even though the theorem 8 is not correct.

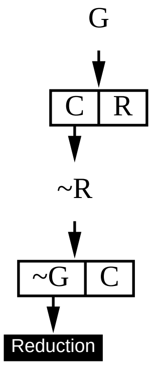


Figure 4.5: Wrong: Failure cache implies no completion (1)

Perhaps if we restrict our algorithm to searching only in a depth-first order, we can assign semantics to the failure cache by limiting the applicability to those subgoals we actually encounter. This would eliminate examples such as the previous one.

**Failed Theorem 9** *If there is no completion of a given subgoal in one context encountered during a proof effort, then there will be no completion of any other occurrence of the same subgoal in any other context encountered during the same proof effort.*

It sounds plausible, but now consider the database in table 4.3 with the same goal `G`, as shown in figure 4.6. Here the subgoal `F` is intended to indicate a subgoal which always fails.[5]

[5]The subgoal `F` is indeed not implied by this database in table 4.3, but in this case a simple check like pure literal elimination (see section 2.5.3) would prevent the second rule from being used, and thus would prune a large part

| | | | |
|---|---|---|---|
| 1. | `G` | $\Leftarrow$ | `C` |
| 2. | `G` | $\Leftarrow$ | `R and F` |
| 3. | `R` | $\Leftarrow$ | `C` |
| 4. | `C` | $\Leftarrow$ | $\neg$`R` |

Table 4.3: Wrong: Failure cache implies no completion (2)

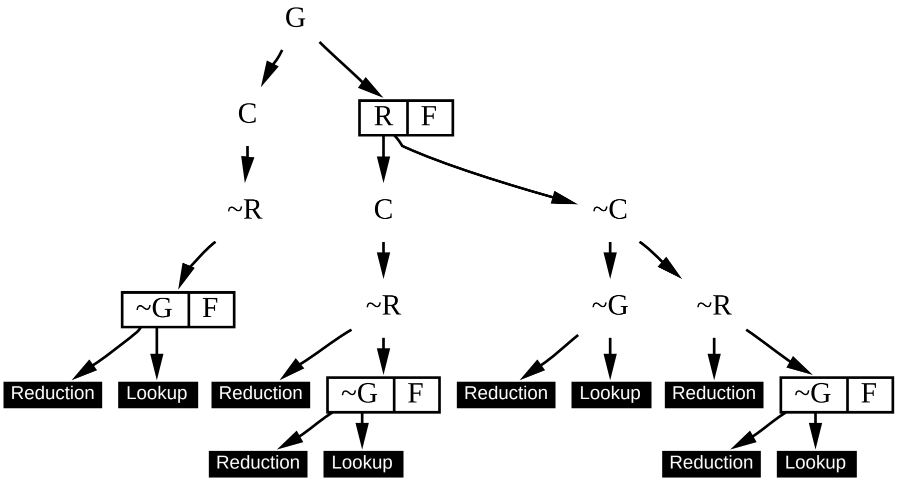


Figure 4.6: Wrong: Failure cache implies no completion (2)

Starting from the goal `G`, we can use the first rule to backchain to `C`. The contrapositive of the third rule gives $\neg$`R`. One of the contrapositives of the second rule, `R`$\Leftarrow \neg$`G`$\wedge$`F`, gives the conjunction $\neg$`G and F`. There are two backward inference steps possible from the $\neg$`G` subgoal at this point: a reduction with the root literal, and a simple lookup of the literal itself which has been temporarily stored in the database for the duration of the proof. (Recall that, for non-Horn problems, the negated goal must be added to the database in order to ensure completeness.) Using the successful reduction of $\neg$`G` with the root node `G`, we can continue with the $\neg$`G and F` conjunction. Unfortunately there is no inference possible from the `F` subgoal, and so this path terminates unsuccessfully. Backtracking through the conjunction,[6] the second solution to $\neg$`G` (namely, the database lookup) is tried, and then the second conjunct `F` is attempted again. As before, it fails. This whole subtree below `C` is unsuccessful, and thus `C` has no completion in this context.

Returning to the top goal `G`, we can also backchain on the second rule, resulting in the conjunctive subgoal `R and F`. From `R` there are two possibilities (using rule 3 and the contrapositive of rule 4). Exploring the `C` branch first, we backchain to $\neg$`R` using the fourth rule. The subgoal $\neg$`R` has two inference possibilities: a reduction with its grandparent `R`, and a resolution with the second rule. The reduction succeeds immediately (and thus there is a completion for `C` in this context) and that success propagates up to the `R and F` conjunction. Working on the second conjunct `F` results in immediate failure, and so another proof is attempted for the `R` conjunct. The search then continues with the second inference below $\neg$`R`, namely the resolution with the second rule to result in $\neg$`G and F`. As in the previous exploration of the similar conjunction, $\neg$`G` has two solutions (by reduction and by lookup), and the second conjunct `F` fails both times.

We then backtrack up the tree to the `R` ancestor which is part of the `R`$\wedge$`F` conjunction, and next explore the result of resolving with the contrapositive of rule 4, namely $\neg$`C`. This subgoal has two disjunctive children ($\neg$`G` and $\neg$`R`), which after searching the subtree below result in three solutions: A reduction between the $\neg$`G` child and the root `G` literal, a successful lookup of the $\neg$`G` child in the database, and a reduction between the $\neg$`R` child and and the `R` parent. None of these completions of $\neg$`C` result in a proof for the overall query, as the `R`$\wedge$`F` conjunction continues to fail on `F`.

The point to note is that while the first occurrence of the subgoal `C` in the space has no completion, the second occurrence (below the parent `R`) does have a completion, and so our new proposed theorem is also not correct. As before, though, the caching algorithm is still logically complete: `G` does not follow from this database, and so no proof should be found.

## 4.4 Formal Results

For the initial presentation, the main results will shown in a framework where

1. the query and database are propositional

2. the goal is a single literal

---

of the space in this example. This is not a solution to our overall problem, however, because more complicated examples (involving failed spaces beneath the subgoal `F`) can easily be constructed that exhibit the same behavior as this example even with a check for pure literals.

[6] A more clever backtracking algorithm such as backjumping (see section 2.5.4) could avoid failing on `F` a second time. For illustration purposes, the entire inference space is shown in figure 4.6.

3. the caching strategy is simple failure caching

The extensions to conjunctive goals, first-order theories, and generalized failure caching come later.

### 4.4.1 Definitions

Failure caching will work if none of the literals in the cache appear in any proof of the goal. Hence a set of interest is all the literals that do appear in some proof.

**Definition 10 (Proof literals)** *Given a set of clauses $\Delta$ and some goal $G$, the set of proof literals $\mathcal{P}(G, \Delta)$ is the set of all literals $l$ such that $l$ appears in some weak* ME *proof of $G$ from $\Delta$.*

In non-Horn inference, literals and their negations are tightly coupled. Thus we will generally be concerned about about the augmented set of literals which is closed under negation.

**Definition 11 (Augmented proof literals)** *Given a set of clauses $\Delta$ and some goal $G$, the augmented set of proof literals $\mathcal{P}^+(G, \Delta)$ is the set of all literals $l$ such that either $l$ or its negation appears in some weak* ME *proof of $G$ from $\Delta$.*

The important property about a literal that will concern us is that of success during a proof, which we will call a *completion*. In the Horn case there is a correspondence between logical inference ($\Delta \models l$, *i.e.* that $l$ follows from $\Delta$) and proofs ($\Delta \vdash l$, *i.e.* $l$ can be derived during a proof attempt). This notion must be generalized in the non-Horn case. There a literal can be successful during a proof attempt if either there is a direct proof of the literal (because it logically follows from the database), or else if it is provable in the context by use of some reduction operations. It is this successful discarding of a literal which is of interest during a proof attempt.

**Definition 12 (Completion)** *A literal $l$ has a completion in a context of attempting to prove $G$ from the set of sentences $\Delta$, if there is some sequence of chains $C_1, \ldots, C_n$ such that*

1. *There is some valid deduction from the goal chain consisting of just the goal $G$, to the chain $C_1$, using the inference operations of weak* ME *(reduction, contraction, or extension from $\Delta$).*
2. *The top literal of chain $C_1$ is $l$.*
3. *For $1 \leq i < n$, chain $C_{i+1}$ is the result of applying one of the valid inference operations from chain $C_i$.*
4. *Chain $C_n$ is identical to chain $C_1$ with the top literal $l$ removed.*

We will refer to the concept of a literal being encountered somewhere in the search space when attempting to prove some goal, by saying that it "occurs below" the goal. The results in this section are about concluding properties of a literal encountered in one context, based on the observed properties of the same literal encountered in some other context.

**Definition 13 (Literal occurs below)** *Literal $l$ is said to occur below $G$ (when attempting to prove $G$ from some set of sentences $\Delta$) if there is some chain $C$ such that*

1. *There is some valid deduction from the goal chain consisting of just the goal $G$, to the chain $C$, using the inference operations of weak* ME *(reduction, contraction, or extension from $\Delta$).*

2. *The top literal of chain $C$ is $l$.*

Finally, many of the proofs proceed by induction on the depth of a literal. This concept (like the phrase "occurs below") comes from the representation of the complete search space as a tree (or graph), with the root node being the goal.

**Definition 14 (Literal depth)** *A literal in some context is said to be at depth $d$ if the length of the path from the literal to the goal is $d$. (The goal itself is at depth 0.) Equivalently, in the notation of chains, a literal $l$ in some chain $C$ is at depth $d$ if*

1. *Literal $l$ is the top literal of chain $C$.*

2. *Literal $l$ is a B-literal.*

3. *The number of A-literals in chain $C$ is $d$.*

### 4.4.2 Propositional theories

We first establish two lemmas before proceeding to the main result: First, that all literals which appear in some proof of $G$ have completions everywhere below $G$, and second, that the negations of all such literals also have completions everywhere below $G$.

**Lemma 15 ($L$ has completions)** *Let $\Delta$ be a set of propositional sentences and let $G$ be a single propositional literal such that $\Delta \models G$. Let $l$ be a literal in $\mathcal{P}(G, \Delta)$,* i.e. *a literal that appears in some weak* ME *proof of $G$. Then everywhere $l$ occurs below $G$, it will have a completion involving only proof literals from the set $\mathcal{P}^+(G, \Delta)$.*

**Proof.** We will establish this by induction on the depth of the literal $l$ in the weak ME proof of $G$.

**Base case.** ($d = 0$) The only literal at depth 0 is $G$ itself. Since $G$ follows from $\Delta$, there is a weak ME proof of $G$ in isolation. As this proof can't possibly rely on any context above $G$ (since the goal literal has no context above it), this same completion is still valid anywhere else that $G$ is encountered. By the same token, every literal used in the completion occurs in a proof of $G$ (and thus is in $\mathcal{P}^+(G, \Delta)$), namely the very proof that the whole completion was taken from.

**Inductive case.** Given that we know, for all literals $a_i$ at depths less than $d$, that $a_i$ has a completion anywhere below $G$, we must show that some new literal $l$ (which appears at depth $d$ in some proof of $G$) also has a completion anywhere below $G$. In both cases, the completions must only involve literals from the set $\mathcal{P}^+(G, \Delta)$.

If $l$ appears in some different context below $G$, then there is some chain of the form

$$l \;\ldots\; [G]$$

derivable from the goal chain $G$. Since we know that $l$ also appears in a proof of $G$, it must have some completion at the point it appears in the proof of $G$. This completion is of the form

$$
\begin{array}{r}
l\ \texttt{[}a_n\texttt{]}\ S_{n,1}\ \dots\ \texttt{[}a_i\texttt{]}\ \dots\ \texttt{[}G\texttt{]} \\
\dots \\
\texttt{[}a_n\texttt{]}\ S_{n,1}\ \dots\ \texttt{[}a_i\texttt{]}\ \dots\ \texttt{[}G\texttt{]}
\end{array}
$$

This last sequence of chains is a completion of $l$ in the context of the known proof of $G$. For each operation in the sequence, we can map it to one or more valid operations in the new, non-proof context. By virtue of the mapping, for every chain in the completion of $l$ occurring in the proof of $G$, there exists a chain in the new context which has an identical sequence of literals from the occurrence of $l$ to the top of the chain.

1. If the proof operation is an extension operation, then that same extension operation is still valid in the new context. In addition, all literals appearing in the extension also appear in the analogous place in the weak ME proof of $G$, and hence each such literal is in $\mathcal{P}^+(G, \Delta)$.

2. If the operation is a contraction, then the same contraction applies. Since no new literals are added to the chain, the condition on $\mathcal{P}^+(G, \Delta)$ is not violated.

3. If the operation is a reduction, where the ancestor literal occurs at a depth $d$ or more in the proof of $G$, then that same reduction operation is still valid. This is because, by the mapping we are constructing, every literal more recent than $l$ occurs identically in the chains of both the completion in the proof context as well as this new completion in the new context.

4. The only remaining possibility is a reduction operation to an ancestor which occurs at a depth $d-1$ or less. In the proof of $G$ there must be some ancestor $a_i$ occurring between $l$ and $G$, which resolves with the top literal $\neg a_i$ by reduction during the proof of $G$. We need to show that this same $\neg a_i$ has a completion in the new context, and a completion which only relies on literals from $\mathcal{P}^+(G, \Delta)$.

   (A graphical representation of this situation is shown in figure 4.7. The left branch shows the occurrence of $l$ in a proof of $G$. The completion of $l$ in that context uses a reduction between $\neg a_i$ and the ancestor $a_i$. On the right branch, $l$ is encountered in some different context. The dotted arrows on the figures in this section are meant to indicate that there is an arbitrary tree between the two nodes. The shaded triangles represent the and-tree completion of $l$ in the proof context, which is duplicated in the new context.)

   We show that $\neg a_i$ has a completion by a second induction, on the depth $d'$ of the ancestor literal $a_i$. For each possible ancestor literal, we show that the negation of the ancestor literal has a completion in the new context.

   **Base case 2.** ($d' = 0$) In this case, the ancestor literal $a_i$ is just the goal $G$ itself. $\neg G$ has a completion everywhere (including the new context), via a simple reduction with the top goal $G$.

   **Inductive case 2.** Given that, for every literal in the proof at a depth less than $d'$, the negation of the literal has a completion in the new context, we need to show that some new literal $a_i$ at depth $d'$ also has a completion in the new context.

   In order for $a_i$ to have appeared in the proof of $G$, there must be some database rule

   $a_{i-1} \Leftarrow a_i$ `and` $S_{i,1}$ `and` $\dots$ `and` $S_{i,m}$

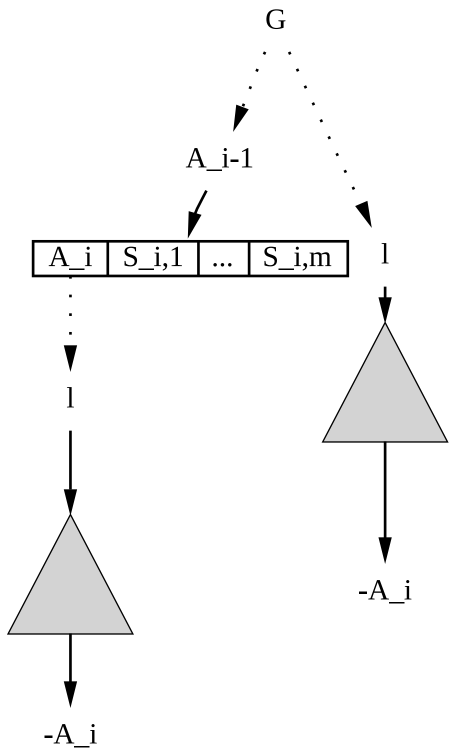


Figure 4.7: Does $l$ have a completion everywhere?

which was used via an extension operation. Since all contrapositives of every rule must be accessible in non-Horn model elimination, another valid rule is

$$\neg a_i \Leftarrow \neg a_{i-1} \texttt{ and } S_{i,1} \texttt{ and } \ldots \texttt{ and } S_{i,m}$$

We can apply this contrapositive rule to the occurrence of $\neg a_i$ in the new context. Note that each of these literals is a member of $\mathcal{P}^+(G,\Delta)$: $\neg a_{i-1}$ because $a_{i-1}$ is in the proof of $G$, and the sibling $S$ subgoals because they occurred directly in the proof of $G$.

In the new context, then, we must show that each of the literals $\neg a_{i-1}$, $S_{i,1}$, ..., $S_{i,m}$ has a completion using only literals from $\mathcal{P}^+(G,\Delta)$. The sibling $S$ literals do because of the main induction hypothesis: they occur at a depth less than $d$ in the proof of $G$, and thus have completions everywhere. (The $S$ literals occur at the same depth as the $a_i$ ancestor. The lowest that ancestor could be is if it were $a_n$, the immediate parent literal of $l$ itself. Given that $l$ is at depth $d$, this makes $a_n$ and thus all of the $S$ sibling literals occur at a depth which is at most $d-1$.)

For the $\neg a_{i-1}$ literal, we can apply the secondary induction hypothesis. $a_i$ appears in the original proof at depth $d'$, and so $a_{i-1}$ appears in the original proof at depth $d'-1$. The secondary induction hypothesis tells us that, for every literal appearing at a depth less than $d'$, the negation of that literal has a completion in the new context (using only literals from $\mathcal{P}^+(G,\Delta)$).

(The completion constructed in this new context is shown in figure 4.8.)

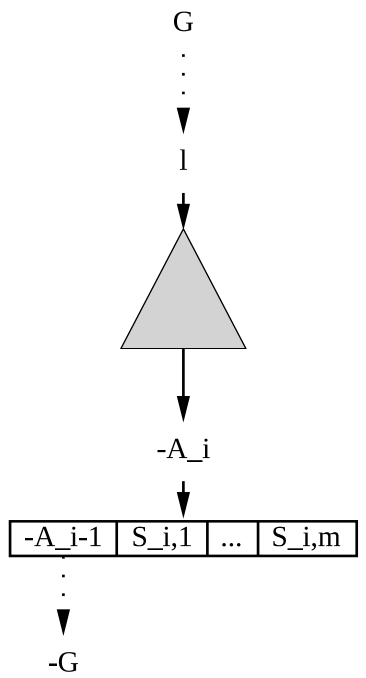


Figure 4.8: $l$ has a completion everywhere

We have demonstrated a sequence of operations resulting in the completion of a $\neg a_i$ subgoal in the new context, which duplicate the effect of the reduction operation to an $a_i$ ancestor that occurred in the proof of $G$.

We have now shown that every operation in the completion of $l$ during the proof of $G$ can be mimicked by one of more operations in any different context where $l$ occurs below $G$. For these derived completions, only literals in $\mathcal{P}^+(G, \Delta)$ are necessary. ∎

**Lemma 16 ($\neg L$ has completions)** *Let $\Delta$ be a set of propositional sentences and let $G$ be a single propositional literal such that $\Delta \models G$. Let $l$ be a literal in $\mathcal{P}(G, \Delta)$,* i.e. *a literal that appears in some weak* ME *proof of $G$. Then everywhere $\neg l$ occurs below $G$, it will have a completion involving only proof literals from the set $\mathcal{P}^+(G, \Delta)$.*

**Proof.** By induction on the depth of $l$ in the proof of $G$.

**Base case.** ($d = 0$) The only literal at depth 0 is $G$ itself. The literal $\neg G$ has an immediate completion anywhere below $G$, via a reduction with the initial goal literal $G$ itself.

**Inductive case.** Given that for every literal $a$ of depth less than $d$, $\neg a$ has a completion everywhere, we must show that for literal $l$ at depth $d$, $\neg l$ also has a completion everywhere, using only literals from $\mathcal{P}^+(G, \Delta)$.

Since $l$ occurs in a proof of $G$, there must be some database rule of the form

$a_n \Leftarrow l$ `and` $S_1$ `and` ... `and` $S_m$

We can thus use the contrapositive rule

$\neg l \Leftarrow \neg a_n$ `and` $S_1$ `and` ... `and` $S_m$

to construct a completion for $\neg l$ in the new context. A single extension operation on $\neg l$ in the new context will result in the conjunctive subgoals $\neg a_n \wedge S_1 \wedge \ldots \wedge S_m$. We need to show that each of these has a completion using only literals from $\mathcal{P}^+(G, \Delta)$.

The literal $\neg a_n$ has a completion in the new context (using only literals from $\mathcal{P}^+(G, \Delta)$) by the induction hypothesis. Each of the $S$ sibling literals also has a completion in the new context, because of Lemma 15. Thus $\neg l$ has a completion in every context below $G$, using only literals from $\mathcal{P}^+(G, \Delta)$. ∎

Given these two lemmas, it is now straightforward to establish the main result, that failure caching will not prune any proofs of $G$.

**Definition 17 (SFC ME)** *Simple failure caching (SFC) model elimination (ME) is a version of weak* ME *augmented with a failure cache.*

1. *If a chain $C$ with top literal $l$ has no child (because no extension, contraction, or reduction operation is valid for $C$), then $l$ is added to the failure cache and $C$ is said to fail.*
2. *If all child chains of some chain $C$ with top literal $l$ are explored and fail, then $l$ is added to the failure cache and $C$ is said to fail.*

3. *If a chain $C$ with top literal $l$ is encountered, and $l$ is present in the failure cache, then chain $C$ fails with no further search.*

**Theorem 18 (Completeness of failure caching)** *Let $\Delta$ be a set of propositional sentences. Let $G$ be a single propositional literal, such that $\Delta \models G$. Then every complete search of the* SFC ME *space will discover some* SFC ME *proof of $G$ from $\Delta$.*

**Proof.** Since weak ME is complete, we know that there exists a weak ME proof of $G$ from $\Delta$. Since SFC ME is a pruning strategy, we need only show that the known weak ME proof of $G$ can not possibly be pruned.

This is true because no literal which is added to the SFC ME failure cache can be part of any proof of $G$ (and neither can its negation). We will show this by induction on the number of items, $n$, in the failure cache.

**Base Case.** ($n = 0$) With no items in the failure cache, it follows immediately that none of them are in any proof of $G$.

**Inductive case.** Let there be $n-1$ literals in the failure cache, such that for each such literal $f$, neither $f$ nor $\neg f$ occurs in any proof of $G$. We need to show that this is also true for literal $l$, the $n^{\text{th}}$ literal added to the failure cache. Since $l$ is being added to the failure cache, we know by the definition of SFC ME that it must have been encountered in the attempt to prove $G$, and it must have failed to have a completion in the context in which it was encountered.

**Part 1.** ($l$ not in proof) By Lemma 15, we know that if $l$ were in some proof of $G$, then it would have a weak ME completion in any context (including this one, where $l$ failed) which only involves literals from the set $\mathcal{P}^+(G, \Delta)$. However, by the inductive hypothesis, none of the literals in the set $\mathcal{P}^+(G, \Delta)$ are in the failure cache, so this weak ME completion would not be pruned during an SFC ME search. Thus the completion is still valid, and $l$ would not have failed in this context. Since it did fail, $l$ cannot occur in any proof of $G$.

**Part 2.** ($\neg l$ not in proof) Analogously, by Lemma 16 we can deduce that $\neg l$ cannot occur in any proof of $G$.

This establishes the inductive case, and thus the induction. Thus we have shown that no literal in the SFC ME failure cache can be part of any proof of $G$ (and neither can the negation of any such literal).

Hence if there is some weak ME proof of $G$, it will never be pruned by an SFC ME search. Since weak ME is complete for propositional inference, so is SFC ME. ∎

It is useful to note the connection between completions of $l$ and $\neg l$, when $l$ occurs in some proof of the goal. If a literal fails to have a completion at some point, then neither it nor its negation can appear in any proof of the goal. This is somewhat counterintuitive, since in the Horn case if a literal cannot be proven then it is typical that its negation is true. In the non-Horn case, though, you can also assume that its negation cannot be proven!

**Corollary 19 (Negated Failure Cache)** *In the course of an* SFC ME *search, if a literal $f$ appears in the failure cache and $\neg f$ is encountered, the space below $\neg f$ may be pruned without sacrificing completeness.*

**Proof.** Completeness is not sacrificed, because $\neg f$ does not appear in any proof of the goal. If it did, then by Lemma 16 its negation (namely, $f$ itself) would have completions everywhere, and thus $f$ could not have been added to the failure cache. ∎

**Corollary 20 (Ground Atomic Completeness)** SFC ME *is complete when all literals in the proof space are ground atomic.*

**Proof.** By inspection of the proofs for Theorem 18 and Lemmas 15 and 16, we can see that those proofs apply unchanged to the ground atomic case. ∎

**Theorem 21 (Soundness of failure caching)** *If there is an* SFC ME *proof of $\phi$ from $\Delta$, then $\Delta \models \phi$.*

**Proof.** Since SFC ME is just a pruning strategy on weak ME proofs, every SFC ME proof is also a weak ME proof. Since weak ME is sound, so is SFC ME. ∎

Now that completeness[7] has been proven for the special case of propositional theories with single literal goals, we proceed to generalize the result. These proofs are essentially identical to the previous results, with the additional complexity appearing at just a few points. The proofs will thus be presented as modifications to the previous proofs. (The results were originally presented in the simplified framework for ease of presentation.)

### 4.4.3 Conjunctive goals

The first generalization will be for when the top goal is conjunctive. Since one could always add a rule resulting in a virtual top goal of a single literal, it is not surprising that theorem 18 generalizes. For example, if the desired query were

$G_1$ `and ... and` $G_n$

it is a simple matter to add the rule

`V` $\Leftarrow$ $G_1$ `and ... and` $G_n$

to the database (where `V` is some relation not appearing in the database) and then instead ask the query "V".

Even this is not necessary, however. Lemmas 15 and 16, and Theorem 18, can all be extended to the case of conjunctive queries. Since these proofs are similar to the proofs already given, only the additional line of reasoning will be presented.

**Lemma 22 ($L$ has completions)** *Lemma 15 is still valid if the goal $G$ is conjunctive,* i.e. $G = G_1 \wedge \ldots \wedge G_n$.

**Proof.** We use the same induction as before, on the depth of $l$.

**Base case.** This case is essentially identical to the previous situation. Rather than having only a single literal at depth 0, we have a the $n$ literals $G_1, \ldots, G_n$. If some $G_i$ is encountered elsewhere in the space, does it have a completion using only literals from $\mathcal{P}^+(G, \Delta)$? If the overall conjunctive goal $G$ follows from $\Delta$, then there must be some proof of $G_i$, and that same proof can serve as a

[7] The soundness result of Theorem 21 already applies to the full first-order case.

completion for $G_i$ in this new context. (And, since it comes from a proof of $G$, all literals in the completion must be in $\mathcal{P}^+(G,\Delta)$.)

**Inductive case.** We need to show the new literal $l$, which appears at depth $d$ in some proof of the conjunctive goal $G$ (say, below $G_j$), has a completion everywhere below $G$ (say, below $G_i$). The new context for $l$ in which we must demonstrate a completion is a chain of the form

$$l \ \ldots \ [G_i] \ G_{i+1} \ \ldots \ G_j \ \ldots \ G_n$$

where all the conjuncts in the goal before $G_i$ have already been solved, $G_i$ is currently being proved (and hence is an A-literal), and the remaining conjuncts in the goal have yet to be solved (and thus remain in the chain as B-literals). This chain is derivable from the goal chain.

The literal $l$ also appears (at depth $d$) in a different chain, one part of a proof of $G$. That chain is of the form

$$l \ [a_{d-2}] \ S_{d-2,1} \ \ldots \ [a_i] \ \ldots \ [G_j] \ G_{j+1} \ \ldots \ G_n$$

where there are $d-2$ ancestor literals between $l$ and the current goal conjunct, $G_j$. (This situation is also shown in figure 4.9.) Since this is a proof of $G$, there is a completion of $l$ in this context. As before, we map this known completion to one that must then exist in the new context.

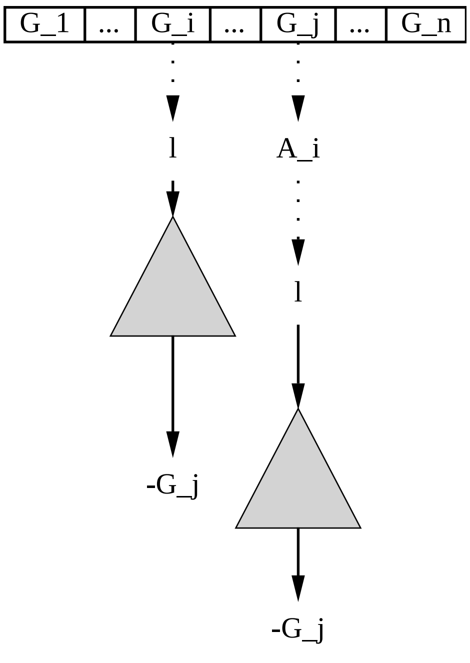


Figure 4.9: Does $l$ have a completion with a conjunctive goal?

Each portion of the mapping proceeds as before until we get to the last one, where a reduction operation occurs between some literal $\neg a_i$ below $l$, and an ancestor $a_i$ of $l$. A second induction, on the depth $d'$ of the ancestor $a_i$, is used (as before) to show that the $\neg a_i$ subgoal has a completion.

**Base case 2.** In this case, the ancestor literal $a_i$ is the parent goal conjunct of $l$, namely $G_j$. We need to show that $\neg G_j$ has a completion (using only literals from $\mathcal{P}^+(G,\Delta)$) everywhere below any of the conjuncts of $G$.

If this new $\neg G_j$ occurs below the $G_j$ goal literal, then it has a completion via an immediate reduction with the top parent. Otherwise, it occurs below some other goal conjunct, $G_k$. Recall from non-Horn model elimination that the negated goal must be available for extension operations. The goal is

$G_1$ `and ... and` $G_j$ `and ... and` $G_n$

so that the database clause becomes

$\neg G_1$ `or ... or` $\neg G_j$ `or ... or` $\neg G_n$

One of the contrapositives of this clause is

$\neg G_j \Leftarrow G_1$ `and ... and` $G_n$

(where $G_j$ is missing from the body of the rule). We can apply this extension operation to the $\neg G_j$ subgoal, resulting in the new subgoal conjunction

$G_1$ `and ... and` $G_n$

*i.e.* all the goal conjuncts except $G_j$. (This situation is shown in figure 4.10.) By the main induction hypothesis, each of these satisfies our criteria of having completions everywhere using only literals from $\mathcal{P}^+(G, \Delta)$.

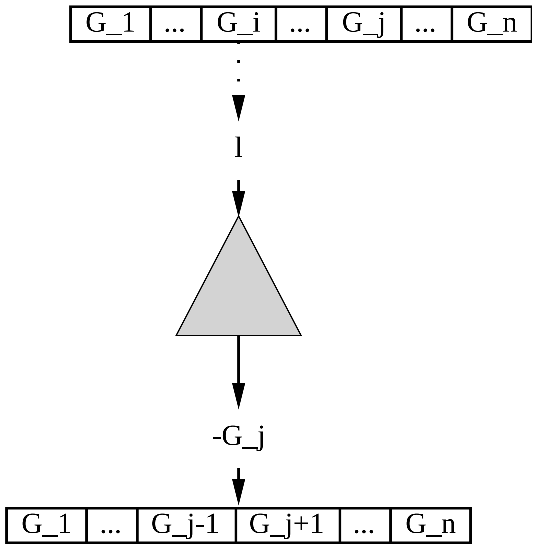


Figure 4.10: $l$ has a completion with a conjunctive goal

**Inductive case 2.** The proof of the secondary inductive case doesn't rely on $G$ being a single literal, and so it may be used unchanged in the case of a conjunctive goal. Hence we have shown that Lemma 15 is still valid if the goal is a conjunctive set of literals. ∎

**Lemma 23 ($\neg L$ has completions)** *Lemma 16 is still valid if the goal $G$ is conjunctive,* i.e. $G = G_1 \wedge \ldots \wedge G_n$.

**Proof.** The inductive case of the proof of Lemma 16 only relied on $G$ being a single literal in its use of Lemma 15. Substituting Lemma 22 for Lemma 15, we have a proof of the inductive case which is valid for conjunctive goals as well.

For the base case (that the negation of any literal at depth 0 has a completion everywhere), we can use the negated goal in much the same way as the proof for Lemma 22.

The only literals at depth 0 are the goal literals $G_1, \ldots, G_n$. We need to show that a negated version of any of them, say $\neg G_i$, has a completion everywhere in this space, using only literals from $\mathcal{P}^+(G, \Delta)$. Since the negated goal is available as a database sentence, we can apply an extension operation using the rule

$\neg G_i \Leftarrow G_1$ `and ... and` $G_{i-1}$ `and` $G_{i+1}$ `and ... and` $G_n$

to give us the conjunctive subgoals $G_1 \wedge \ldots \wedge G_n$ excluding $G_i$. Each of these has a completion in this context, using only literals from $\mathcal{P}^+(G, \Delta)$, by virtue of Lemma 22. ∎

**Theorem 24 (Completeness of failure caching)** *Theorem 18 is still valid if the goal $G$ is conjunctive,* i.e. $G = G_1 \wedge \ldots \wedge G_n$.

**Proof.** The proof of theorem 18 didn't rely on the single-literal nature of the goal $G$, except via the use of Lemmas 15 and 16. The same proof, using Lemmas 22 and 23 instead, will suffice to establish the completeness of failure caching in the case of a conjunctive goal. ∎

Similarly, Corollaries 19 and 20 can be extended to the case of conjunctive goals.

### 4.4.4 First-order theories

As promised at the beginning of this section, we now generalize the completeness result to the case of first-order predicates. The technique is similar to the use of a lifting lemma introduced by Robinson [Rob65], and used, for example, by Lloyd [Llo87].

The proofs in the section essentially duplicate those of section 4.4.2, with the additional notion of a binding list to map completions from one part of the space to completions in another part. The proofs here are expressed more concisely than previously, under the assumption that the reader is already familiar with the propositional versions. In addition, the proofs here are spread out into a greater number of lemmas.

**Definition 25 (Instance)** *A literal $P$ is said to be an* <u>instance</u> *of a more general literal $P'$ if there exists some (possibly empty) binding list $\theta$ such that $P = P'\theta$.*

The first lemma corresponds to the inside induction of Lemma 15. In order to separate it from the main proof, the conclusion must be a conditional. This is precisely the condition, however, that the outside induction of the main proof will establish.

**Lemma 26** (**$\neg L$ has completions if $L$ has completions**) *Let $\Delta$ be a set of clauses and $G$ be a conjunction of literals such that $\Delta \models G$. Let $l'$ be a literal in $\mathcal{P}(G, \Delta)$, so that $l'$ appears in some proof of $G$, and let $l$ be some instance of $l'$.*

*Let $\mathcal{I}(\alpha)$, the instance property, be the property that, for a literal $\alpha$, a completion of $\alpha$ exists anywhere $\alpha$ appears below $G$, using only instances of literals from $\mathcal{P}^+(G, \Delta)$.*

*Consider a proof of $G$ in which $l'$ appears. If the property $\mathcal{I}$ is true for all instances of all literals which appear in the proof at the depth of $l'$ or less, then $\mathcal{I}(\neg l)$ is also true.*

**Proof.** By induction on the depth $d$ of literal $l'$ in the proof of $G$.

**(Base case.)** ($d = 0$) In this case, $l'$ is just one of the goal literals $G_i(\overline{x})$. We need to show that $\neg l$, the negation of an instance of $G_i$, has a completion everywhere below $G$ using only (instances of) literals from proofs of $G$.

We know that there exists some binding list $\theta$ such that $l = G_i\theta$. Also, $\neg l$ occurs below some goal literal, say $G_j$. If $i = j$, then $\neg l$ has a completion by a simple reduction with the goal literal ancestor using the binding list $\theta$.

Otherwise, we can use one of the contrapositives of the negated goal, which is

$$\neg G_i \Leftarrow G_1 \wedge \ldots$$

The subgoal $\neg l$ then unifies (via the binding list $\theta$) with the head of this rule, resulting in the conjunctive subgoal $(G_1 \wedge \ldots)\theta$ (which is missing $G_i$). Each of these literals is an instance of some goal literal, and hence in $\mathcal{P}^+(G, \Delta)$. Since the original goal follows from $\Delta$, there is a proof of each of the original goal literals. Each such proof, when specialized by $\theta$, serves as a proof of the corresponding literal in the new conjunctive subgoal. Trivially, all literals in this new completion are instances of literals in some proof of $G$, and hence are in $\mathcal{P}^+(G, \Delta)$.

**(Inductive case.)** We know that for all literals at depth $d - 1$ or less in some proof of $G$, negations of instances of those literals have completions everywhere. We need to show the same for negations of instances of $l'$, which appears at depth $d$.

Since $l'$ appears in a proof of $G$, there must be some database rule which, by extension from a previous ancestor subgoal $a$ results in $l'$ (and possibly some siblings $S$). A contrapositive of the rule is

$$\neg l' \Leftarrow \neg a \wedge S_1 \wedge \ldots$$

Since $l$ is an instance of $l'$, there is some binding $\theta$ such that $l = l'\theta$. Hence the subgoal $\neg l$ unifies with the contrapositive rule via the binding $\theta$, resulting in the conjunction $(\neg a \wedge S_1 \wedge \ldots)\theta$. The subgoal $\neg a\theta$ has a completion using only (instances of) literals from $\mathcal{P}^+(G, \Delta)$ by the induction hypothesis, since the subgoal $a$ appears at depth $d - 1$ and $\neg a\theta$ is the negation of an instance of $a$. If each sibling $S_i\theta$ literal had a completion, then the original subgoal of interest $\neg l$ would as well. But (because the $S_i$ literals all appear at depth $d$ in the proof, the depth of $l'$) this is just the conditional statement of the theorem, and so the proof is done. ∎

This next lemma is the first-order generalization of Lemma 15.

**Lemma 27** (**$L$ has completions**) *Let $\Delta$ be a set of clauses and $G$ be a conjunction of literals such that $\Delta \models G$. Let $l'$ be a literal in $\mathcal{P}(G, \Delta)$. Then for any $l$ which is an instance of $l'$, everywhere $l$ occurs below $G$ it will have a completion involving only instances of proof literals from $\mathcal{P}^+(G, \Delta)$.*

**Proof.** By induction on the depth $d$ of literal $l'$ in the weak ME proof of $G$.

Since $l$ is an instance of $l'$, call the binding list which unifies them $\theta$, such that $l = l'\theta$.

**Base case.** ($d = 0$) The literals at depth 0 are just those of the goal itself, $G = G_1(\overline{x_1}) \wedge \ldots$. So $l' = G_i(\overline{x_i})$. Since $G$ follows from $\Delta$, there is a weak ME proof of each of the literals of $G$ from $\Delta$, and hence a proof of $G_i$. Thus wherever $l$ is encountered below $G$, it will have a completion: namely, that same proof of $G_i$ but specialized by $\theta$. This completion uses only instances of literals from $\mathcal{P}(G, \Delta)$, and thus trivially only instances of literals from $\mathcal{P}^+(G, \Delta)$.

**Inductive case.** Since $l'$ appears (at depth $d$) in a proof of $G$, it has a completion in that proof. We map each operation in that completion to one or more operations in whatever new location $l$ is encountered. Assume that the completion of $l'$ in the proof is some sequence of chains $C_1, \ldots, C_k$. The completion of $l$ will be a possibly longer sequence of chains $D_1, \ldots, D_l$ such that

1. For every chain $C_i$, there is some related chain $D_j$ such that if one removes all literals in the chain older than $l$ from $C_i$ (call this $\mathcal{R}(C_i)$), and similarly removes all literals in $D_j$ older than $l'$ (*i.e.* $\mathcal{R}(D_j)$), then the remaining chains will unify with a binding of $\theta$, *i.e.* $\mathcal{R}(C_i)\theta = \mathcal{R}(D_j)$.

2. The ordering on the chains is preserved, so that if $C_i$ is related (in the sense above) to $D_m$ and $C_j$ is related to $D_n$ and $i < j$, then $m < n$.

The sequence $D_1, \ldots$ we construct will also be a valid weak ME deduction. Since the sequence of $C$ chains is a completion of $l'$, the last chain $C_k$ must only contain literals older than $l'$. This means that $\mathcal{R}(C_k)$ is the empty chain, and thus so also must be $\mathcal{R}(D_l)$. This means that the sequence of chains $D$ is a completion for literal $l$ in the new context.

Contraction operations are immediately valid. If the operation between $C_i$ and $C_{i+1}$ is an extension operation, then there must have been some database rule allowing the extension, and that same rule specialized by $\theta$ is valid in the context for $l$. Similarly, if there is a reduction operation between some ancestor $a'$ which is more recent than $l'$, and a top literal $\neg a'$, then a specialized reduction between $a$ $(= a'\theta)$ and $\neg a$ $(= \neg a'\theta)$ is similarly valid.

The remaining operation to map is a reduction operation in the original proof, which is to an ancestor at depth $d - 1$ or less. The original proof includes a chain

$$\neg a(\overline{y}) \;\ldots [l'] \;\ldots [a(\overline{x})] \;\ldots [G]$$

In the new context, the top literal is $\neg a(\overline{y})\theta$, *i.e.* $\neg a$ specialized by $\theta$. This literal has a completion, using only literals from $\mathcal{P}^+(G, \Delta)$, because

1. $\neg a\theta$ is a negated instance of a literal (namely, $a(\overline{x})$) which appears in a proof of $G$

2. By the induction hypothesis, the ancestor $a(\overline{x})$ itself (and all other literals at that depth or less) has completions using only literals from $\mathcal{P}^+(G, \Delta)$

3. By Lemma 26, the top literal in the new context, $\neg a\theta$, does as well

Hence we have constructed a completion for literal $l$, given that $l'$ appears in some proof of $G$ at depth $d$, and the inductive case is complete. This finishes the proof. ∎

**Lemma 28 ($\neg L$ has completions)** *Let $\Delta$ be a set of clauses and $G$ be a conjunction of literals such that $\Delta \models G$. Let $l'$ be a literal in $\mathcal{P}(G, \Delta)$. Then for any $l$ which is an instance of $l'$, everywhere $\neg l$ occurs below $G$ it will have a completion involving only instances of proof literals from $\mathcal{P}^+(G, \Delta)$.*

**Proof.** By induction on the depth $d$ of $l'$ in the proof of $G$. Since $l$ is an instance of $l'$, $l = l'\theta$ for some binding list $\theta$.

**Base case.** ($d = 0$) $l'$ must be one of the goal literals $G_i$. Thus $\neg l = \neg G_i\theta$. Since $\neg l$ occurs below $G$, it must occur below some particular goal literal $G_j$. If $i = j$, then $\neg l$ has a completion via a simple reduction with that goal literal, since $G_i\theta$ and $G_i$ always unify. Otherwise, a contrapositive of the negated goal

$$\neg G_i \Leftarrow G_1 \ldots \wedge$$

permits an extension operation from the $\neg l$ subgoal via the binding $\theta$. This results in the conjunctive subgoal $(G_1 \wedge \ldots)\theta$, where $G_i$ is missing. Since the goal $G$ is provable, each literal $G_j$ has a completion (using only literals from $\mathcal{P}^+(G, \Delta)$). Since each literal in our new conjunctive subgoal is an instance of some goal literal, each of these conjuncts will also have a completion (using only instances of literals from $\mathcal{P}^+(G, \Delta)$).

**Inductive case.** For every literal $a'$ at depth less than $d$, each negated instance $\neg a$ has a completion everywhere below $G$. We need to show this is also true for literal $\neg l$.

Since $l'$ appears in some proof of $G$, there must be a database rule

$$a_n \Leftarrow l' \wedge s_1 \wedge \ldots$$

A contrapositive of this rule is

$$\neg l' \Leftarrow \neg a_n \wedge s_1 \wedge \ldots$$

An extension operation on the $\neg l$ subgoal, using that rule and the binding $\theta$, results in the conjunction $(\neg a_n \wedge s_1 \wedge \ldots)\theta$. The literal $\neg a_n\theta$ has a completion here by the induction hypothesis, as it is an instance of a literal (namely, $a_n$) which appears at a depth less than $d$ is the proof of $G$. Each of the sibling $s_i\theta$ literals has a completion by Lemma 27, as they are instances of $s_i$ literals which appear in a proof of $G$.

Thus $\neg l$ has a completion in every context below $G$, using only literals from $\mathcal{P}^+(G, \Delta)$. ∎

Before moving on to the main result, we need to introduce one more concept. The notion of a "completion" needs to be generalized somewhat. When attempting to prove a subgoal such as `P(x)`, it's often the case that the search instead discovers proofs of instances of `P(x)`, say that the subgoal is true for `x` bound to `A`. If the subspace below `P(x)` is fully explored, this tells us that `P(A)` has a completion, and that no other instance of `P(x)` does.

If later the subgoal `P(y)` is encountered, we would like to be able to conclude that `y` bound to `A` is the only solution. The next results justify this conclusion.

**Definition 29 (Solution)** *Let $P$ be a subgoal encountered during a proof effort. A binding list $\theta$ is a <u>solution</u> of $P$ if there is a completion of $P\theta$ from the same context.*

To add simple failure caching to model elimination in a first-order situation, Definition 17 must be interpreted in a more generalized context. A literal fails (and is thus added to the failure cache) if it has no solutions, not merely if it has no completions.

**Lemma 30** *Let $\Delta$ be a set of clauses and $G$ be a conjunction of literals such that $\Delta \models G$. Let $f'$ be a literal appearing in the failure cache during an* SFC ME *search for a proof of $G$ from $\Delta$. Then for any $f$ which is an instance of $f'$, it is the case that $f \notin \mathcal{P}^+(G, \Delta)$,* i.e. *neither $f$ nor $\neg f$ appears in any weak* ME *proof of $G$ from $\Delta$.*

**Proof.** By induction on the number of literals, $n$, in the failure cache of the SFC ME algorithm. Since $f$ is an instance of $f'$, there is some binding list $\theta$ such that $f = f'\theta$.

**Base case.** ($n = 0$) With an empty failure cache, it is trivially the case that none of them appear in any proof of $G$.

**Inductive case.** None of the $n - 1$ literals (nor any instance) in the failure cache appears in any proof of $G$. Then literal $f'$ is encountered, and fails to have a completion. We need to show that neither $f'$ nor $\neg f$ appears in any proof of $G$.

**Part 1.** ($f$ not in proof) If $f$ appeared in some proof of $G$, then by Lemma 27 it would have a weak ME completion anywhere below $G$ using only (instances of) literals from $\mathcal{P}^+(G, \Delta)$. By the induction hypothesis, no instance of a literal from $\mathcal{P}^+(G, \Delta)$ appears in the failure cache. Thus there is a completion of $f$ in the place where $f'$ failed. But since $f = f'\theta$ has a completion, this means that $f'$ must have a solution (namely, the solution $\theta$ or perhaps one more general) in this same context. Since $f'$ did not have a solution at this point, $f$ cannot appear in any proof of $G$.

**Part 2.** ($\neg f$ not in proof) The same argument as for Part 1, but using Lemma 28 instead of Lemma 27, shows that $\neg f$ cannot occur in any proof of $G$.

Thus no instance of any literal in a proof of $G$ (or the negation of such an instance) can occur in the failure cache of the SFC ME algorithm. ∎

**Theorem 31 (Completeness of failure caching)** *Let $\Delta$ be a set of clauses and $G$ be a conjunction of literals such that $\Delta \models G$. Then every complete search of the* SFC ME *space will discover some* SFC ME *proof of $G$ from $\Delta$.*

**Proof.** Since weak ME is complete for first-order inference, we know that there exists a weak ME proof of $G$ from $\Delta$. Since SFC ME is a pruning strategy, we need only show that the known weak ME proof of $G$ cannot possibly be pruned. This is the case because a subspace of the proof is pruned only if the top literal is found in the failure cache, and by Lemma 30 none of the literals in the failure cache appear in any weak ME proof of $G$. Thus every proof of $G$ will still be in the SFC ME space. ∎

## 4.5 Flushing Between Queries

As a practical matter, theorem prover caching for Horn clause databases typically is implemented taking advantage of a variety of other properties of the caching, not all of which hold in the first-order case. For example, it is often the situation that a large number of queries are asked of a static database. In that case, a bounded-sized cache could remember the last $n$ "interesting" solutions to various subgoals, and keep this information across queries.

In the non-Horn case, conclusions about intermediate subgoals are unfortunately dependent upon the specific query, and thus you must flush caches between proof attempts, preventing the reuse of cached results from one proof effort in a subsequent problem. The following is an illustration of the incompleteness that would otherwise arise without flushing the failure cache:

From the database in table 4.4 the sequence of goals `G`, `¬G`, and `C`, would be solved incorrectly by a caching prover that does not flush the failure cache, as it would not find a proof for any of the three goals, despite the fact that the third follows from the database.

| | | |
|---|---|---|
| `C` | $\Leftarrow$ | `G` |
| `C` | $\Leftarrow$ | `¬G` |

Table 4.4: Must flush the failure cache

The goal `G` resolves with (the contrapositive of) the second rule, but no further resolutions are possible, as shown in figure 4.11. Thus `G` and `¬C` would be put in the failure cache. For the second query, the goal `¬G` resolves with the first rule, resulting in the space shown in figure 4.12. `¬G` and `¬C` (which is already present) would be added to the failure cache.

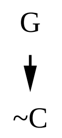


Figure 4.11: Not flushing: `G`

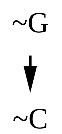


Figure 4.12: Not flushing: `¬G`

When finally attempting to prove `C`, resolution with the first rule results in the subgoal `G`, and with the second in the subgoal `¬G`, as shown in figure 4.13. Unfortunately, both are already in the failure cache, so all remaining subtrees are pruned, and the prover returns unsuccessfully. This is despite the fact that there is a reduction proof of the goal `C` from the database, as shown in figure 4.14.

The success cache also must be flushed between queries. Since non-Horn inference requires adding the negated goal to the database, all proofs become context sensitive. Consider attempting to prove a tautology, *e.g.* `P or ¬P` from the empty database. The proof succeeds (as it should) as follows:

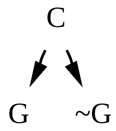


Figure 4.13: Not flushing error: `C`

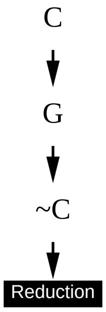


Figure 4.14: Reduction proof of `C`

1. The negated goal, namely `¬P and P` is added to the database, which results in a temporarily-augmented database with two sentences: `P` and `¬P`.

2. The disjunctive goal is split, and each disjunct is attempted separately. If any disjunct succeeds, the overall goal also succeeds.

3. The first disjunct, `P`, succeeds immediately, by looking up the database sentence `P`.

Obviously, the sentence `P` is not, in general a consequence of an arbitrary database. During the course of the proof of the tautology, `P` can indeed be added to the success cache. If the cache is not flushed between queries, however, then `P` might erroneously be concluded during the course of some other proof.

With some extra effort, it is possible to retain only those conclusions that actually follow from the database. Much like the suggestion in section 2.5.2 for dealing with identifying failure cache entries in the presence of identical ancestor pruning, propagating three-valued answers throughout the tree would allow an algorithm to distinguish whether a conclusion followed from the database or was valid only in the context of a particular query. In this case, the third value would indicate whether specially marked sentences (those derived from the negated goal) were used to produce a particular answer. Note that complex answers must be propagated in any case, as solutions using reductions occur throughout first-order model elimination proofs, and such solutions indicate nothing about the truth of intermediate subgoals.

## 4.6 Depth Bounds

Depth-bounded search is a common way to control inference, especially via a complete mechanism like depth-first iterative deepening [Kor85]. Using a depth bound complicates the process of adding a cache to an inference algorithm. With bounded search, a subgoal may fail (or may have fewer answers than it otherwise would) because part of the space below exceeded the bound, rather than because there was no possible proof of the subgoal from the database.

Thus subgoal caches typically must be augmented with a field indicating the depth to which the subgoal had been searched. If `Fruit(x)` is explored with a depth limit of 5, and the answers `x`→`Apple` and `x`→`Banana` are found, this can be stored in a cache with the notation that it is valid information for searches up to a limit of 5. If later the subgoal `Fruit(y)` is encountered, but this time with a remaining depth limit of 3, the answers from the cache can replace what would otherwise be further search.[8] If instead, however, the new `Fruit(y)` subgoal has a remaining depth of 9, then the cache entry is not valid and normal inference must continue. (This is assuming, as is usually the case, that we wish to prevent possible adverse search effects. The alternative is to start with the answers from the cache, and then proceed to searching the space below up to the needed depth limit if the cached answers aren't sufficient in this context.)

How does this relate to caching in non-Horn theorem proving? Since we now have access to the reduction operation, solutions to a subgoal can be context-dependent. This means that just because no solution is found to `Fruit(x)` up to depth 3 in one context, does not mean that there will similarly be no solutions to depth 3 in some other context. The results of section 4.4 seem to indicate that solutions will be independent of context. A careful reading of those results, however, shows the claim is only that the *complete* space below a subgoal in one context yields total information about the complete space below the same subgoal in any other context. It is specifically not the case that depth-bounded explorations will yield the same set of answers.

A simple example is shown in table 4.5. A search for a proof of the goal `G` with a limit of 3 and with failure caching enabled is shown in figure 4.15. Notice that the first occurrence of the subgoal `C` fails with a remaining depth of 1; the second occurrence is pruned erroneously in this case, using the incorrect justification that `C` is in the failure cache with a depth of 1. This is an error because, as shown in figure 4.16, there actually is a proof of `G` down this second branch, and the proof is even within the depth limit.

If the second `C` succeeded, then why did the first fail? What happened to the alternate proof predicted by the results of section 4.4? It exists, as shown in figure 4.17, but at a depth deeper than the cutoff currently in place.

This particular example shows only that the alternate proof guaranteed by section 4.4 is below the depth cutoff. In this case, however, there actually is another proof of the goal even within the original depth cutoff, and even using this form of failure caching. The complete space to depth 3 is shown in figure 4.18; even with the unjustified pruning of the second `C` subgoal, a proof of the goal is found.

There is not always a different proof within the current depth bound. By examining the construction of section 4.4, it appears that if a subgoal fails with a particular depth bound, then other

[8]Allowing cache hits to be successful when the remaining depth is *less* than that in the cache can result in more solutions than would otherwise have been found. This generally is not considered to be a problem, and is in fact a potential advantage of using caching.

| | | |
|---|---|---|
| G | $\Leftarrow$ | A |
| A | $\Leftarrow$ | C |
| ¬R | $\Leftarrow$ | C |
| C | $\Leftarrow$ | R |
| G | $\Leftarrow$ | ¬R |

Table 4.5: Depth limits and non-Horn inference

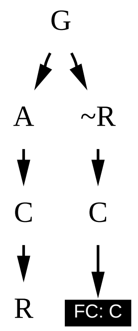


Figure 4.15: Incorrect depth-limited failure caching

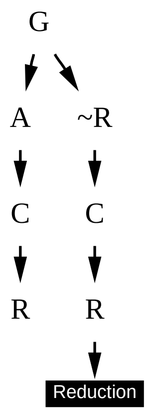


Figure 4.16: Context-dependent depth-limited inference

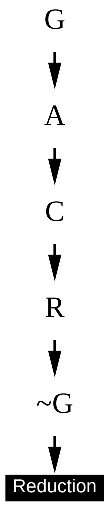


Figure 4.17: An alternate proof below the depth cutoff

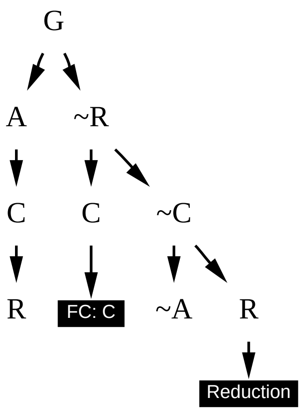


Figure 4.18: A different proof within the depth cutoff

occurrences of that subgoal with a bound around half the original bound should also fail. At this time, however, tight limits on transferring bounded failures are not known.

Without care in using bounded failures, incompleteness can result. Consider the example in table 4.6 for the goal

```
¬P(x) and D(x)
```

which was discovered by David Sturgill. If it is explored with this same algorithm (model elimination, iterative deepening starting at a depth of 1 and incrementing by 1 on each step, and failure caching), no solution will ever be found despite the fact that there is a proof at depth 6.

(Recall that the negated goal must also be added to the database, so the rule

```
P(x) ⇐ D(x)
```

is also present in the database.)

| | | |
|---|---|---|
| ¬P(x) | ⇐ | ¬Q(x) |
| ¬P(x) | ⇐ | P(S(x)) |
| Q(x) | ⇐ | ¬Q(S(x)) |
| ¬Q(x) | ⇐ | Q(S(x)) |
| D(x) | ⇐ | Q(S(x)) |

Table 4.6: Sturgill anomaly

Sturgill explains:[9]

> Let's assume that we fail to prove the first conjunct in our goal, ¬P(x), on some iteration, $D$. As a result of this failure and its failed subgoals, we will insert (among other things), the following into the failure cache:
>
> ¬P(x) failed with depth cutoff $D$
> ¬Q(x) failed with depth cutoff $D-1$
> P(x) failed with depth cutoff $D-1$
> Q(x) failed with depth cutoff $D-2$
>
> When we then proceed to try to find a proof at depth $D+1$, every proof of ¬P(x) fails because the cached failures from the previous iteration block every solution; every proof of ¬P(x) depends on a proof of of an instance of ¬P(x), ¬Q(x), P(x) or Q(x) with less marginal search depth remaining than the previously cached failures.

The expanded proof space to depth 3 is shown in figure 4.19. A proof of the query without failure caching is shown in figure 4.20.

[9]Private communication.

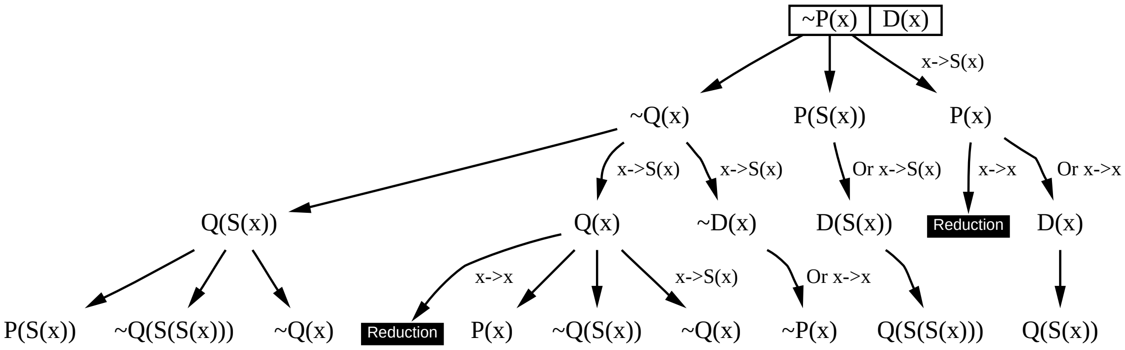


Figure 4.19: Sturgill's iterative-deepening anomaly

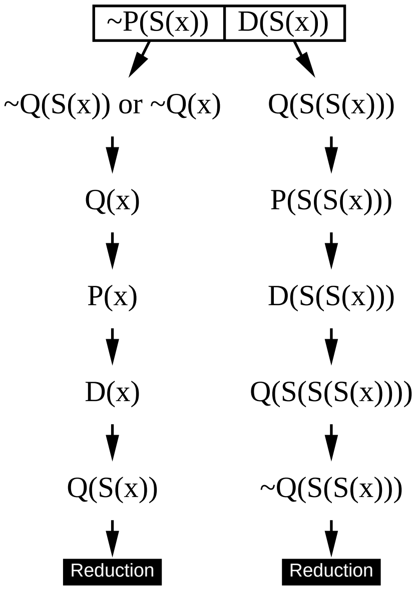


Figure 4.20: The correct proof in Sturgill's anomaly

## 4.7 Related Work

### 4.7.1 Positive Refinement

Plaisted's positive refinement [Pla90] of model elimination established that only positive subgoals need to be checked for reductions with their ancestors, and the algorithm would remain complete. This refinement is especially attractive for near-Horn problems, because ancestor lists then contain few negative subgoals, and the reduction check is computationally cheap. In fact, for Horn theories, no reductions take place at all.

While the positive refinement makes cache hits more likely, it appears that in many cases they are still not likely enough, and the net result is that the benefit from caching is never realized. Stickel [Sti94] writes:

> A refinement [Pla90] of the model elimination procedure that uses negative but not positive ancestor goals may make looking up solutions in the cache succeed more frequently, but probably still not often enough.

### 4.7.2 Problem Reduction Format

Plaisted [Pla88] also suggested an approach completely different from model elimination, the modified problem reduction format. This algorithm is complete for non-Horn theories, without using contrapositives or reductions.

Since there are no context-sensitive reduction operations, caching is easy and much like the Horn case. In his experimental analysis, Plaisted reported the utility of caching in this non-Horn setting to be much like caching in any other context:

> Caching seems to help, on the average, although some problems are much faster without caching.

But he notes

> Another disadvantage of caching is that the storage required can be prohibitive on large problems, although this is not often the case.

Experiments were run on a Sun 3, and compared a caching theorem prover to the same prover with caching turned off. On the average, the caching prover performed 2.2 inferences per second, took 346.31 cpu seconds per problem, and required 760.5 inferences per problem. The non-caching version performed 15.0 inferences per second, took 1125.01 cpu seconds per problem, and required 16840.5 inferences per problem.

Non-Horn sentences are dealt with by explicit case analysis. Unfortunately, this means that it is most efficient for Horn theories, and the efficiency degrades as the set of clauses becomes more and more non-Horn.

In the simple problem reduction format, the required splitting rule for case analysis forces the search to be essentially forward from the premises to all conclusions, rather than backward from the goal in the way that model elimination or the set of support restriction for resolution allow. For arbitrary propositions $\phi$ and $\psi$, the splitting rule essentially says

$$[\ (\ \phi \Rightarrow \psi\ )\ \texttt{and}\ (\ \neg\phi \Rightarrow \psi\ )\ ]\ \texttt{implies}\ \psi$$

In the backward direction, this axiom schema applies to every subgoal $\psi$ encountered, and suggests trying case analysis on every unrelated subgoal $\phi$ in the database.

For example, in the database of table 4.7[10] with goal P, the natural backward chaining is from P to Q to R and then to ¬N. This proof cannot be completed, however, since the subgoal ¬N does not follow from the database and thus cannot be solved.

| | | |
|---|---|---|
| P | $\Leftarrow$ | Q |
| Q | $\Leftarrow$ | R |
| R | $\Leftarrow$ | ¬N |
| P | $\Leftarrow$ | N |

Table 4.7: Must split early

Instead, the splitting axiom must be applied as the first rule to backchain on from the goal P, resulting in the two subgoals ¬N $\Rightarrow$ P and N $\Rightarrow$ P. In the problem reduction format, the first subgoal eventually reduces to the tautology ¬N $\Rightarrow$ ¬N, and the second to the tautology N $\Rightarrow$ N. Unfortunately, there is no simple way to know that the subgoal ¬N will be generated later on, making it necessary to do case analysis on N directly from the goal P.

In the modified problem reduction format, whenever a negative subgoal is encountered it is assumed to succeed and is added to the list of assumptions in the problem format. Thus the solution to a goal may have additional assumptions attached. At that point, either the case analysis split is performed (and a new attempt to prove the positive version of the encountered negative subgoal), or else the assumption of the negative subgoal's truth is simply passed up to the parent subgoal.

As before, though, the opportunities for case analysis explode as the theory becomes more and more non-Horn. While this modified format may be useful for near-Horn theories, it is not feasible for strongly non-Horn theories. Ideally we would like to implement effective caching in a non-Horn goal-directed theorem prover like weak ME, without sacrificing completeness.

Baumgartner and Furbach [BF94] propose a similar scheme called restart model elimination. Restart model elimination is even more restricted than the simple problem reduction format, while still remaining complete. The modified problem reduction format essentially allows restarts with any goal along the current path, whereas restart model elimination only allows restarts with the original goal literal. In addition, restart model elimination includes the negative literals along paths, which Baumgartner and Furbach have found to be valuable information during experiments.

### 4.7.3 Foothold Format

Backward-chaining proof spaces have a large amount of duplication in them: proofs of a given solution occur multiple times in the space. Spencer [Spe90] suggests an inference procedure which avoids duplicate proofs during its search. (The following description has been adapted from Spencer's description.)

[10] This example is from Plaisted [Pla88].

The basic idea behind the foothold format is to avoid redundant proofs that arise when reasoning by cases by breaking up the symmetry of the cases. If the database contains a disjunction `A or B`, then implicitly there are two contrapositive rules: $\mathtt{A} \Leftarrow \neg\mathtt{B}$ and $\mathtt{B} \Leftarrow \neg\mathtt{A}$. In a foothold proof, labels from the set $-1, 0, +1$ are assigned to the literals on the right hand side of the contrapositive rules. In this example the symmetry is broken by assigning different labels to $\neg\mathtt{B}$ and $\neg\mathtt{A}$.

All positive literals (and separately, all negative literals) are placed in an ordered sequence, *e.g.* alphabetical. If a rule has a positive head literal, then a positive body literal is assigned the label $+1$ if it appears before the head literal in the ordered sequence, and $-1$ if it appears after. Negative literals are assigned the label 0. (The complementary labelling holds for rules with a negative literal head.) For example, the database in table 4.8 is rewritten into the labelled rules in table 4.9.

| | | |
|---|---|---|
| P | $\vee$ | $\neg$A |
| P | $\vee$ | $\neg$B |
| A | $\vee$ | B |

Table 4.8: A clausal database

| | | |
|---|---|---|
| P | $\Leftarrow$ | $\mathtt{A}^{0}$ |
| $\neg$A | $\Leftarrow$ | $\neg\mathtt{P}^{0}$ |
| P | $\Leftarrow$ | $\mathtt{B}^{0}$ |
| $\neg$B | $\Leftarrow$ | $\neg\mathtt{P}^{0}$ |
| A | $\Leftarrow$ | $\neg\mathtt{B}^{+1}$ |
| B | $\Leftarrow$ | $\neg\mathtt{A}^{-1}$ |

Table 4.9: Labelled rules

Proof spaces are built using the labelled contrapositive rules. During the search for a negated ancestor (in order to perform a reduction inference operation), the labels of all the literals encountered are summed up. If the sum is positive when the negated ancestor is reached, then the proof is accepted. Otherwise it is rejected.

In a model elimination prover, there are two reduction proofs for the goal `P` from the database in table 4.9, namely

$$\mathtt{P} \leftarrow \mathtt{A}^{0} \leftarrow \neg\mathtt{B}^{+1} \leftarrow \neg\mathtt{P}^{0}$$

and

$$\mathtt{P} \leftarrow \mathtt{B}^{0} \leftarrow \neg\mathtt{A}^{-1} \leftarrow \neg\mathtt{P}^{0}$$

Using the restriction of the foothold format, only the first of these is acceptable; the second can be pruned.

# Chapter 5

# Horn-Clause Postponement Caching

Postponement caching is the idea of exploring the inference space as a graph rather than as a tree. Thus subgoals that are repeated in the space (and the whole subtrees below them) are only explored in a single place during a proof effort.

When a subgoal is encountered for the first time, a new literal appears in the inference space. As much of that subspace below the subgoal as needed to find the first solution is explored, as usual. In addition, however, a continuation is kept in case more answers are ever required, whether from the original parent or from some other parent (in what has now become a search graph). The technique stores an intermediate amount of information, compared to the complete or partial table caches. Each cache entry contains the set of answers found so far, along with the continuation in case more answers are needed. Identical proof spaces are never repeated. Instead, if a subsequent slaved subgoal needs an additional answer and the existing cache only has fewer stored, then the original subproof is resumed by restarting the continuation attached to the cached subgoal. All subsequent answers discovered are copied to each existing slave subgoal.

Postponement of subgoals along the ancestor path (a subset of full proof space caching) controls recursion by effectively forcing exploration of base cases before chasing down recursive branches of the space. It can serve much of the purpose of iterative deepening search, which is typically the response to potential recursion in the domain theory. It is a superior method for controlling the most common kind of recursion, that involving syntactically similar subgoals. (Dealing with other kinds of infinite recursion, such as that arising from repeated function application to terms, requires other techniques.)

## 5.1 Example

Consider the database in table 5.1.[1] The goal is to find all (three) things that lions can outrun.

The hard part for automated inference is the first rule, which states that the `outrun` relation is transitive. For a traditional depth-first search algorithm, such a rule results in an infinite search space, so a query asking for all answers will not terminate. Postponement caching does recursion control by slaving new subgoals to identical parents.

The initial goal

[1]This example is from Ginsberg [Gin93b, exercise 8.14].

```
Outrun(fast,slow)                ⇐
     Outrun(fast,medium) and Outrun(medium,slow)
Outrun(predator,prey)            ⇐  Eat(predator,prey)
Eat(carnivore,Food(carnivore))   ⇐  Carnivore(carnivore)
Eat(Lion,Zebra)
Outrun(Zebra,Dog)
Carnivore(Dog)
```

Table 5.1: Carnivore database

`Outrun(Lion,food)`

resolves with the first two rules in the database. Assuming we explore the transitive rule first, we now have a conjunctive subgoal:

`Outrun(Lion,medium) and Outrun(medium,food)`

To solve the conjunction, work begins with the first conjunct. At this point a standard backward chainer would resolve the subgoal `Outrun(Lion,medium)` with the same two database sentences above, and the process would loop indefinitely. Postponement caching, on the other hand, notices that this new subgoal is the same (up to variable renaming) as the original goal. Instead of continuing inference with the subgoal, it attaches the subgoal to the top level goal, such that any subsequent answers to the top level goal will be propagated to this subgoal.

The search then continues on the fringe of the proof space, that now contains only the resolution of the original goal with the second rule. This leads to the subgoal `Eat(Lion,food)`, that in turn resolves with the first ground fact to yield the answer `food`→`Zebra`. This answer propagates to the root of the proof space, yielding the first answer to the original query

`Outrun(Lion,Zebra)`

as shown in figure 5.1.

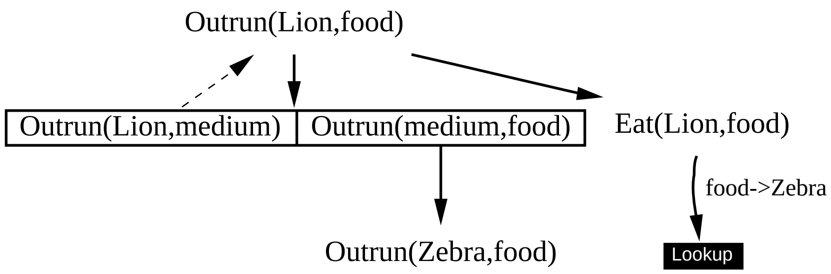


Figure 5.1: Lions outrun Zebras

Since the top goal discovered a new answer, and it is master to a slave, the new answer must be propagated to the slave. The answer is first transformed by the same unifier that makes the top goal and the subgoal identical, so the new answer `medium`→`Zebra` is added to the subgoal `Outrun(Lion,medium)`. With this binding, we can return to the second conjunct of the conjunction, that (after plugging in the bindings so far) becomes `Outrun(Zebra,food)`. This has three resolutions with the database, and the resolution with the second ground fact results in an immediate answer: `food`→`Dog`. This second answer propagates up to the top of the tree, yielding the second answer to the original query

```
Outrun(Lion,Dog)
```

as shown in figure 5.2.

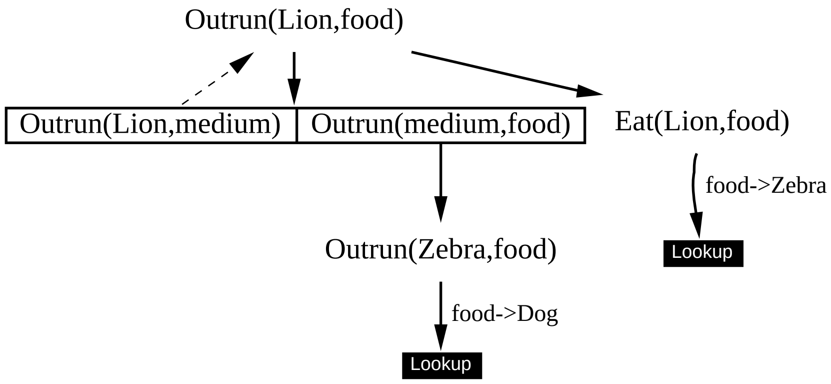


Figure 5.2: Lions outrun Dogs

The proof space showing the third answer

```
Outrun(Lion,Food(Dog))
```

is shown in figure 5.3. If the search for more answers is continued, the algorithm soon stops, reporting no more answers are available. (The final space is shown in figure 5.4.) For typical inference engines, this problem has an infinite search space. Postponement caching, however, modifies the space sufficiently that it becomes finite (and small!), and so it can be searched completely in a short amount of time. This ability, to report that there are no more possible answers when exploring a recursive space, is a hallmark of postponement caching.

## 5.2 Cycles

It is possible for a dependency cycle to arise, where two goals are mutually dependent. This is similar to the potential problem mentioned for goal displacement in section 2.5.5. There, in order to preserve soundness, the solution was to only allow displacement if the ancestor sibling had not yet been expanded.

Such a constraint is not necessary for postponement caching. In goal displacement, the halted subgoal is treated as having a successful completion. In postponement caching, no label is placed on

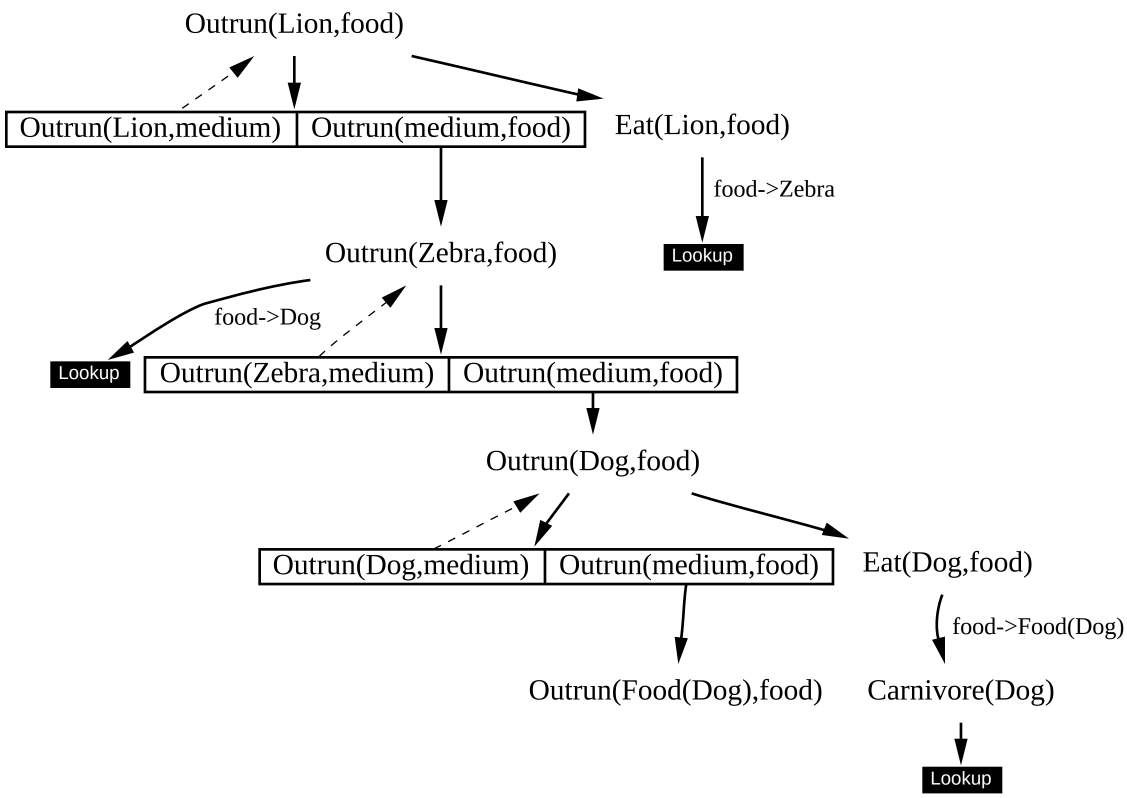


Figure 5.3: Lions outrun Dog Food

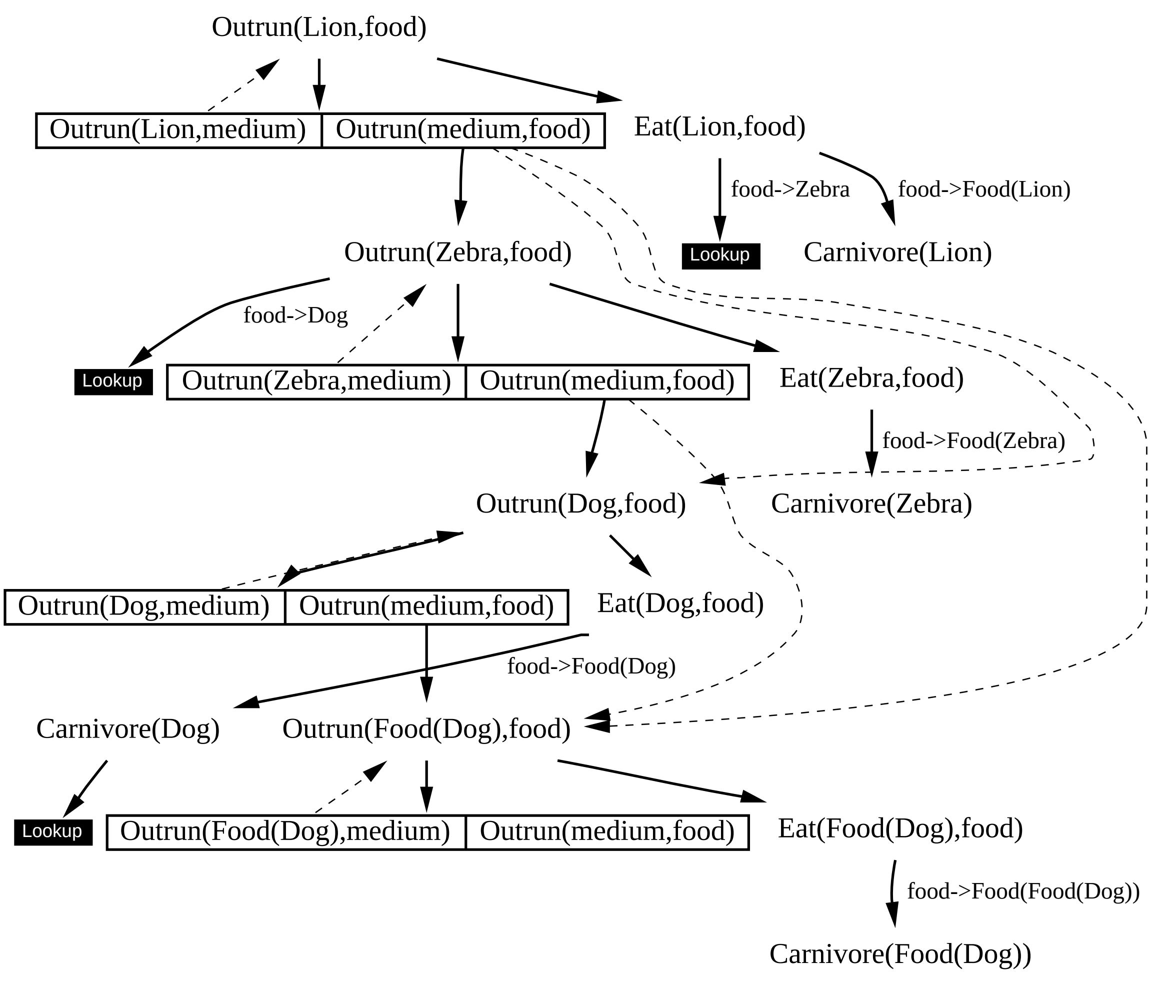


Figure 5.4: Lions only outrun three things

the slaved subgoal until the master subgoal is conclusively labelled. Thus in a mutually dependent goal cycle, both slaved subgoals are waiting for the ancestors to complete, which (if the subgoals actually don't follow from the theory) will never occur. The search will halt, with a finite graph and no proof discovered, exactly as it should.

There remains, however, a more complex situation that must be dealt with specially. Postponement caching interacts with conjunction solving. In most theorem proving systems, conjunctions are typically explored with a depth-first search.[2] A portion of the conjunction space may halt with a goal cycle, but there still may be a solution to the conjunction elsewhere in the conjunction space.

This means that the mechanism which searches the conjunction space must be capable of forking, allowing a subtree of the space to continue (in case a distant master subgoal finally finds a further answer), and also being able to continue the search of the conjunction space under the assumption that an answer will never return.

Consider the database in table 5.2. The space in figure 5.5 shows the error that can result with no special mechanism. The correct space (with a proof) is shown in figure 5.6.

| | | |
|---|---|---|
| `G` | $\Leftarrow$ | `P(x) and Q(x)` |
| `P(1)` | | |
| `P(2)` | | |
| `Q(2)` | | |
| `Q(1)` | $\Leftarrow$ | `R and S` |
| `R` | $\Leftarrow$ | `T and S` |
| `T` | $\Leftarrow$ | `Q(1)` |
| `S` | | |

Table 5.2: Postponement blocks

## 5.3 Formal Results

Postponement caching for Horn theories is an augmentation of the Horn clause version of Loveland's MESON procedure[3] [Lov78]. (The Horn version of MESON merely eliminates the reduction operation.)

**Definition 32 (Postponement caching)** *Begin with the standard* MESON *procedure. In addition, maintain a separate cache table. Each entry in the cache has four elements:*

1. *the cached literal*
2. *a set of answers (*i.e. *bindings) found so far*
3. *a continuation data structure in case more answers are required*

[2] See section 2.5.4 for an alternative search strategy.

[3] The MESON procedure is isomorphic to weak model elimination.

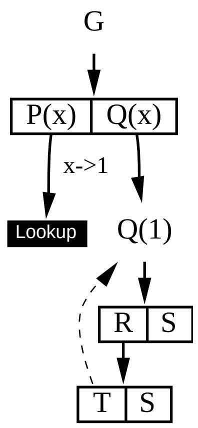


Figure 5.5: An apparently complete space

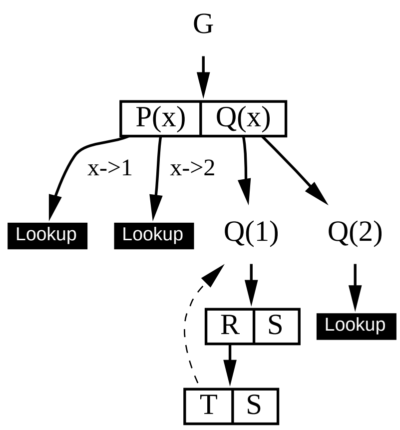


Figure 5.6: The actual complete space

*4. a set of slaved subgoals to which answers should be propagated*

*When expanding a subgoal for the first time:*

1. *If the subgoal literal is not found in the cache, then add a new entry to the cache with this literal, with an empty set of answers, with the children of the subgoal being the continuation, and with an empty set of slaved subgoals*

2. *Whenever a new solution is found to this subgoal, that answer is propagated to the parent, and also added to the set of answers in the cache entry.*

*When searching for a subsequent answer for a slaved subgoal, select the next unused answer in the cache entry. If there are no more unused answers in the entry, and the continuation is non-empty, then reactivate the master subgoal for it to search for a subsequent answer. If the continuation is empty, then fail.*

*To search for a subsequent answer from a master subgoal, retrieve the next item from the continuation and expand it. If it returns an answer, then propagate that solution to this subgoal's parent, as well as to every slave subgoal in the cache entry. If this child has no more answers, then remove it from the continuation list and proceed to the next item in the continuation. If there are no more items in the continuation, then propagate a "termination" notification to the parent and to each slaved subgoal.*

**Theorem 33** *Postponement caching is sound.*

**Theorem 34** *Postponement caching is complete for Horn theories.*

**Proof.** Although it was developed independently, postponement caching turns out to perform essentially the same computations as the OLDT augmentation of logic programming described in section 5.4.2. In the case of Horn clause inference, then, we can simply defer to the work referenced there in order to establish the soundness and completeness properties. ∎

## 5.4 Related Work

### 5.4.1 Recursion Control

In this section we describe an algorithm that was formally proposed by Smith [SGG86], having earlier been discovered independently by both Black [Bla68] and McKay and Shapiro [MS81].

Consider the database in table 5.3 that has two ground facts about paths between three cities, and one rule stating that the `Path` relation is transitive. The goal is to find all paths from city `A`.

For a traditional depth-first search algorithm, the rule results in an infinite search space, as illustrated in figure 5.7. Even for a slightly more sophisticated search strategy like iterative-deepening, a query asking for all answers will not terminate. (Note also that the need to look for multiple answers arises naturally in the course of solving subgoals to a top-level query, so even if only a single answer is asked for, controlling recursion is still a crucial problem.) Recursion control deals with the situation by slaving new subgoals to identical ancestors.

The proof space for finding the first answer is shown in figure 5.8. The initial goal

```
Path(start-city,end-city)  ⇐
     Path(start-city,city) and Path(city,end-city)
Path(A,B)
Path(B,C)
```

Table 5.3: Paths between cities

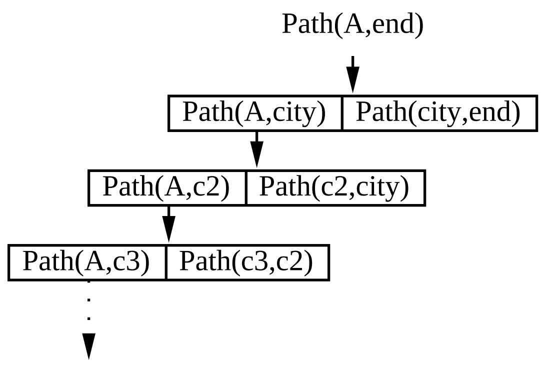


Figure 5.7: An infinite search space: `Path(A,end)`

```
Path(A,end)
```

resolves with both the rule and the first ground fact in the database. Assuming we explore the transitive rule first, we now have a conjunctive subgoal:

```
Path(A,city) and Path(city,end)
```

To solve the conjunction, we work on the first conjunct, namely `Path(A,city)`. At this point a naïve backward chainer would resolve the subgoal with the transitive rule again, and the process would loop indefinitely.

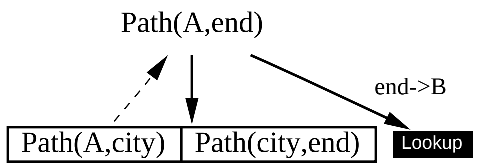


Figure 5.8: Proof of the first `Path` solution

Recursion control notices that this new subgoal is the same (up to variable renaming) as the original goal. Thus instead of continuing inference with the subgoal, it attaches the subgoal to the top level goal, such that any subsequent answers to the top level goal will be propagated to this subgoal.

The search then continues on the fringe of the proof space, that now contains only the resolution of the original goal with the first ground fact. This resolves successfully yielding the answer `end`→`B`.

If we were searching for more than one (or all) answers, the space would expand immediately. Since the top goal discovered a new answer, and it is master to a slave, the new answer must be propagated to the slave. The answer is first transformed by the same binding list that makes the top goal and the subgoal identical, so the new answer `city`→`B` is added to the subgoal `Path(A,city)`. With this binding, we can return to the second conjunct of the conjunction, that (after plugging in the bindings so far) becomes `Path(B,end)`. This similarly has two possible resolutions with the database. The child from the transitive rule is again postponed, and then the second ground fact is used to derive the answer `end`→`C` to this new subgoal `Path(B,end)`. This second answer propagates up to the top of the tree, yielding the second answer to the original query: `Path(A,C)`. This space is shown in figure 5.9.

The complete proof space for finding all answers is shown in figure 5.10. Note that there is no longer any active fringe to explore; not only have all the answers been found, but it is also known that there are no more answers derivable from the theory.

The postponement caching scheme from chapter 5 is a direct extension of Smith's Algorithm 3.8. The difference is that, in addition to slaving subgoals to (similar) parents, postponement caching also allows slaving to similar subgoals anywhere in the space. Contrast, for example, the space in figure 5.10, which is the full search space using recursion control, with figure 5.11, which is the smaller full space using postponement caching.

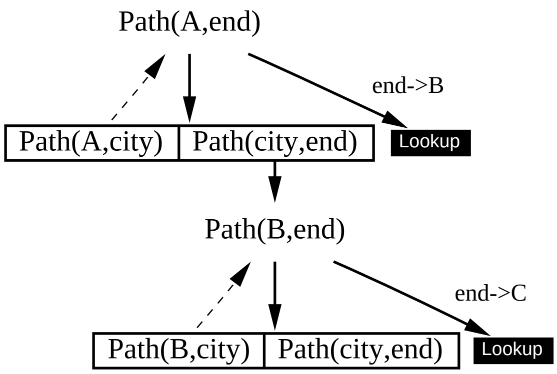


Figure 5.9: Second solution of `Path` query

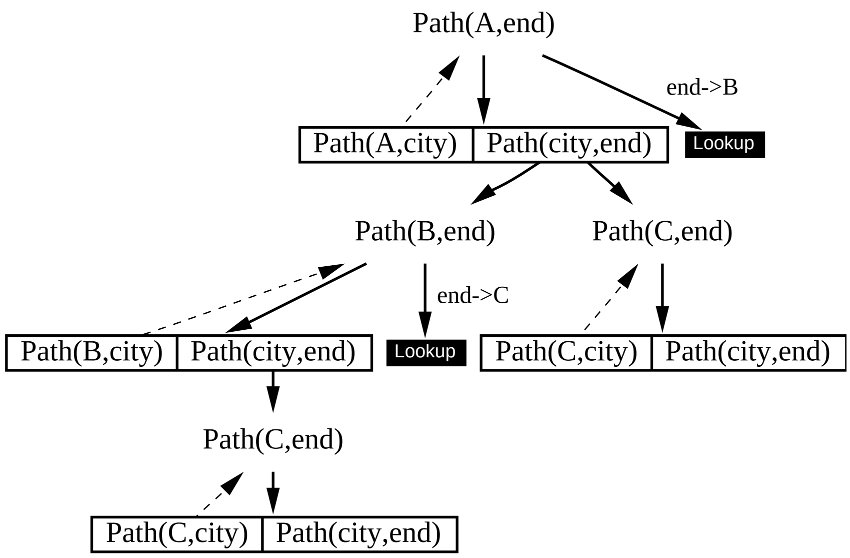


Figure 5.10: Proof space for all `Path` solutions

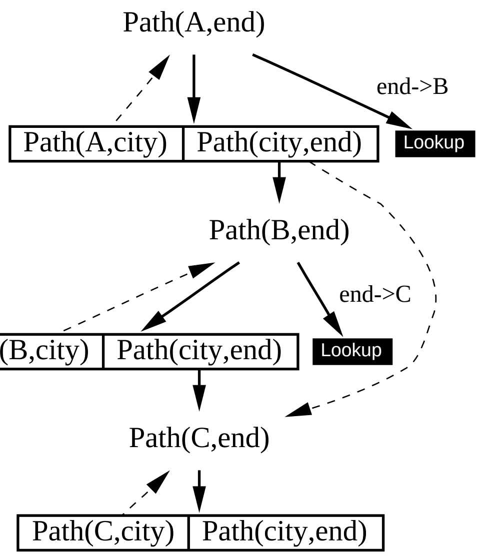


Figure 5.11: All `Path` solutions using postponement caching

### 5.4.2 Logic Programming

In logic programming, Warren [War92] gives an inference algorithm called OLDT. Warren describes PROLOG as being a non-deterministic language, where the interpreter carries out a depth-first search through the tree of possible alternative executions. In this scheme, OLDT is simply adding memoing to this procedural interpreter.

As Warren writes in [War92, pg. 97]:

> Intuitively, we think of a machine that is carrying out a nondeterministic procedure as duplicating itself at a point of choice, and as disappearing when it encounters failure. Thus at any time, we have a set of deterministic machines computing away. The set gets larger when any one has to make a nondeterministic choice, and it gets smaller when any one fails. To add memoing, we imagine a single global table containing every procedure call that has been made by any machine, and for each such call, the answers that have been returned for it. Since the situation is nondeterministic, there may be none, one, or many answers for any single call. Now each machine, before it makes a procedure call, looks in the global table to see if the call has already been made. If not, it adds the call to the table and continues computing. During its computation, whenever a machine returns from a procedure, it finds the associated call in the global table, adds the answer it has just computed, and continues computing. (If the answer is already in the table, then this answer is a duplicate, and the machine fails.) When a procedure is about to be called, if the call is found to be already in the table, then for each associated answer in the table, the machine must fork off a new copy of itself to continue the computation with that answer. It is possible that not all the answers are in the table at this time; some may still be in the process of being computed by other machines and will show up later. Thus when a machine encounters a call already in

the table, it forks off copies of itself to continue with the answers that are there, and it remains suspended on that table entry. Then whenever a new answer gets added to the table, the suspended machine makes a duplicate of itself to continue computing with that new answer. When a machine finishes its computation successfully, it disappears. The entire computation is complete when (and if) no machines are computing.

In the case of Horn databases, postponement caching is essentially equivalent to OLDT applied to a version of PROLOG that has sound unification and no `cut` operation.

### 5.4.3 Magic Sets

*This description of magic sets is adapted from a description by Warren [War92, pp. 99-101].*

The magic set algorithm [Ull89] in deductive databases provides much the same kind of control of infinite recursive spaces, by means of an automatic reformulation. (A similar rewriting of the given rules is the basis for the Alexander method [RLK86].)

The standard way to evaluate database queries is to evaluate them as expressions, using the relational operations to combine component relations. This is a bottom-up strategy that calculates new relations by combining old ones. The obvious way to evaluate recursive definitions is to begin with all the defined relations empty, and then iteratively compute new values for the relations, using the old relation values as input, until the relations produced as output are the same as the ones that are input, which indicates that a fixed point has been reached.

Consider the database in table 5.4. We start with inferred relations, like `Path`, being empty, while the extensional relation `Direct` contains the tuples

```
<A,B> <C,B> <B,D>
```

On the first iteration we can use the second rule to add those same tuples to the `Path` relation. (The first rule doesn't add any tuples because the `Path` relation is currently empty.)

```
Path(x,z)    ⇐  Direct(x,y) and Path(y,z)
Path(x,y)    ⇐  Direct(x,y)
Direct(A,B)
Direct(C,B)
Direct(B,D)
```

Table 5.4: Magic Set database before

The second iteration now uses the new value of `Path` that was computed on the first iteration. The second rule once again computes

```
<A,B> <C,B> <B,D>
```

for `Path`. The first rule now computes the join of the `Direct` and `Path` relations of the previous iteration and adds

```
<A,D> <C,D>
```

Thus the relation for `Path` at the end of the second iteration is

```
<A,B> <C,B> <B,D> <A,D> <C,D>
```

On the third iteration, the same value of `Path` is computed again, indicated that the least fixed point of the relational operator has been reached.

This bottom-up evaluation procedure terminates for any Datalog program over a finite domain. Duplicates can be eliminated at each step since the tuples are accumulated together. One source of inefficiency, however, is the redundant computation at each level that completely recomputes the previous level, as shown in the example. This is easy to avoid: New tuples generated on the current iteration can be distinguished from the old tuples. When generating the tuples for the next iteration, no new tuples are generated by using only old tuples computed on previous iterations.[4]

Another source of inefficiency is that this strategy may compute many tuples that are completely irrelevant to the query, which is only used at the final step to determine which of the computed tuples provide an answer via a selection. Computation of irrelevant tuples can be avoided in the nonrecursive relational algebra by a compile-time optimization known as "pushing selects in". If a query is expressed as first joining two large relations and then selecting a few tuples from that relation, the optimizer can determine that it is equivalent, but more efficient, to first select a few tuples from one of the relations and then join that much smaller relation with the other larger one to obtain the answer. In effect, the select operation has been pushed inside the join operation. Unfortunately, this optimization does not apply in the presence of recursion.

The solution to this problem, magic sets, can loosely be described as pushing selects in at run-time. A new literal is added to the body of the rule. This filtering relation is defined to be true of the "selecting values" that would occur in some call executed in a top-down evaluation of the query.

For example, adding the filtering relation to the rules of table 5.4 transforms the first two rules into the first pair shown in table 5.5. In general, every rule for some predicate `P` gets a new literal in its body consisting of the `Calls-To-P` applied to the arguments of the head. (Ground facts remain unaltered.)

The new relations, selecting for the calls that are possible when answering a query, also must be defined. If there is a rule like

```
P ⇐ Q and R and S
```

then we know that there will be a call to `S` whenever there is a call to `P` and the computation succeeds through `Q` and `R`. This knowledge can be captured using a rule like

```
Calls-To-S ⇐ Calls-To-P and Q and R
```

One additional fact must be added when the query arrives. Since the query itself defines the first call, a ground fact for `Calls-To-G` must be inserted for the query `G`.

For the query `Path(C,end)`, the transformed database of table 5.5 results. The base case `Path-Call` literal comes from the query. The last rule is generated from the first rule for `Path`: a

[4] This approach is called semi-naïve bottom-up evaluation [Ull88].

```
Path(x,z)          ⇐  Path-Call(x,z) and Direct(x,y) and Path(y,z)
Pathx,y            ⇐  Path-Call(x,y) and Direct(x,y)
Direct(A,B)
Direct(C,B)
Direct(B,D)
Path-Call(C,end)
Path-Call(y,z)     ⇐  Path-Call(x,z) and Direct(x,y)
```

Table 5.5: Magic Set database after

call to `Path` that succeeds through the `Direct` conjunct will cause another call to `Path`. To finally evaluate the query, the seminaïvebottom-up procedure is run on the new database.

On the first iteration, neither of the rules for `Path` applies, since the relation `Path-Call` (and `Path`) is empty. But we can add the ground facts to their respective relations, so `Direct` becomes

```
<A,B> <C,B> <B,D>
```

and `Path-Call` becomes

```
<C,end>
```

In the second iteration, the second rule for `Path` applies, so a new tuple is added to the previously-empty `Path` relation, resulting in

```
<C,B>
```

The `Path-Call` rule also applies, adding the tuple `<B,end>` and resulting in the expanded relation

```
<C,end> <B,end>
```

On the third iteration `<B,D>` is added to the `Path` relation, and `<D,end>` to the `Path-Calls` relation. On the fourth iteration, `<C,D>` joins `Path`. The fifth iteration produces nothing new, indicating that the fixed point has been reached. The final relations are

```
Direct     <A,B> <C,B> <B,D>
Path-Call  <C,end> <B,end> <D,end>
Path       <C,B> <B,D> <C,D>
```

Finally, a select can be done on the `Path` relation to return the results for our original query `Path(C,end)`, yielding the solutions

```
Path(C,B) and Path (C,D)
```

The algorithm presented here is essentially the Magic Template algorithm [Ram91], which is a generalization of the Magic Sets algorithm to handle tuples containing variables. Handling variables requires a more complex (and perhaps less efficient) mechanism than relational-algebra processors normally contain. By allowing the transformation to apply only to *range-restricted* Datalog programs,[5] it can be modified to generate programs that are guaranteed not to generate tuples with variables during bottom-up evaluation. In this situation the *magic* relations turn out to be projections of the `Calls-To-P` relations (*i.e.*, to contain only some of their columns). For example, in the table 5.5 database, `Path-Calls` would be a unary predicate and the fact defining it would be `Path-Calls(C)`, which is range-restricted, instead of the Path-Calls(C,end) in table 5.5, which is *not* range-restricted.

In postponement caching (and the OLDT logic programming algorithm), a data structure indicating dependencies is maintained, so that answers are returned directly to those computations desiring them. In the magic algorithm, no such pointers are maintained, and so answers are distributed to those needing them by (re-)performing a join.

Magic strategies are typically thought of as being bottom-up strategies, and backward chainers like postponement caching are thought of as top-down. As described here, the algorithms and computations are very similar. Adding memoing or postponement introduces a bottom-up component to a top-down strategy, and making the magic transformation introduces a top-down component to a bottom-up strategy. The resulting algorithm is essentially the same in either case.

Bry has given a common framework, the Backward Fixing Procedure [Bry90], which unifies the top-down and bottom-up approaches for Horn clause inference. The framework shows the equivalence of particular bottom-up magic set methods[6] and particular top-down memoing approaches[7].

Magic sets arose in the deductive database community. The technique is clearly superior to simple bottom-up approaches, as it avoids computing tuples which do not relate to the query. It is also superior to simple top-down approaches, which recompute the solutions to subgoals whenever the recur in the search space.

The question remains, though, how magic sets compare to more sophisticated top-down inference engines with caching, such as logic programming memoing or postponement caching. Ullman writes [Ull89]:

> There are a number of reasons why bottom-up calculation is preferable to top-down.
>
> 1. ... Top-down calculation ... can get trapped in infinite loops and never find the answer. ...
> 2. ... Detecting termination, even for datalog rules, is not easy.
> 3. ... We can be led to expand the same subgoal many times in the tree, thus repeating significant amounts of work.
> 4. The bottom-up algorithms can make use of efficient techniques for taking joins of massive relations. ... In comparison, the top-down approach tends to deal with many small relations, each associated with one of the nodes of the rule/goal tree. ...
> 5. Top-down algorithms require unification, but bottom-up algorithms only need term-matching, a simpler operation.

[5] These are programs where each variable in the head of a rule appears in the body.

[6] Magic sets and the Alexander method were shown to be specializations of the Backward Fixing Procedure.

[7] Bry considered the following memoing extensions to SLD-resolution: ET*, $\mathrm{ET}_{\mathrm{interp}}$, OLDT-resolution, SLDAL-resolution, and the RQA/FQI procedure

Ullman was comparing simple magic sets on datalog theories with a simple backward-chainer. We're actually more interested in comparing a generalized magic sets algorithm (which is complete for Horn theories), which a sophisticated top-down algorithm like memoing or postponement caching. In such a case, we can observe the following:

1. In the presence of functions there may be an infinite number of solutions to a query, in which case any complete algorithm can get trapped in an infinite loop. Simple recursive loops without functions, however, can be detected by the top-down strategies which notice that a new subgoal is an instance of some ancestor.

2. The computations between the two are essentially isomorphic, and so termination can be detected in much the same way.

3. Caching allows the algorithm to expand a given subgoal only once, and to look up that value whenever the subgoal recurs.

4. As mentioned previously, magic sets performs the same search of the space as these top-down approaches, but it does not keep the data structure representing the top-down space itself. This means that many more joins are actually required by the bottom-up approach, where as the top-down methods can efficiently propagate answers to those subgoals waiting for them.

5. When the language is more general than subgoal-rectified datalog (*e.g.* Horn clauses), bottom-up approaches are forced to use unification as well.

That was a defense of top-down approaches, given the challenge of the effective magic sets approach. We can also consider whether top-down approaches might have some advantages over a magic set-style bottom-up approach. First we dispense with two potential points that aren't real advantages:

1. Database problems are generally looking to find every solution to a query, whereas top-down approaches are often addressing a problem of finding a single solution. One might imagine that this could lead to an efficiency gain for the top-down approach.

   This potential gain is fairly minimal, though, since it is easy to imagine a version of magic sets which can return a single answer (or a finite number of answers). The bottom-up approach iteratively computes tuples. As soon as the required number of tuples have been produced, the process can be halted. The computations will have been essentially the same as the top-down approach. The normal fixed point version of magic sets is only needed if all answers are sought. Thus a single-answer scenario is not generally an advantage of the top-down approach.

2. Database scenarios generally involve relatively small rule sets and very large sets of ground facts (the EDB relations). AI scenarios often involve large sets of rules with only a few ground facts, such as the CYC commonsense knowledge base [LG88]. In the latter case, answering a query may only involve a small fraction of the database rules. Bottom-up approaches require rewriting each rule in the database, whether relevant to the query or not.

   As it turns out, actual magic set implementations typically explore enough of the top-down goal space to determine exactly which rules need to be rewritten. In any case, almost all of the rewriting is independent of the query, so the database can be rewritten once and then used to answer multiple queries.

The magic set rules are, however, generally larger (or more numerous) and less clear than the original rules, so this is a minor disadvantage for the rewriting approach.

The two approaches are not identical, however. The search tree maintained by top-down approaches provides additional information, and clever algorithms can take advantage of this extra knowledge.

1. Conjunct ordering can be delayed much longer in top-down approaches, resulting in more efficient orderings since more information is available. Bottom up approaches much decide what order to expand subgoals based only on the expected binding pattern of the variables. While this is a reasonable amount of information, it is less than that available to top-down approaches.

   Say that the database contained a rule `P(x)` $\Leftarrow$ `Q(x,y) and R(x,y)`. A magic set rewrite might know that the goal `P(x)` will only be called with a bound argument, and might rewrite the rule to take advantage of that. But it will have to choose between solving `Q(x,y)` first and solving `R(x,y)` first, knowing only that the variable `x` is bound.

   Imagine a situation where the top-down prover knew that `Q(1,y)` had only a few solutions, `R(1,y)` had many solutions, `Q(2,y)` had many solutions, and `R(2,y)` had only a few. (Also assume that looking up `Q` or `R` when both arguments are bound is a constant-time operation.) And say that, in the actual top-down search, only `P(1)` and `P(2)` ever appeared as subgoals. When attempting to solve `P(1)` using this rule, it is faster to compute the few `Q` tuples first, and then filter them through the `R` relation. But when solving `P(2)`, the opposite order is superior.

   Few top-down theorem provers actually exploit this flexibility[8] or consider such meta information when solving conjunctions, but the potential still exists with such an approach.

2. It is true that the magic set reformulation explores the same space as that implied by the query expanded with a backward-chaining inference rule. A top-down algorithm, however, can sometimes prune a portion of this space. This advantage is separate from the pruning resulting from caching subgoals, which is part of the sophisticated top-down approaches we are considering, and which arises naturally from the bottom-up approaches.

   For example, imagine a ground subgoal which has multiple possible proof paths. Each of these subspaces is part of the implicit top-down search space, and hence each will be explored by the bottom-up algorithms. If one branch has a very short proof, it is possible that a top-down algorithm can find the simple proof quickly. The other spaces can be pruned, even though they may actually correspond to valid proofs as well. It is the search space data structure which allows top-down algorithms to realize that the result of any computation on the other branches can at best yield a solution which is a duplicate of the one already known. Since

[8] The MRS system of Genesereth [Gen83] allowed the definition of an entire logical metatheory to describe how to solve particular baselevel subgoals.

bottom-up approaches don't have access to this data structure, they are forced to explore all possible subspaces.

A similar example can be constructed for non-ground goals, if the number of solutions is known. For example, there are only two solutions to `Parent(x,y)` if the first argument is bound, since a given person only has two parents. If two different parents have already been found, then all further subgoals in the service of a goal `Parent(A,y)` may be pruned.

3. Lastly, bottom-up approaches are basically stuck with a breadth-first search of the top-down space. Top-down approaches have the flexibility to trade off time for space, searching using depth-first search, iterative deepening, best-first search, or even the same breadth-first search. This choice of control may allow for much better performance on some problems.

Further analysis on the relationship of magic sets and postponement caching in the case of non-Horn theories can be found in section 6.3.

### 5.4.4 Other techniques

*The material in this section is adapted from Smith's description [SGG86, section 1.2].*

The primary benefit of the schemes discussed in this chapter is the transformation of some infinite search spaces (typically resulting from recursive rules) into finite spaces. There are other techniques to address the problem of infinite recursion, but all have characteristics that limit their applicability.

#### Breadth-first search

Breadth-first search is guaranteed eventually to find any answer present in the search space. If we are looking for all answers to a query, however, then breadth-first search will never halt. Barring other information about when to stop expanding the search space, the entire infinite space must be explored in order to guarantee that all answers have been found.

Even if only a specific number of answers are requested, breadth-first search will halt only if the space contains at least that many answers. As perhaps the most common case, only a single answer might be requested for a query that has no solution in the search space. Again, breadth-first search will not halt.

#### Selective Forward Inference

One approach to controlling recursion is to only allow recursive rules to be used for forward inference, in carefully selected cases. Unfortunately, a number of problems prevent this from being a feasible solution for many situations.

1. Since forward inference is not goal-directed, such a scheme results in the computation and storage of many irrelevant facts. This is typically a large source of inefficiency in the algorithm.

2. In the selection phase, one cannot limit forward inference to just the recursion rules. For example, consider the database in table 5.6. Here only the first rule is recursive, but if the second is not involved in the forward inference then the inference will be incomplete.

So unfortunately it is the case that if any axiom is restricted to forward inference, then all axioms that can be used in the proof of any of its premise clauses must also be subject to forward inference. This of course increases the number of irrelevant facts that must be computed and stored.

| | | |
|---|---|---|
| `Path(x,y) and Path(y,z)` | $\Rightarrow$ | `Path(x,z)` |
| `Bridge(x,y)` | $\Rightarrow$ | `Path(x,y)` |
| `Path(A,B)` | | |
| `Bridge(B,C)` | | |

Table 5.6: Forward inference: Many rules

3. Even forward inference can result in infinite deduction. The database in table 5.7 for computing Fibonacci numbers can cause an infinite loop in either the forward or backward directions.

| | | |
|---|---|---|
| `Fib(i)=z` | $\Leftarrow$ | `Fib(i-2)=x and Fib(i-1)=y and z=x+y` |
| `Fib(0) = 1` | | |
| `Fib(1) = 1` | | |

Table 5.7: Fibonacci numbers

### Reformulation

PROLOG programmers often use the technique of reformulation to facts such that the search space is no longer infinite. In the `Path` example, for instance, one can introduce a new predicate `Direct` meaning that the two cities are directly connected. Thus the old database of table 5.8 becomes the one in table 5.9.

| | | |
|---|---|---|
| `Path(x,z)` | $\Leftarrow$ | `Path(x,y) and Path(y,z)` |
| `Path(A,B)` | | |
| `Path(B,C)` | | |

Table 5.8: PROLOG reformulation before

The recursion on all left-hand branches of the tree has been eliminated. The search space is thus no longer infinite, as long as the `Direct` conjunct is solved before the `Path` conjunct when using the first rule.

What goes wrong with this?

| | | |
|---|---|---|
| `Path(x,z)` | $\Leftarrow$ | `Direct(x,y) and Path(y,z)` |
| `Path(x,y)` | $\Leftarrow$ | `Direct(x,y)` |
| `Direct(A,B)` | | |
| `Direct(B,C)` | | |

Table 5.9: PROLOG reformulation after

1. Reformulations work well for only a few of the possible forms of the query. For example, the above reformulation works well for the query `Path(A,end)`, but for the different query `Path(start,C)` the search explodes. Using the first rule, we get the intermediate subgoals

   `Direct(start,y) and Path(y,C)`

   If we solve the `Direct` conjunct first, we must explore every direct connection in the knowledge base. If we explore the `Path` conjunct first, the infinite search space returns.

   A different reformulation can work well for queries like `Path(start,C)`, but it in turn performs poorly for the original query `Path(A,end)`.

2. Reformulations are fragile. Adding the symmetric fact `Direct(B,A)` again leads to an infinite search space.

3. Reformulation can be an arbitrarily difficult programming task. If the original rule set had a symmetry rule in addition to the transitivity rule, the reformulation would require four new rules and another constructed relation. At each step, it becomes more difficult to understand, explain, and modify the reformulated rules.

4. Lastly, reformulation results in an implicit embedding of control information into the domain information. By merging domain and control facts, the resulting complex programs have little advantage over coding expert systems in more traditional programming languages.

However, it should be noted that the Magic Set algorithm described in the previous section is indeed a clever form of automatic reformulation, and it solves many (although not all) of the difficulties listed here.

# Chapter 6

# Non-Horn Postponement Caching

## 6.1 Incompleteness

As shown in section 4.4, subgoal solutions can be made independent of context. We can attempt to use this result to extend postponement caching to the case of non-Horn theories. The result appears to sanction an algorithm that only copies non-reduction answers across cache links. This is because if a reduction answer would have occurred in some new place, then a non-reduction answer will occur in the original place by the construction shown previously.

This algorithm unfortunately turns out to be incomplete, as it is possible for the cache links to get into a deadlock. Note that this is different from just having a cycle in the inference graph: the Horn clause recursion control of section 5.4.1 in fact relies on cycles in the graph.

Consider the example in table 6.1.[1] A normal model elimination proof (without caching) of the goal `X and Y` is easy. The chain sequence is shown in table 6.2 and the corresponding space is in figure 6.1. When using postponement caching, however, the search completes with no proof found (and thus the algorithm is incomplete). As figure 6.2 shows, the subgoal `X` is in effect waiting for a solution to the subgoal `Y`, which itself is waiting for a solution to the original `X`. (After this chain of reasoning fails, various other contrapositives allow other parts of the space to be explored, but they all eventually slave to something already in the space, and thus they fail too.)

| | | |
|---|---|---|
| X | $\Leftarrow$ | A and B |
| A | $\Leftarrow$ | $\neg$X |
| B | $\Leftarrow$ | $\neg$X |
| Y | $\Leftarrow$ | C and D |
| C | $\Leftarrow$ | $\neg$Y |
| D | $\Leftarrow$ | $\neg$Y |

Table 6.1: Query: `X and Y`

[1] I am indebted to H. Scott Roy for this counterexample.

| | |
|---|---|
| 1. | X Y |
| 2. | A B [X] Y |
| 3. | ¬X [A] B [X] Y |
| 4. | B [X] Y |
| 5. | ¬X [B] [X] Y |
| 6. | Y |
| 7. | C D [Y] |
| 8. | ¬Y [C] D [Y] |
| 9. | D [Y] |
| 10. | ¬Y [D] [Y] |
| 11. | □ |

Table 6.2: Success of `X and Y` with no caching

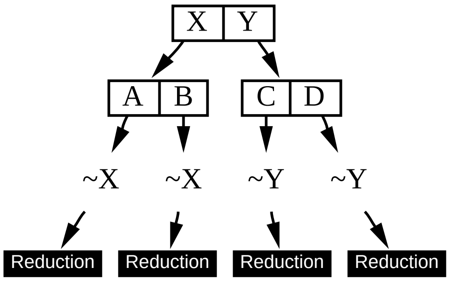


Figure 6.1: Success of `X and Y` with no caching

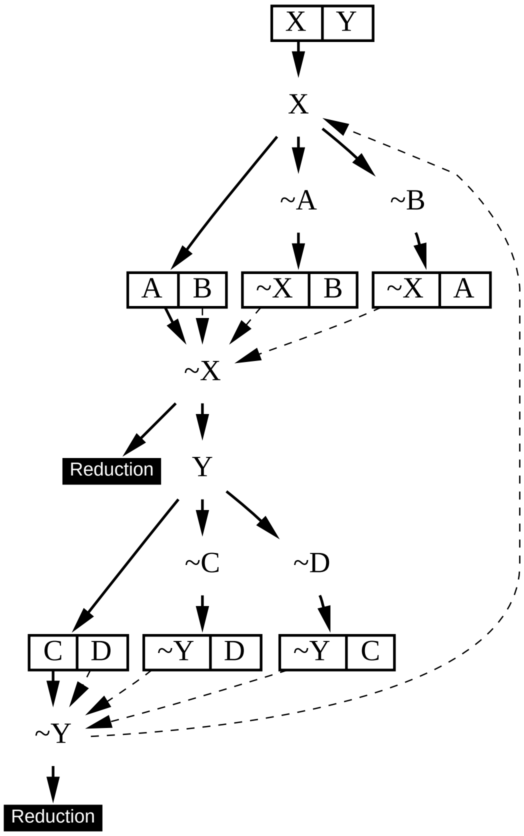


Figure 6.2: Failure with postponement caching

So what goes wrong in this example? Let's follow the construction of section 4.4.2 on this example. We prove the subgoal `A` successfully, using a reduction proof involving the parent `X`. The next subgoal, `B`, slaves to the existing `¬X` subgoal, which has a single answer we can't use, as it's a reduction answer. The construction suggests that there will be a different, direct, proof, that we can find it by following the missing reduction proof to the goal. And, indeed, the subgoal `¬X` immediately reduces to `Y`, the conjunctive sibling of the current goal `X`.

If the original goal had a proof (from the empty context), then the subgoal `Y` should have one also (from a more complex context). So we continue to try to prove `Y` as we would have without caching. The subgoal `C` succeeds easily. The subgoal `D` slaves to the now existing `¬Y` that needs a direct answer. And now the loop occurs, as that direct answer depends on proving `X`, which indeed follows from the database, but which we are unfortunately currently waiting for.

**Theorem 35 (Incompleteness of postponement caching)** *The postponement caching algorithm is not complete for non-Horn theories.*

**Proof.** The example in table 6.1 is a counterexample. ∎

## 6.2 Future Work

Is the incomplete postponement caching algorithm patchable? A small patch, such as checking for reduction solutions before examining the cache, is susceptible to a counterexample which is a slightly extended version of table 6.1. All that is required is to separate the reduction from the cached subgoal and the same loop reappears, as shown in figure 6.3 which uses an intermediate subgoal `I`.

More promising is an attempt to reuse those context-sensitive answers that are still valid in the new context. This is similar to the original ideas for caching in first-order inference, but it is now the *solutions* that become context-dependent, not the subgoals themselves. Consider then this augmented postponement caching scheme, where answers to subgoals are annotated with the required ancestors that enabled the proof to succeed. If those same ancestors are present in the new context, then the proof (and thus the recorded solution) is valid there as well.

This new algorithm easily solves the previous counterexample, as shown in figure 6.4. When the child of `B` slaves to the existing `¬X`, the stored solution (namely, "true if `X` is an ancestor") is still valid in the new context, and is thus copied immediately. The same pattern occurs for subgoals `C` and `D`.

### 6.2.1 The Pigeonhole Problem

Unfortunately, even augmented postponement caching is not complete for non-Horn theories, as can be shown by attempting to prove the 4-in-3 pigeonhole problem. The generic pigeonhole problem of size $N$ is to try to fit $N$ pigeons in $N-1$ holes, where every pigeon must be in some hole and no two pigeons may be assigned to the same hole. This problem is of occasional interest in theorem proving:

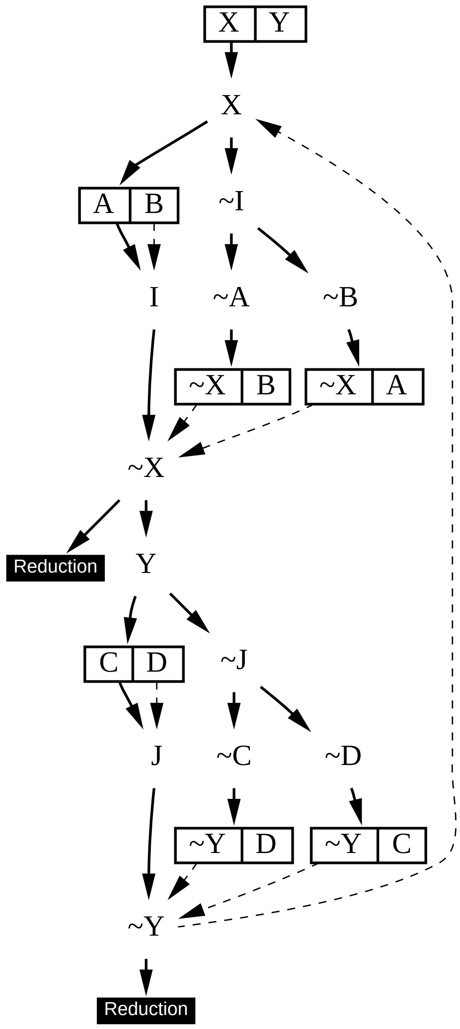


Figure 6.3: Failure with patched postponement caching

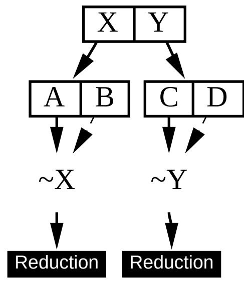


Figure 6.4: Success with augmented postponement caching

1. By a simple counting argument, it is easy to see that the query follows from the database. However this kind of meta-proof is generally outside the scope of theorem provers, allowing us easily to construct a problem of arbitrary difficulty.
2. Proofs of the query using only resolution, on propositional versions of the database, are known to be exponential in the size of the query. Hence the proof space grows quite rapidly, providing a good scaling test for particular provers. (The reasoning must essentially be done by cases, and the number of cases grows exponentially.)

The problem is of interest to us in this section for a different reason: none of the subgoals (*e.g.* that a particular pigeon must be in a particular hole) actually follows from the database, and thus the only proofs of any subgoal are context-sensitive reduction proofs.

The smallest pigeonhole problem of interest to us has $N = 4$, with four pigeons and three holes. We will label the pigeons A, B, C, and D, and the holes 1, 2, and 3. For further simplification, we'll ground out all the clauses. A proposition like "B3" will indicate the concept of pigeon B being located in hole 3. The database is shown in table 6.3 with a goal of

```
¬D1 and ¬D2 and ¬D3
```

*i.e.*, that there is no location to place the last pigeon.

A proof of the query from the database (using perfect search control, and without caching) is shown in figure 6.5. Basic postponement caching solves the first conjunct (figure 6.6) in isolation, but fails to solve even the first conjunct when the query is `¬D1 and ¬D2`, as shown in figure 6.7.

Augmented postponement caching solved our previous counterexample (table 6.1), but unfortunately fails to duplicate that success here. It too solves the first conjunct (figure 6.8) in isolation. When queried about two conjuncts, it solves the first but fails on the second (figure 6.9). (Note that, because the negated goal plays a significant role in these inference spaces, these different queries are only loosely related.) For comparison purposes, the full (failing) space for the original three-conjunct query is shown in figure 6.10.

## 6.3 Related Work: Magic Sets

Traditional top-down inference engines suffer from subgoal repetition, in that the same subproblems are often solved and re-solved many times. Traditional bottom-up inference engines suffer from irrelevancy, in that many subgoals are derived that are unrelated to the query. Memoing (section 5.4.2) eliminates the repetition problem in a top-down system, by noticing similar subgoals and solving each only once. The technique of magic sets (section 5.4.3) adds filtering subgoals to each forward-inference rule, such that only relevant subgoals are derived when processing the rule set bottom-up. As shown by Bry [Bry90], these two approaches result in the same fundamental computations.

The postponement caching described in this chapter is a (failed) attempt to combine the failure caching results from chapter 4 with the top-down approach of memoing, so that caching can be effective in non-Horn theorem proving.

To compare this effort with the bottom-up approach, the magic set technique must first be extended to non-Horn theories. As Bry described, the magic set technique basically adds a meta-description of a top-down inference engine to the database rule. This is similar in spirit to the

```
¬D1 or ¬A1
¬D1 or ¬B1
¬D1 or ¬C1
¬D2 or ¬A2
¬D2 or ¬B2
¬D2 or ¬C2
¬D3 or ¬A3
¬D3 or ¬B3
¬D3 or ¬C3
¬A1 or ¬B1
¬A1 or ¬C1
¬A2 or ¬B2
¬A2 or ¬C2
¬A3 or ¬B3
¬A3 or ¬C3
¬B1 or ¬C1
¬B2 or ¬C2
¬B3 or ¬C3
A1 or A2 or A3
B1 or B2 or B3
C1 or C2 or C3
```

Table 6.3: Four-in-three pigeonhole problem

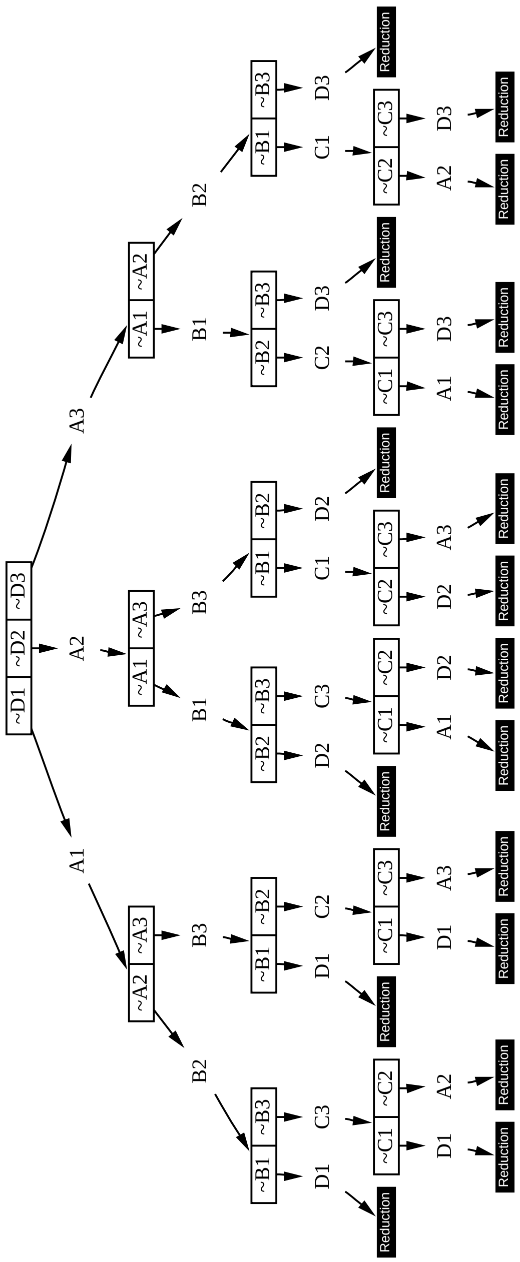


Figure 6.5: Pigeonhole solution

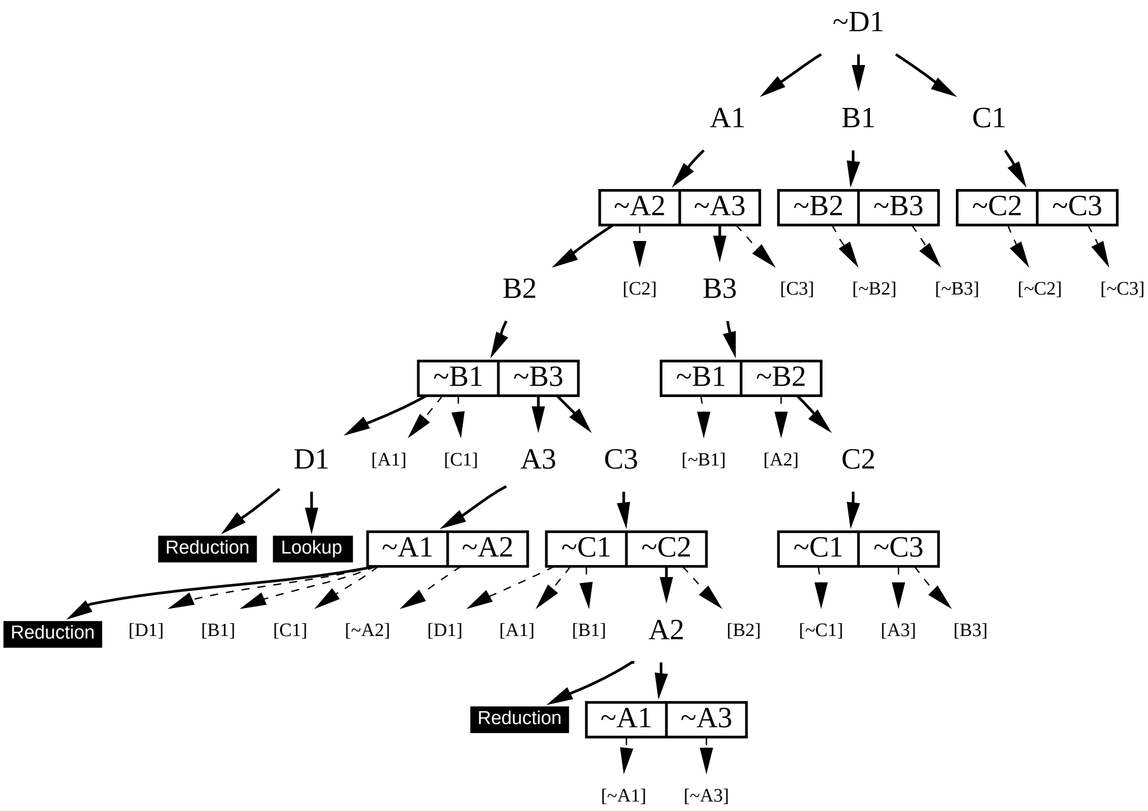


Figure 6.6: Pigeonhole: Basic postponement (1 conjunct)

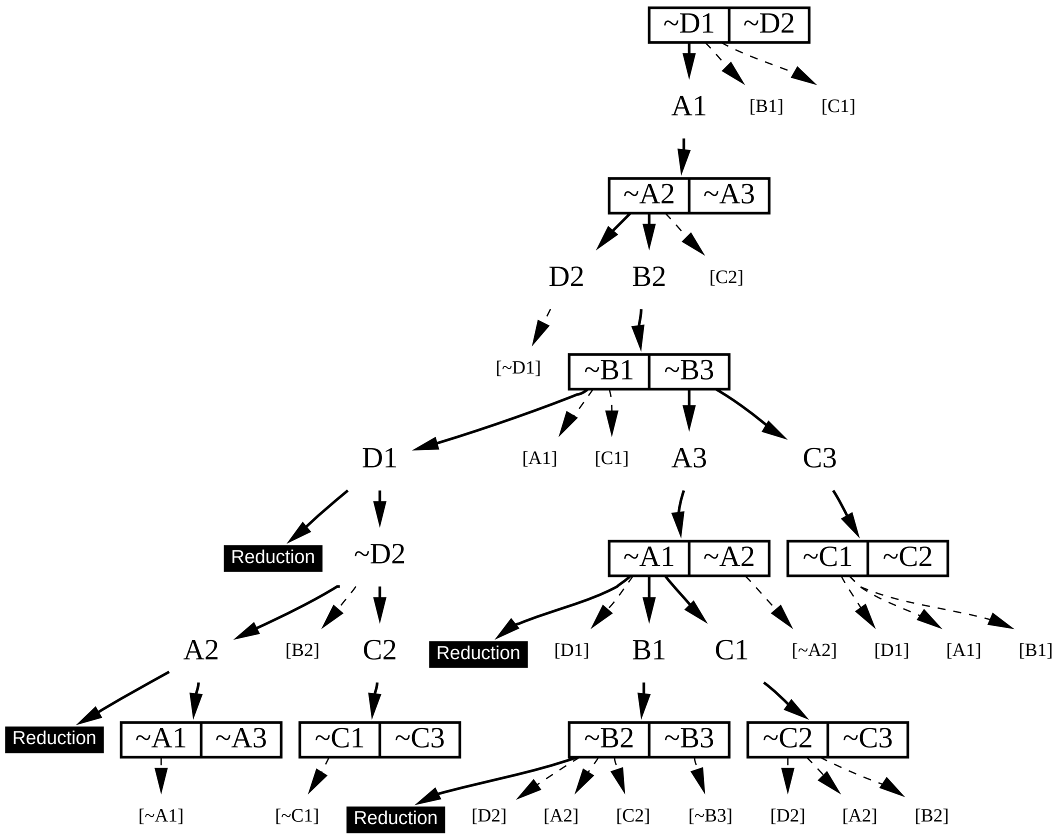


Figure 6.7: Pigeonhole: Basic postponement (2 conjuncts)

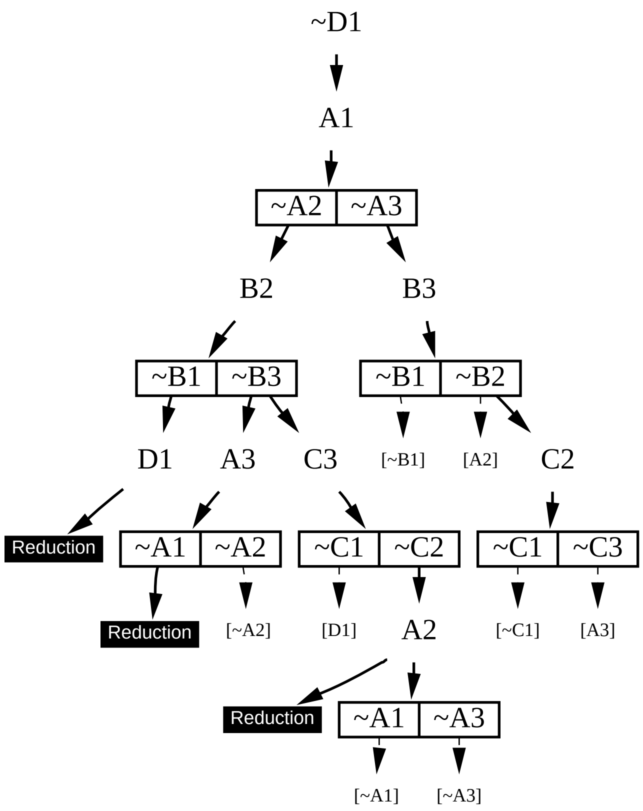


Figure 6.8: Pigeonhole: Augmented postponement (1 conjunct)

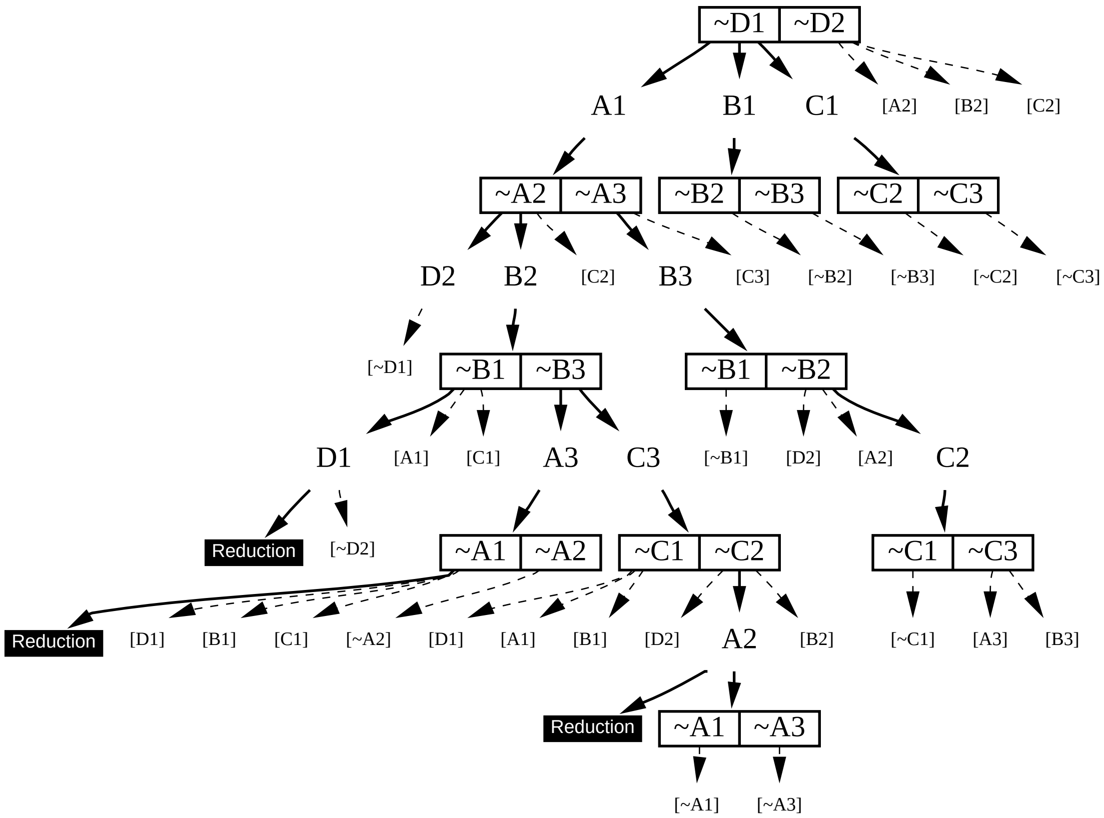


Figure 6.9: Pigeonhole: Augmented postponement (2 conjuncts)

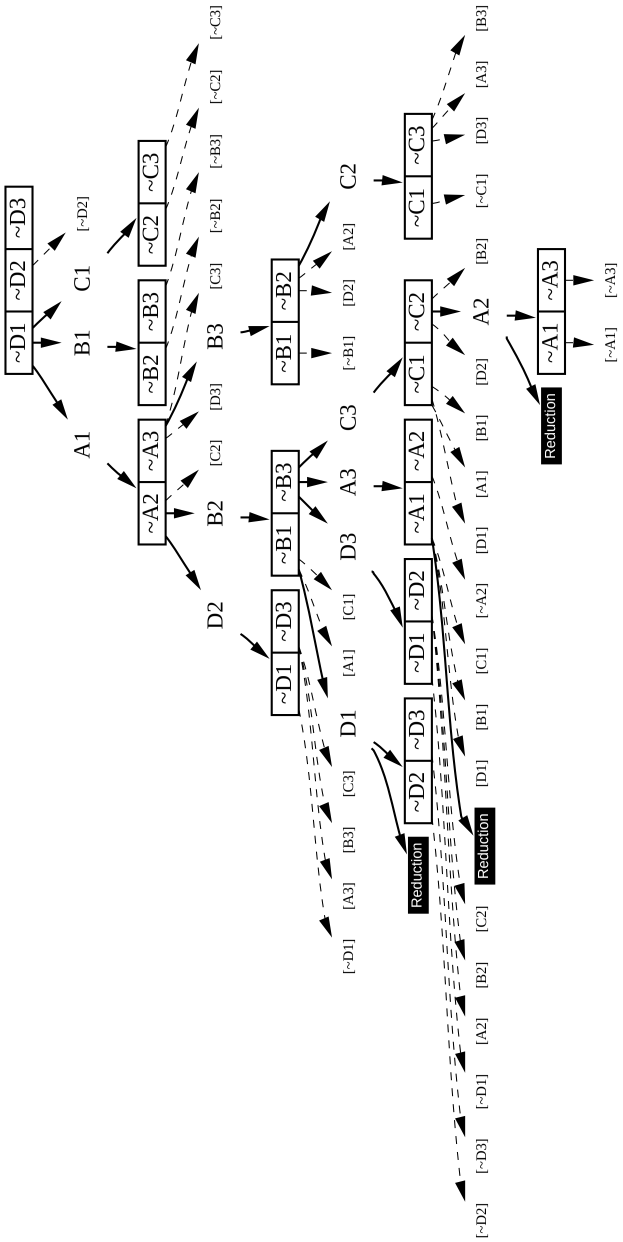

Figure 6.10: Pigeonhole: Augmented postponement (3 conjuncts)

definition of a Lisp interpreter written in Lisp, or a Prolog interpreter in Prolog. In the first case, the definition is easy because of building on Lisp's `eval` function. In the second case the definition is easy because of the use of Prolog's unification routines. (The caching behavior which must be explicit in the top-down memoing approaches comes for free in the bottom-up systems, via the standard semi-naïve fixed-point evaluations.)

The usual magic set method describes the top-down behavior of a simple Horn clause logic programming system. To convert this to an effective inference engine for non-Horn theories, we must instead describe the top-down behavior of a non-Horn inference engine. There are many such algorithms, but one of the most popular (and the one considered in this thesis) is model elimination [Lov78]. With the addition of a single rule of inference, namely a reduction (goal-goal resolution) between a subgoal and an ancestor of that subgoal, a Prolog-like inference engine becomes complete for non-Horn theories.[2]

Stickel [Sti94] has given such a bottom-up rewriting for non-Horn inference. The description here is adapted from his.[3]

The metatheoretic predicate `Fact` has two arguments: a literal and a set of ancestor subgoals sufficient to prove it. The bottom-up interpretation of the rule

$$\mathtt{P} \Leftarrow \mathtt{Q}_1 \text{ and } \ldots \text{ and } \mathtt{Q}_m$$

can be expressed by

$$\begin{aligned}&\mathtt{Fact(P,(a}_1 \cup \ldots \cup \mathtt{a}_m\mathtt{)\text{-}\{P\})} \Leftarrow \\ &\qquad\qquad \mathtt{Fact(Q}_1\mathtt{,a}_1\mathtt{)} \text{ and } \ldots \text{ and } \mathtt{Fact(Q}_m\mathtt{,a}_m\mathtt{)}\end{aligned}$$

This sentence can be interpreted as saying: The subgoal `P` is true in some particular context (*i.e.* with a particular set of ancestors $\cup \mathtt{a}_i$) if each subgoal $\mathtt{Q}_i$ is true in a subset context (or else using `P` itself as an ancestor).

To describe the reduction operation, we must say that a fact follows if the complement of the single literal subgoal is in the set of ancestors. (In addition, if the subgoal unifies with a complement of an ancestor, that is also a valid derivation.) The simple version of this is just that a fact is true with a negated version of itself as an ancestor:

$$\mathtt{Fact(x,\{\neg x\})}$$

Single literal facts are translated to

$$\mathtt{Fact(F,\emptyset)}$$

As with Horn magic set rewritings, the top-down progress of goals must be described. For the same rule

$$\mathtt{P} \Leftarrow \mathtt{Q}_1 \text{ and } \ldots \text{ and } \mathtt{Q}_m$$

[2] Prolog itself must be modified in order to be complete even for Horn theories: unification must be made sound, via the addition of an occurs check, and the search strategy must be complete, *e.g.* by using iterative deepening. Stickel [Sti89] describes a version of Prolog which has been modified in this way.

[3] Stickel's auxiliary sets were subgoals that could be assumed. Here we use the negated version of this idea, so that the auxiliary sets are ancestor subgoals.

we have

$$\begin{aligned}
&\texttt{Goal}(\texttt{Q}_1,\texttt{a}\cup\{\texttt{P}\}) \Leftarrow \texttt{Goal}(\texttt{P},\texttt{a})\\
&\texttt{Goal}(\texttt{Q}_2,\texttt{a}\cup\{\texttt{P}\}) \Leftarrow \texttt{Goal}(\texttt{P},\texttt{a}) \texttt{ and } \texttt{Fact}(\texttt{Q}_1,\texttt{a}_1) \texttt{ and } \texttt{a}_1 \subseteq \texttt{a}\cup\{\texttt{P}\}\\
&\ldots\\
&\texttt{Goal}(\texttt{Q}_m,\texttt{a}\cup\{\texttt{P}\}) \Leftarrow \texttt{Goal}(\texttt{P},\texttt{a})\\
&\qquad \texttt{ and } \texttt{Fact}(\texttt{Q}_1,\texttt{a}_1) \texttt{ and } \texttt{a}_1 \subseteq \texttt{a}\cup\{\texttt{P}\} \texttt{ and } \ldots\\
&\qquad \ldots \texttt{ and } \texttt{Fact}(\texttt{Q}_{m-1},\texttt{a}_{m-1}) \texttt{ and } \texttt{a}_{m-1} \subseteq \texttt{a}\cup\{\texttt{P}\}
\end{aligned}$$

The single literal goal `G` is translated to

$$\texttt{Goal}(\texttt{G},\emptyset)$$

*i.e.* a proof a `G` with no ancestors is sought.

Note the following subsumption conditions:

- `Fact`$(f,a)$ subsumes `Fact`$(f',a')$, where $f' = f\theta$ and $a' \supseteq a\theta$ for some substitution $\theta$. Facts that are less general or require more assumptions can be deleted.
- `Fact`$(f,a)$ subsumes `Goal`$(g,a')$ where $g = f\theta$ and $a' \supseteq a\theta$ for some substitution $\theta$. such facts solve the goal without instantiating it.

Various model elimination pruning strategies can be introduced as well. For example, identical ancestor pruning (section 2.5.2) implies that

- `Goal`$(g,a)$ can be deleted if $g \in a$

Other deletions are possible, such as if $\neg g \in a$ or $a$ contains complementary literals.

As an example, consider the proof that `P and Q` follows from `P or Q`, `¬P or Q`, and `P or ¬Q`. We need a single literal goal (`G`), and the contrapositives of all rules. (Since the goal is a single literal, it is not necessary to add the negated goal to the database.) Hence the initial database is as shown in table 6.4. The non-Horn magic set rewriting of the facts is shown in table 6.5, and the description of the goals is shown in table 6.6.

| | | | |
|---|---|---|---|
| 1. | P | ⇐ | ¬Q |
| 2. | Q | ⇐ | ¬P |
| 3. | ¬P | ⇐ | ¬Q |
| 4. | Q | ⇐ | P |
| 5. | P | ⇐ | Q |
| 6. | ¬Q | ⇐ | ¬P |
| 7. | G | ⇐ | P and Q |

Table 6.4: `P` and `Q`: A non-Horn theory

A bottom-up derivation from these magic rules is shown in table 6.7.

| | | | |
|---|---|---|---|
| 1. | Fact( P,$a_f$-{ P}) | $\Leftarrow$ | Goal( P,$a_g$) |
| | | | and Fact($\neg$Q,$a_f$) and $a_f \subseteq a_g \cup$ { P} |
| 2. | Fact( Q,$a_f$-{ Q}) | $\Leftarrow$ | Goal( Q,$a_g$) |
| | | | and Fact($\neg$P,$a_f$) and $a_f \subseteq a_g \cup$ { Q} |
| 3. | Fact($\neg$P,$a_f$-{$\neg$P}) | $\Leftarrow$ | Goal($\neg$P,$a_g$) |
| | | | and Fact($\neg$Q,$a_f$) and $a_f \subseteq a_g \cup$ {$\neg$P} |
| 4. | Fact( Q,$a_f$-{ Q}) | $\Leftarrow$ | Goal( Q,$a_g$) |
| | | | and Fact( P,$a_f$) and $a_f \subseteq a_g \cup$ { Q} |
| 5. | Fact( P,$a_f$-{ P}) | $\Leftarrow$ | Goal( P,$a_g$) |
| | | | and Fact( Q,$a_f$) and $a_f \subseteq a_g \cup$ { P} |
| 6. | Fact($\neg$Q,$a_f$-{$\neg$Q}) | $\Leftarrow$ | Goal($\neg$Q,$a_g$) |
| | | | and Fact($\neg$P,$a_f$) and $a_f \subseteq a_g \cup$ {$\neg$Q} |
| 7. | Fact( G,$a_p \cup a_q$-{G}) | $\Leftarrow$ | Goal( G,$a_g$) |
| | | | and Fact( P,$a_p$) and $a_p \subseteq a_g \cup$ { G} |
| | | | and Fact( Q,$a_q$) and $a_q \subseteq a_g \cup$ {G} |
| Fact: | Fact(x,{$\neg$x}) | | |

Table 6.5: P and Q: Non-Horn magic facts

| | | | |
|---|---|---|---|
| 1. | Goal($\neg$Q,a$\cup$ { P}) | $\Leftarrow$ | Goal( P,a) |
| 2. | Goal($\neg$P,a$\cup$ { Q}) | $\Leftarrow$ | Goal( Q,a) |
| 3. | Goal($\neg$Q,a$\cup$ {$\neg$P}) | $\Leftarrow$ | Goal($\neg$P,a) |
| 4. | Goal( P,a$\cup$ { Q}) | $\Leftarrow$ | Goal( Q,a) |
| 5. | Goal( Q,a$\cup$ { P}) | $\Leftarrow$ | Goal( P,a) |
| 6. | Goal($\neg$P,a$\cup$ {$\neg$Q}) | $\Leftarrow$ | Goal($\neg$Q,a) |
| 7a. | Goal( P,a$\cup$ { G}) | $\Leftarrow$ | Goal( G,a) |
| 7b. | Goal( Q,a$\cup$ { G}) | $\Leftarrow$ | Goal( G,a) |
| Goal: | Goal(G,$\emptyset$) | | |

Table 6.6: P and Q: Non-Horn magic goals

| No. | Derived tuple | Justification |
|---|---|---|
| 1. | `Goal( G,∅)` | Initial goal |
| 2. | `Goal( P,{G})` | Goal rule 7a |
| 3. | `Goal( Q,{P,G})` | Goal rule 5 |
| 4. | `Fact(¬P,{P})` | Initial fact |
| 5. | `Fact( Q,{P})` | Fact rule 2 |
| 6. | `Fact( P,∅)` | Fact rule 5 |
| 7. | `Fact( Q,∅)` | Fact rule 4 |
| 8. | `Fact( G,∅)` | Fact rule 7 |

Table 6.7: `P` and `Q`: Bottom-up derivation of the goal `G`

This now gives us a bottom-up evaluation of non-Horn theories. How do the results of this thesis relate to such a framework? They essentially correspond to complex subsumption rules. Consider the simple failure caching of chapter 4. The results in that chapter basically state that if a subgoal in one context fails to have a completion, then the same subgoal can be pruned in any other context. Assume some subgoal `P` occurs below the goal `G` in two places, once down a path

$\mathtt{G}\rightarrow\mathtt{A}_1 \rightarrow\mathtt{A}_2 \rightarrow\mathtt{P}$

and once down a path

$\mathtt{G}\rightarrow\mathtt{A}_3 \rightarrow\mathtt{A}_4 \rightarrow\mathtt{P}$

In such a situation, the bottom-up meta-interpretation would derive a tuple

$\mathtt{Goal(P,\{A_2,A_1,G\})}$

and then later derive a tuple

$\mathtt{Goal(P,\{A_4,A_3,G\})}$

The two tuples appear to be different, and hence the bottom-up evaluation would continue to explore the space below the second one. If the first tuple led to no additional facts, then the second would not either.

This kind of pruning is relatively easy to implement in a top-down algorithm, because the search space data structure is available for inspection, and solutions to subgoals are indexed with the subgoals themselves. Such connecting data structures are typically lost in the bottom-up rewriting. It probably is possible to write a more sophisticated meta-description of the top-down algorithms, such that the bottom-up methods have the ability to perform this same kind of pruning, but such a re-writing would no doubt be very complex.

To implement failure caching bottom up, the algorithm must know whether a particular `Goal` tuple ever led to useful `Fact` tuples. (It also must know that the `Goal` tuple had been fully explored, *i.e.* a local fixed point was reached.) To implement postponement caching bottom up, even more information is required. Rather than just knowing about the existence of derived `Fact` tuples, such an algorithm would have to be able to associate particular `Fact` tuples with the expansion of particular `Goal` tuples. This is so that the algorithm can immediately derive similar `Fact` tuples when similar `Goal` tuples are encountered (as well as pruning further forward inference on the now-redundant `Goal` tuples).

This kind of pointer structure is exactly the data kept by top-down approaches, and it appears infeasible to attempt to take advantage of the same results in a bottom-up implementation.

# Appendix A

# Implementation

The caching strategies mentioned in this thesis have been implemented in a model elimination-style theorem prover called DTP [Ged]. The source code is written in Common Lisp with some CLtL2 [Ste90] extensions (*e.g.* the LOOP macro). It was developed under Franz Allegro CL 4.2.beta.0 on a Sun Sparc, and occasionally tested on MCL 2.0p2 (Apple Macintosh) and Lucid HP Common Lisp Rev. A.04.01 (HP-9000 Series 300/400). DTP is available on the World Wide Web at

`<URL:http://logic.stanford.edu/software/dtp/>`

The DTP prover provides a sound and complete inference engine for full first-order predicate calculus. Functions are permitted, but there is no special reasoning for equality. Subgoal caching options include the ones mentioned in chapter 3 (success, failure, answers, subgoal, and generalizations of each), as well as recursion control (section 5.4.1) and postponement caching (chapter 5). The core inference engine also implements the refinements of iterative deepening, identical ancestor pruning, pure literal elimination, and backjumping when solving conjunctions. (See section 2.5 for more details about these refinements.)

There are two auxiliary systems that make DTP more useful. For displaying postscript graphs of the proof spaces, AT&T's `dot` program is required. Contact Stephen North at

`<URL:mailto:north@research.att.com>`

for more information. (Of course, some mechanism for viewing the output is also needed: either a postscript previewer like the public domain unix utility `ghostview`, or else a postscript printer.)

A large collection of theorem proving examples is the TPTP (Thousands of Problems for Theorem Provers) collection [SSY94]. It is available on the World Wide Web at

`<URL:http://wwwjessen.informatik.tu-muenchen.de/~suttner/tptp.html>`

# Bibliography


[AHU87] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, Menlo Park, California, 1987.

[AS91] Owen L. Astrachan and Mark E. Stickel. Caching and lemma use in model elimination theorem provers. Technical Note 513, SRI International, November 1991.

[Ast92] Owen L. Astrachan. *Investigations in Model Elimination Based Theorem Proving*. PhD thesis, Duke University, Durham, North Carolina, December 1992.

[BF94] Peter Baumgartner and Ulrich Furbach. Model elimination without contrapositives. In *Proceedings of the 12th International Conference on Automated Deduction (CADE-12)*, pages 87–101, Germany, 1994. Springer-Verlag.

[Bla68] F. Black. A deductive question-answering system. In M. Minsky, editor, *Semantic Information Processing*, pages 354–402. MIT Press, Cambridge, MA, 1968.

[Bry90] François Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering*, 5(4):289–312, October 1990.

[Ged] Donald F. Geddis. DTP: Don's Theorem Prover. Available online as `<URL:http://logic.stanford.edu/dtp/>`.

[Ged95] Donald F. Geddis. *Caching and Non-Horn Inference in Model Elimination Theorem Provers*. PhD thesis, Stanford University, Stanford, CA, 1995. Available online as `<URL:http://logic.stanford.edu/papers/geddis-thesis.ps>`.

[Gen83] Michael R. Genesereth. MRS: A metalevel representation system. Technical Report HPP–83–28, Knowledge Systems Laboratory, Stanford University, 1983.

[Gin93a] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research (JAIR)*, 1:25–46, 1993. Available online as `<URL:ftp://t.uoregon.edu/papers/dynamic.dvi>`. JAIR is published online as `<URL:news:comp.ai.jair.papers>` and `<URL:http://www.cs.washington.edu/research/jair/home.html>`.

[Gin93b] Matthew L. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1993.

[GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[Kor85] Richard E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kor92] Richard E. Korf. Linear-space best-first search: Summary of results. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI–92)*, pages 533–538, Menlo Park, California, 1992. AAAI Press.

[LG88] Doug Lenat and R. V. Guha. The world according to CYC. MCC Technical Report ACA–AI–300–88, MCC, September 1988.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Germany, 2nd edition, 1987.

[Lov78] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland Publishing Company, 1978.

[MS81] D. P. McKay and S. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 368–374, Vancouver, BC, 1981.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, CA, 1980.

[Pla88] David A. Plaisted. Non-horn clause logic programming without contrapositives. *Journal of Automated Reasoning*, 4(3):287–325, 1988.

[Pla90] David A. Plaisted. A sequent style model elimination strategy and a positive refinement. *Journal of Automated Reasoning*, 6(4):389–402, 1990.

[Pla94] David A. Plaisted. The search efficiency of theorem proving strategies. In *Proceedings of the 12th International Conference on Automated Deduction (CADE-12)*, pages 57–71, Germany, 1994. Springer-Verlag.

[Ram91] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3–4):189–216, 1991.

[RLK86] J. Rohmer, R. Lescœur, and J. M. Kerisit. The Alexander method: A technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4(3), 1986.

[Rob65] J. A. Robinson. A machine-oriented logic based on the rsolution principle. *Journal of the Association for Computing Machinery (JACM)*, 12(1):23–41, January 1965.

[SGG86] David E. Smith, Michael R. Genesereth, and Matthew L. Ginsberg. Controlling recursive inference. *Artificial Intelligence*, 30(3):343–389, December 1986.

[Spe90] Bruce Spencer. Avoiding duplicate proofs. In Saumya Debray and Manuel Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference*, pages 569–584, Cambridge, Massachusetts, 1990. The MIT Press.

[SS93] A. M. Segre and D. Scharstein. Bounded-overhead caching for definite-clause theorem proving. *Journal of Automated Reasoning*, 11(1):83–113, August 1993.

[SSY94] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In *Proceedings of the 12th International Conference on Automated Deduction (CADE-12)*, pages 252–266, Germany, 1994. Springer-Verlag. Available online as `<URL:http://wwwjessen.informatik.tu-muenchen.de/ suttner/tptp.html>`.

[Ste90] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[Sti88] Mark E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, December 1988.

[Sti89] Mark E. Stickel. A prolog technology theorem prover: A new exposition and implementation in prolog. Technical Note 464, SRI International, June 1989.

[Sti94] Mark E. Stickel. Upside-down meta-interpretation of the model elimination theorem-proving procedure for deduction and abduction. *Journal of Automated Reasoning*, 13(2):189–210, 1994.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1803 Research Boulevard, Rockville, MD 20850, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press, 1803 Research Boulevard, Rockville, MD 20850, 1989.

[War92] David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, March 1992.