CLIN
An Automated Reasoning System Using Clause Linking

by

Shie-Jue Lee

A Dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1990

Approved by:

Advisor: David A. Plaisted

Reader: Donald W. Loveland

Reader: Gopalan Nadathur

Acknowledgements

I am deeply grateful to my advisor, Professor David Plaisted, for his guidance, support and encouragement throughout the course of this research. This dissertation could never have been done without him. I also thank the other committee members Professor Donald Loveland, Professor Gopalan Nadathur, Professor Jan Prins, Professor Donald Stanat, Professor AKhilesh Tyagi, and Professor Jennifer Welch for their valuable comments and suggestions.

I gratefully acknowledge the financial support provided by the National Science Foundation (Grant CCR-8802282).

No expression of gratitude is adequate to my family, who never fail to give their support through my course of study. Especially I offer thanks to my brother, Wen-Yeou Lee, who has been teaching and supporting me since my elementary school years.

Finally, I would like to thank my loving and caring wife Michelle for her patience, support, and encouragement. To whom, I dedicate this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

An inference system is able to derive conclusions from a set of *axioms*. An axiom is a true statement about the properties of the underlying theory that we are interested in. Other true assertions which can be inferred from the truth of the axioms are called *theorems*. A *proof* of a theorem is an argument which establishes that the theorem is true. To establish that one assertion follows from another, we must use only principles of reasoning which we accept as valid; these principles are called *inference rules* [SM77].

An inference system for a specified theory is *complete* if all valid statements in the underlying theory can be derived from the given axioms using the inference rules. A complete system guarantees the existence of a proof for a valid statement, but may not provide any information about the construction of the proof. Constructing proofs is a complex information processing task. The human's activity involved in this task is still a mystery. Moreover, constructing detailed proofs by humans is tedious and error prone.

Modern computers offer new ways to perform the task of proof construction because they do inferences rapidly and accurately. Part of the mystery in human theorem proving activity is that humans can often find the proof efficiently without doing too much search. Computers may not be so good as humans in applying various tricks, but they can do thorough search mechanically much faster without making any errors. Usually, we have to instruct computers how to search for a proof. A search strategy is complete if it can obtain a proof for all theorems. Some complete search strategies that are used most often are breadth-first search, best-first search, and depth-first iterative deepening. A useful automated reasoning system usually consists of a complete inference system and complete search strategies.

Although computers are very good at doing inference very fast, we still have efficiency problems. Finding proofs is a combinatorial problem. Trying all possibilities thoroughly without appropriate control is often beyond the reach of any modern computers, either in time or space. Therefore, the efficiency of an automated reasoning system must be seriously considered. To improve the efficiency of an automated reasoning system, we can design an inference system which is stronger or more restrictive. We can also improve the search strategy to do more restricted search.

The possible applications of an automated reasoning system include expert systems, planning, common sense reasoning, program verification, hardware verification, logic programming, and aids to human mathematicians. Indeed, automated reasoning systems can conceivably be used anywhere inference is used.

This thesis is concerned with the design of an efficient automated reasoning system in first-order logic. The inference system consists of a restricted instance generator and a contradiction detector. It gains efficiency from the avoidance of duplication of instances of clauses. It turns out that this inference system is complete. The inference system fits naturally to a breadth-first search. This search is improved and is able to simulate best-first search.

The automated reasoning system can be used either as a theorem prover or as a problem solver. As a theorem prover, it tries to derive the conclusion, the empty formula, from a set of axioms and the negation of the theorem. As a problem solver, it obtains conclusions which logically follow from a set of constraints.

1.2 Terminology

We will define the terminology in this thesis. Our work is based on Skolemized first order formulas (clause form). We use lower case letters, possibly subscripted, for individual constants, function constants, and predicate constants, and upper case letters, possibly subscripted, for (individual) variables. Each predicate constant and function constant has an *arity*, which is a non-negative integer telling how many arguments it takes. An individual constant may be viewed as a 0-arity function. A *term* is a well formed expression composed of individual constants, variables, and function constants. For example, a, f(X), and g(f(Y), b, f(b)) are terms, given that a and b are individual constants, f and g are function constants, and X and Y are variables. An *atom* is a predicate constant followed by a list of terms. For example, if p is a predicate constant then p(a, f(X)) is an atom. A *literal* is an atom or an atom preceded by a negation sign. For example, p(a, f(X)) and ¬p(a, f(X)) are literals. We represent a negation sign by either "not" or "¬". We also treat p and the negation of the complement of p as identical for any literal p. A literal is called *positive* if it is an atom, *negative* if it is the negation of an atom. A literal and its negation are called *complementary literals*. A *clause* is a disjunction of literals, expressed as a set. For example, clause {p(a, f(X)), ¬q(g(f(Y), b, f(b)))} represents a disjunction of literals p(a, f(X)) and ¬q(g(f(Y), b, f(b))). The empty clause denotes the logical constant FALSE. A *set of clauses* represents the conjunction of the clauses in the set. A *positive clause* is a clause containing only positive literals; a *negative clause* is a clause containing only negative literals. A clause containing only one literal is called a *unit clause*. A term (literal, clause) is a *ground* term (literal, clause) if it contains no variables.

A *substitution* is a mapping Θ from variables to terms, such that for only finitely many variables X do we have Θ(X) ≠ X. We represent a substitution by a finite set $\{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$, indicating that $\Theta(X_i)$ is t_i and Θ(X) is X if X is a variable distinct from the X_i. If t is a term (literal, clause), then by tΘ we indicate the result of simultaneously replacing all variables X_i in t by t_i. For example, suppose t is p(X, f(a, Y)) and Θ is {X ← b, Y ← g(Z)}. Then tΘ is p(b, f(a, g(Z))). We say a term u is an *instance* of a term t if there is a substitution Θ such that u is tΘ, and similarly for literals and clauses. Two terms (literals, clauses) are *variants* if they are instances of each other. For example, p(X, f(a, Y)) and p(W, f(a, V)) are variants. A *unifier* of two terms (literals, clauses) L and M is a substitution Θ such that LΘ and MΘ are identical. A *most general unifier* Θ of two terms (literals, clauses) L and M is a unifier of L and M such that for any other unifier Γ of L and M there is a substitution Φ such that LΘΦ is identical to LΓ (or MΘΦ is identical to MΓ). For example, the most general unifier for p(a, X, f(g(Y))) and p(Z, f(Z), f(U)) is {Z ← a, X ← f(a), U ← g(Y)}. An *interpretation* of a set S of clauses consists of a domain D, (nonempty set) together with assignments I_v and I_c of meanings to variables and constants. For a variable x, I_v assigns an element of D to x, for an individual constant a, I_c assigns an element of D to a, for a function constant f, I_c assigns a function from D^n to D, where n is the arity of f, and for a predicate constant P, I_c assigns a function from D^n to {TRUE, FALSE}, where n is the arity of P. For example, suppose we have a set S of clauses {p(X)} and {¬p(Y)}. An interpretation of S may consists of a domain D with only one element a, together with I_v assigning X and Y to a and I_c assigning p(a) to FALSE. A set S of clauses is *satisfiable* if there is one interpretation of S which makes all clauses contained in S true simultaneously. Such interpretation is called a *model* for S. For example, the set {{a,b}, {¬a, ¬b}} is satisfiable since the interpretation, a assigned TRUE and b assigned FALSE, is a model for this set. A set S of clauses is *unsatisfiable* if there is no interpretation of S which makes all clauses contained in S true simultaneously. For example, the set {{a,b}, {¬a}, {¬b}} is unsatisfiable.

1.3 The Theorem Proving Problem

One approach to theorem proving in first order logic is to reduce the theorem proving problem to the following problem, by Herbrand's theorem: Given a set S of clauses, to determine if there is a set T of ground clauses such that for each clause D in T there is a clause C in S such that D is an instance of C, and such that T is propositionally unsatisfiable. We call this the *ground instantiation problem.* For example, given S = {{p(a)}, {¬p(X), q(f(X))}, {¬q(f(Y))}}, the set T = {{p(a)}, {¬p(a), q(f(a))}, {¬q(f(a))}} is as specified. This set T is propositionally unsatisfiable because p(a) and (¬p(a) ∨ q(f(a))) imply q(f(a)). This contradicts ¬q(f(a)). Note that if the ground instantiation problem can be solved, then S, with free variables regarded as universally quantified, is unsatisfiable.

This approach allows the search for the proof of any theorem in first order logic to be reduced to the ground instantiation problem by negating the theorem and then applying a well known process called Skolemization [CL73][Lov78]. For example, consider the theorem that (∀X p(X)) implies (∀X (p(X) ∨ q(X))). This theorem is first negated, giving (∀X p(X)) and ¬(∀X (p(X) ∨ q(X))), which is equivalent to (∀X p(X)) and (∃X (¬p(X) ∧ ¬q(X))). Replacing existential quantifiers by a new constant a, we obtain (∀X p(X)) and (¬p(a) ∧ ¬q(a)). Removing universal quantifiers, we have p(X) and (¬p(a) ∧ ¬q(a)). This is converted to the following set S of clauses: {{p(X)}, {¬p(a)}, {¬q(a)}}. There is the set T of ground instances {{p(a)}, {¬p(a)}} which is propositionally unsatisfiable. Therefore the original theorem (∀X p(X)) implies (∀X (p(X) ∨ q(X))) is valid. Note that if S contains the empty clause, then we can choose T to be the set containing the empty clause, solving the ground instantiation problem.

1.4 Motivation

Many theorem proving methods involve duplication of instances of clauses during the process of inference. This duplication can cause combinatorial problems, since a small number of formulas may combine in many different ways. One motivation of our work is to avoid such duplication. There are two kinds of duplication. The first we call *duplication by combination.* This appears in resolution, model elimination, the connection graph method, and other strategies. The same instances of a clause can contribute to many resolvents, for example. The second kind of duplication is *duplication by case analysis.* This appears in the linked conjunct procedure of [Dav63] as well as in the Davis-Putnam propositional calculus decision procedure [DaPu60]. The same instances of clauses may appear in more than one case of a case analysis, although in each case there may be no duplication. Some early provers used duplication by case analysis, but few if any clause form provers currently use it. Andrews' prover [MCA82] seems to be one of the few current provers that use duplication by case analysis.

We now make more precise how resolution and other methods involve the duplication of instances. Given clauses C and D, resolution computes a most general unifier Θ and then combines literals of CΘ and DΘ to obtain a resolvent E. Note that if C and CΘ are identical, then the same literals may appear in both C and the resolvent E. This is one kind of duplication of instances by combination. Another kind appears when a clause C resolves against many clauses D_1, D_2, ..., D_n using the substitutions Θ_1, Θ_2, ..., Θ_n. In this case, it may be that many of the instances CΘ_i are identical. Then the literals in CΘ_i may appear in many different resolvents. This is another kind of duplication by combination. Duplication by combination occurs for similar reasons in model elimination and the connection graph method. The linked conjunct procedure of [Dav63] involves duplication by case analysis. Given a set S of clauses, a linked conjunct is obtained by matching the literals in clauses in S in different ways. Each matching has to be looked at separately; each such matching is another case to be considered. The same instances will appear in many such matchings; thus we have duplication by case analysis. Even if S is propositional, there may be many cases to consider and the method may be slow. Since a given literal may match many other literals, there may be many cases to consider for each literal. Similarly, the Davis-Putnam procedure involves case analysis on the truth values of atoms appearing in S. If an atom A appears in S, the cases A = TRUE and A = FALSE are considered separately. Many clauses and literals will appear in both cases, so we have duplication by case analysis.

A related problem with connection graph methods is the need to store the connections. A connection is a pair (L, M) of literals such that L and the complement of M are unifiable. If there are n literals, there may be up to n^2 connections between them. This can be a large number of connections, and may require excessive storage and time to process. The mating strategy of [And81] is similar to the connection graph method and has similar properties. In fact, the mating strategy is much like the linked conjunct procedure of [Dav63]. Since it explicitly constructs links, or connections, between literals, the number of links with a particular literal may be much larger than two. Mating appears to have duplication by case analysis rather than duplication by combination, since matings are constructed and considered one by one. However, since the number of links per literal may be much larger than two, the number of cases may be much larger than in the Davis-Putnam procedure for propositional calculus, it appears. Even the modified problem reduction format prover [Pla88] has duplication by combination on non Horn problems.

Another motivation of our work is to justify the claim that a theorem proving method may benefit from applying the low-level heuristics a human mathematician uses to obtain a proof. By low-level heuristics, we mean such methods as preferring terms that are small, or contain a small number of variables, and so on. Some of the methods used in theorem proving, such as variable elimination [Hin88] and the complexity measure of Wang [WB87], successful though they are, seem to be mysterious, since it is unlikely that a human would use such methods. We believe that a theorem prover which works well with these low-level heuristics provides a better rational reconstruction of human theorem proving activity.

This dissertation

1. presents a novel proving method which avoids duplication of instances of clauses.
2. presents a fast propositional decision procedure which is optimal in some weak sense.
3. provides a facility to do guided search.
4. provides a mechanism which allows controlled use of memory and has the same effect as best-first search.
5. proposes a rational reconstruction of human theorem proving activity.
6. describes a friendly interface for the user.
7. describes how our system is used as a problem solver.
8. presents a new paradigm for proving temporal logic theorems.

1.5 Our Method

Our thesis: *A theorem prover may gain efficiency by avoiding duplication of instances of clauses. Such avoidance can be achieved by reducing first order logic to propositional calculus and then applying a propositional calculus decision procedure.*

The automated reasoning system, CLIN, is the realization of the thesis. It is an instance-based reasoning system. The system is based on the hyper-linking strategy prover. The hyper-linking strategy prover consists of two parts. The first part is called the clause generator; it generates selected instances of the input clauses by hyper-linking. The second part is called the contradiction detector; it tries to detect a contradiction from the ground set of the clauses stored in the database using a propositional calculus decision procedure. Hyper-linking does not combine parts of separate clauses, so the combinatorial problem caused from duplication by combination is avoided. A little duplication by case analysis occurs in the propositional calculus decision procedure, but this duplication is negligible most of the time.

CLIN incorporates some refinements to improve the performance of the hyper-linking strategy prover. These refinements include support sets, predicate replacement, unit simplification, small proof checking, and sliding priority. Figure 1.1 shows the major components of the system.

CLIN has some analogies to human problem solving methods. The clause generator may be considered as creative and the contradiction detector as analytic. The system is also related to the associative network model of artificial intelligence, in which nodes are connected by links and activations can spread from node to node along the links. For us, the nodes are assertions, the links are unifications between parts of formulas, and the activations are substitutions that propagate from node to node. There is an obvious if possibly superficial resemblance to neural networks. The similarities with semantic networks in artificial intelligence, and with connection graph methods in theorem proving, are closer.

CLIN has been implemented in Prolog. Its effectiveness has been shown by comparing it with other major provers, such as OTTER[McC89], "sprfn"[Pla88] , and PTTP[Sti88]. CLIN performs better than the other provers for propositional and near-propositional problems. For other problems, CLIN can prove more of them than the other provers. This shows that duplication of instances of clauses is a real problem for theorem proving, and that an instance-based approach which is able to eliminate such duplication may be superior to other approaches.

1.6 Related Work

Some of the early work in theorem proving was similar in spirit to ours, such as the linked conjunct procedure of [Dav63]. Even the methods of [Gil60][PPV60] have similarities to ours. However, they also have significant differences. Gilmore's procedure is not based on unification. This means that the number of instances generated may be very large. The Prawitz procedure and the linked conjunct procedure are based on something like unification. However, they both consider a large number of possibilities for which literals to unify, resulting in many cases to consider. The paper [Dav63] (linked conjunct procedure) also doesn't explicitly give a completeness proof or a method of generating instances. Our method is based on unification but does not generate such a large number of cases. In this our method is similar to resolution. This may explain the effectiveness of resolution (and our method) as compared to methods used prior to 1965. Methods developed since 1965, such as model elimination and connection graph methods, also tend to be based on unification without requiring a large number of cases to be considered. Since 1965, much work has been done on the resolution method of Robinson [Rob65]. Unlike our method, resolution combines parts of different clauses. This has advantages and disadvantages. The

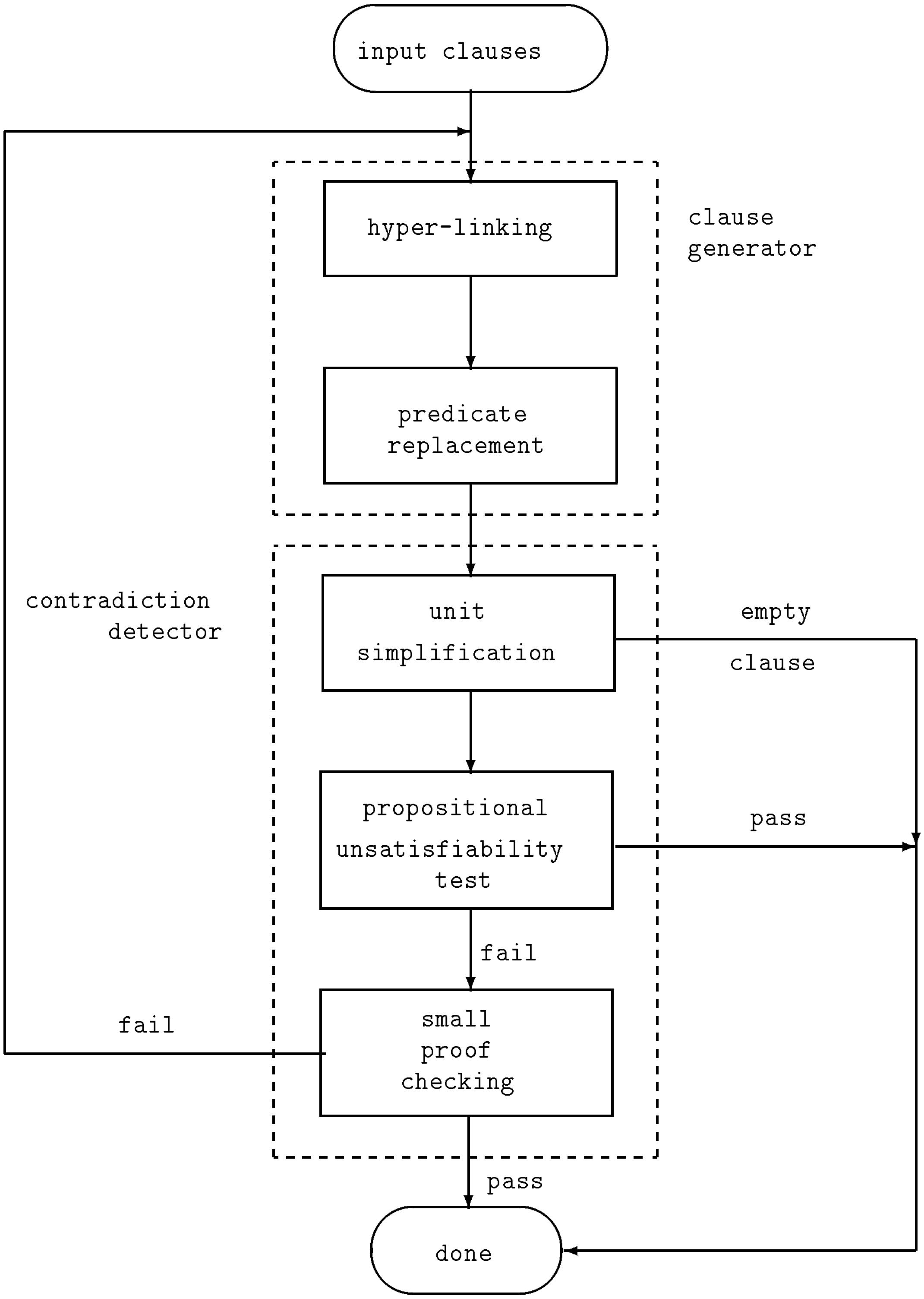

Figure 1.1: Major Components in CLIN

connection graph methods of [Bib82] are close to ours, but even these combine literals from different clauses. We feel that instance-based approaches deserve more attention, and our experience with our implementation confirms this belief. We propose that instance-based provers can be competitive or superior to resolution provers using equivalent technology.

1.7 Organization of the Thesis

In this section, we give an overview of the remainder of this dissertation. We should mention here that we test the performance of CLIN on a test set which includes 120 problems. The sources of these problems are mentioned in Appendix A. Throughout the dissertation, we present experimental results or give examples for illustration. These problems are mainly drawn from the test set. All the times related to our work are obtained by running ALS-Prolog (Version 0.60) of Applied Logic System, Inc. on a SUN3/60M workstation with 12 MB memory.

In Chapter 2, we present the hyper-linking strategy prover. We describe how we obtain instances selectively by hyper-linking. A Davis-Putnam like propositional calculus prover using a technique similar to dependency-directed backtracking is presented. We also show that the hyper-linking strategy is complete.

In Chapter 3, we present some refinements to the hyper-linking strategy prover. Unit simplification simplifies the clauses retained and makes proofs simpler. Support sets and relevance criterion enable the prover to concentrate on related clauses so as to reduce the search space. Small proof checking is used to find an early proof.

In Chapter 4, we present a facility, called predicate replacement, by which the user can guide CLIN to search for a proof. The user may convey his or her intuition or knowledge about how to prove the underlying problem to the system through replace rules. The rules then direct the system to do guided search.

In Chapter 5, we present the sliding priority method. With the help of this method, CLIN has a better control over the use of memory and obtains the same effect as best-first search.

In Chapter 6, we present a top-level supervisor which serves as a friendly interface between CLIN and the user. The supervisor uses expert knowledge to find proofs for the user without intervention from the user. This encourages users who have little knowledge about theorem proving techniques to use our system.

In Chapter 7, we show the capability of proving hard theorems of our system. Replace rules are used to guide the system very efficiently.

In Chapter 8, we do a comparison between CLIN and some other major proving systems on some selected cases. This comparison serves as a demonstration of the effectiveness of our system, and also shows the significance of removing duplication of instances of clauses.

In Chapter 9, we discuss some applications of our system. We show how constraint satisfaction problems (CSPs) and logic puzzles are solved by CLIN. The elimination of duplication without constraint preprocessing is an advantage of our system. We also show how replace rules are used for proving propositional temporal logic theorems. We also present some ideas about potential applications to database design and axioms debugging.

In Chapter 10, we give a summary and present some possible extensions to our system.

The thesis has three appendices. We have run CLIN with various options on the test set. Some results are listed in Appendix A. Appendix B lists the input files of some test problems. Appendix C includes a brief user manual for CLIN.

1.8 Related Publications

This thesis is partially based on [PL90]. Parts of chapters 2, 4, and 9 appear in preliminary forms in [LP89][LP90a][LP90b].

Chapter 2

The Hyper-Linking Strategy Prover

The philosophy of the hyper-linking strategy prover is never to combine literals from two clauses. The prover consists of a two stage process. The first stage, hyper-linking, generates the selected instances of the given clauses. The second stage, propositional unsatisfiability test, tests for unsatisfiability on the ground set of clauses retained in the database using a propositional calculus prover. We completely avoid duplication by combination in the hyper-linking stage. A little duplication by case analysis occurs in the propositional unsatisfiability test. However, there are at most two cases, TRUE and FALSE, to be considered per literal. It turns out that this propositional calculus prover is so efficient that such duplication is negligible most of the time, compared to the overall performance of the hyper-linking strategy prover. The two stages are performed iteratively until the input problem is solved or until the time limit is exceeded. Figure 2.1 shows the block diagram of the hyper-linking strategy prover.

Several prior implementations have been tried. The idea first appeared in [Pla80] and the first implementation was in [JP84]. An extension of this to a unification-based prover was attempted, without much success [PM88]. A key to our current success seems to be that we do not store any links between clauses, which sets our method apart from the connection graph resolution of [Bib82] which relies heavily on such links. Also, another key to our success is the idea of hyper-linking, that is, considering all the literals in a clause at the same time, rather than one by one as in the previous attempts. The use of support strategies, explained later, is also important. Finally, in previous strategies [JP84][PM88], a "fully linked subset," was computed which was essentially the same as the linked conjunct of [Dav63]. However, we found that it takes too much time to compute this subset, and instead we consider the set of all clauses, which may be much larger.

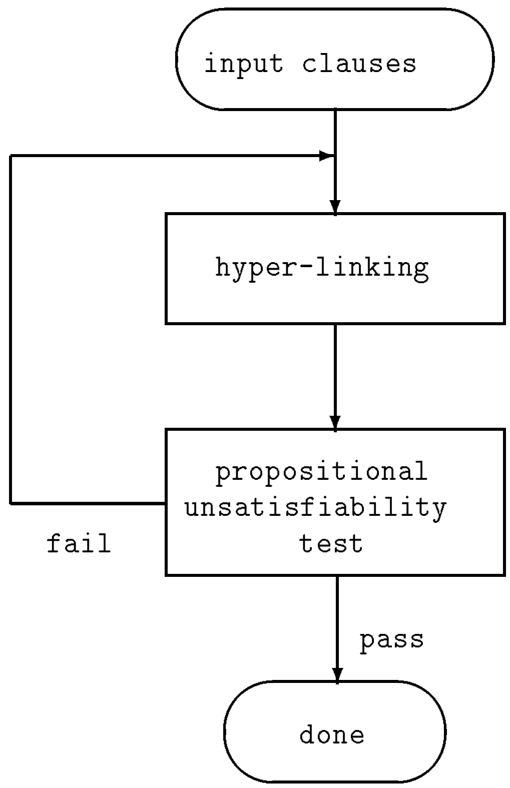

Figure 2.1: The hyper-linking strategy prover

Another advantage of the hyper-linking strategy prover is that clauses are kept as lists of literals, as in resolution. All literals are on an equal footing. This contrasts with the modified problem reduction format of [Pla88], in which contrapositives of clauses are considered separately. The conceptual simplicity of having clauses as lists has enabled us to add some refinements to the prover without having to deal with the complexities of contrapositives. The hyper-linking strategy prover also makes use of links between clauses which are reminiscent of associations between facts in human thought.

2.1 Hyper-Linking

2.1.1 Descriptions

Given a set S of clauses, define a *link* in S to be a pair (L, M) of literals such that both L and M appear in clauses in S and such that L and the complement of M are unifiable. We may have to rename the variables in L or M so that L and M have no common variables. We treat (L, M) and (M,L) as identical. Note that this is essentially the same as a connection in Kowalski's proving procedure for Horn clauses [Kow75] or in the connection graph resolution [Bib82], except that we don't explicitly store the links, so that they cannot be deleted as they are in Kowalski's proving procedure or the connection graph resolution.

Definition. Suppose S is a set of clauses. The *connection graph* of S, denoted (S, E), is the graph whose *nodes* are the clauses of S, and which has an *edge* labeled (L_1, L_2) in E between C_1 and C_2 if (L_1, L_2) is a link and C_1 and C_2 contain L_1 and L_2 respectively.

Definition. The *initial connection graph* (S, E) is a connection graph of S if S is a set of input clauses.

Definition. Suppose S is a set of clauses. Gr(S) is defined to be {CΦ: C $\in$ S}, where Φ is a substitution replacing all variables of all clauses in S by a new constant symbol \$. We call Gr(S) a *ground set* of S.

For example, if S is {{¬p(X)}, {q(X,Y)}} then Gr(S) is {{¬p(\$)}, {q(\$,\$)}}.

Definition. Suppose (S, E) is a connection graph. If C = {L_1, ..., L_m} is a clause then a *hyper-link* of C in E is a set {(L_1, M_1), ..., (L_m, M_m)} of links in E such that there exists a substitution Θ such that $L_i\Theta$ and $M_i\Theta$ are complementary for all i, $1 \leq i \leq m$. We may have to rename the variables in L_i or M_i so that L_i and M_i have no common variables. A most general such Θ is called the *substitution* of the hyper-link and CΘ for this Θ is called the *instance* of the hyper-link. We call this instance generation a *hyper-link operation*. We call C the *nucleus* of the hyper-link and we call the M_i (or clauses D_i containing M_i) the

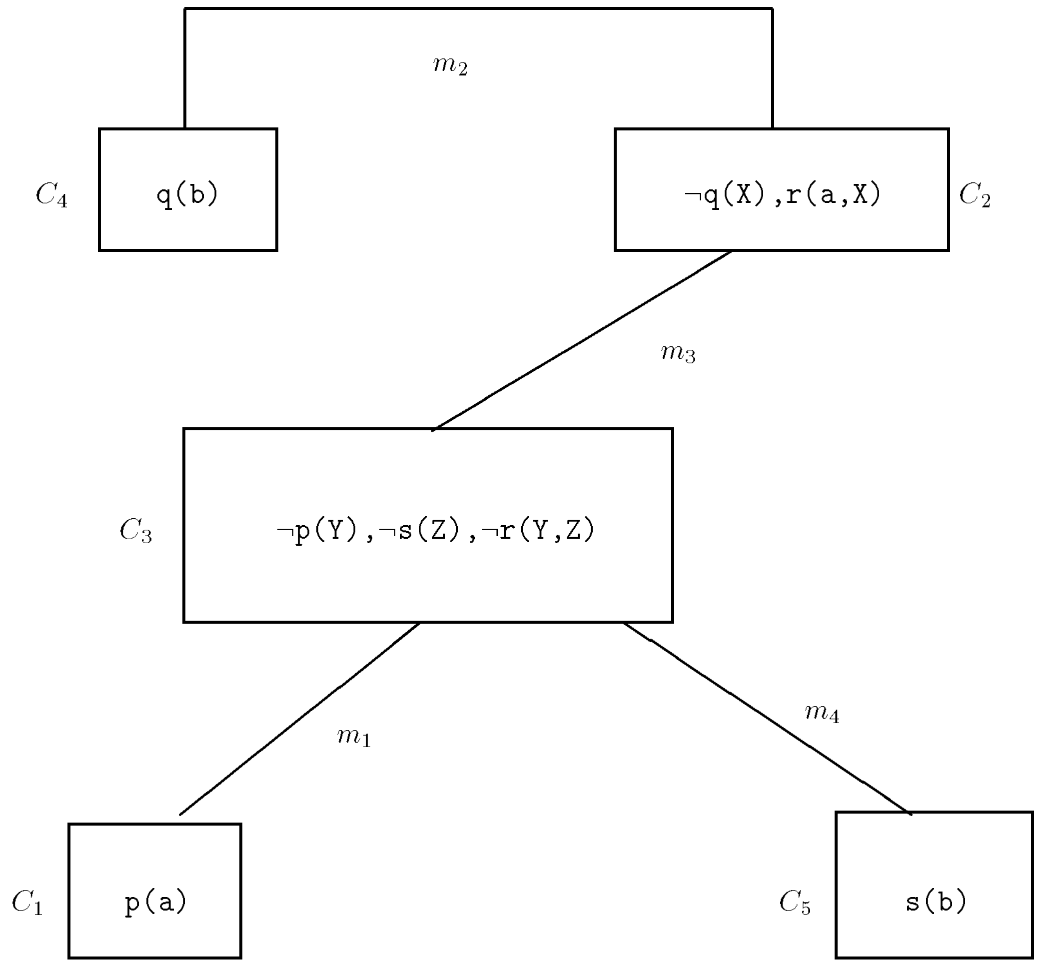

Figure 2.2: A connection graph

electrons of the hyper-link.

Suppose we have the following clauses: {p(a)}, {¬q(X), r(a, X)}, {¬p(Y), ¬s(Z), ¬r(Y, Z)}, {q(b)}, and {s(b)}. Let's call them C_1 through C_5 respectively. The connection graph for these clauses is shown in Figure 2.2. Each node represents one clause. There are four links (edges). m_1 is (p(a), ¬p(Y)), m_2 is (q(b), ¬q(X)), m_3 is (r(a, X), ¬r(Y, Z)), and m_4 is (¬s(Z), s(b)). There are five hyper-links. The hyper-link with nucleus C_1 contains link m_1. The hyper-link with nucleus C_2 contains links m_2 and m_3. The hyper-link with nucleus C_3 contains links m_1, m_3, and m_4. The hyper-link with nucleus C_4 contains link m_2. The hyper-link with nucleus C_5 contains link m_4.

Given a nucleus C = $\{L_1, \ldots, L_m\}$ of a hyper-link $\{(L_1, M_1), \ldots, (L_m, M_m)\}$, the computation of an instance is as follows: Compute the most general unifier θ_i such that

$$L_i\theta_1 \cdots \theta_{i-1}\theta_i = (\neg M_i)\theta_i$$

for $1 \leq i \leq m$. The substitution Θ of the hyper-link then is $\theta_1 \cdots \theta_m$. The instance corresponding to this substitution is CΘ.

The hyper-linking stage works this way: Suppose (S, E) is a connection graph. For each clause C in S, all hyper-links of C in E are computed, together with their substitutions Θ. Then, the set of all such instances CΘ and consequent links are added to (S, E), for all clauses C in S and all substitutions Θ of their hyper-links. This results in a new graph (S′, E′). We call this expansion of (S, E) to (S′, E′) a *round of hyper-linking* or a *round*. For example, if {¬p(X), q(X)} is in S and literals p(a) and ¬q(a) appear in clauses in S, then the instance {¬p(a), q(a)} is generated. If {¬p(X), q(f(X))} is in S and the literals p(a) and ¬q(X) appear in clauses in S then the instance {¬p(a), q(f(a))} is generated.

After a round of hyper-linking, the hyper-linking strategy prover performs propositional unsatisfiability testing. Suppose S′ is the set of clauses retained after a round. We then test Gr(S′) for propositional unsatisfiability using a propositional calculus prover. It turns out that this is complete. That is, if an input set S is unsatisfiable, then after some number of rounds of hyper-linking, Gr(S″) will be propositionally unsatisfiable where S″ is the resulting set of clauses.

We now give an example to show how the hyper-linking strategy prover works. Suppose S is {{p(a)}, {¬p(X), p(f(X))}, {¬p(f(f(a)))}}. We call these clauses C_1, C_2, and C_3, respectively. There are three hyper-links. First, we consider the hyper-link of (¬p(X), p(a)), (p(f(X)), ¬p(X)) with nucleus C_2. The instance of this hyper-link is {¬p(a), p(f(a))}. Another hyper-link contains (¬p(X), p(f(X))) and (p(f(X)), ¬p(X)) and the instance of this hyper-link with nucleus C_2 is {¬p(f(X)), p(f(f(X)))}. The third hyper-link contains (¬p(X), p(f(X))) and (p(f(X)), ¬p(f(f(a)))) and the instance of this hyper-link with nucleus C_2 is {¬p(f(a)), p(f(f(a)))}. This completes the first round of hyper-linking. At this point the set S′ of clauses is {{p(a)}, {¬p(X), p(f(X))}, {¬p(f(f(a)))}, {¬p(a), p(f(a))}, {¬p(f(X)), p(f(f(X)))}, {¬p(f(a)), p(f(f(a)))}}. We ground all clauses, replacing all variables with \$, obtaining the set Gr(S′) of ground instances {{p(a)}, {¬p(\$), p(f(\$))}, {¬p(f(f(a)))}, {¬p(a), p(f(a))}, {¬p(f(\$)), p(f(f(\$)))}, {¬p(f(a)), p(f(f(a)))}}. This set is found to be propositionally unsatisfiable. To see this, consider the subset {{p(a)}, {¬p(f(f(a)))}, {¬p(a), p(f(a))}, {¬p(f(a)), p(f(f(a)))}}. The first and the third clauses imply p(f(a)). This, together with the fourth clause, implies p(f(f(a))) which contradicts the second clause. Note that if we had grounded the set S of input clauses, we would get {{p(a)}, {¬p(\$), p(f(\$))}, {¬p(f(f(a)))}}. This set is satisfiable. The extra instances generated by hyper-linking were needed for the proof.

2.1.2 The Computation of the Ground Set

For a connection graph (S, E), the ground set Gr(F) on which propositional unsatisfiability test is performed can be obtained in two ways. F can be either S itself or a fully linked subset of S. We allow users to choose either way. The former is the default setting. The set Gr(S) can be computed much more efficiently, but it may be large. For the application to a large database, a fully linked subset of S described below may be more practical.

Definition. A link (L, M) is a *variant* link if L and the complement of M are variants.

Definition. A subset F of S in a connection graph (S, E) is said to be *fully linked* if for all clauses C_1 in F and for all literals L_1 in C_1 there is a clause C_2 in F and a literal L_2 in C_2 such that (L_1, L_2) is a variant link in E.

For example, consider a set S of the following clauses

1. {p(a)}
2. {¬p(X), p(f(X))}
3. {¬p(f(f(a)))}
4. {¬p(a), p(f(a))},
5. {¬p(f(X)), p(f(f(X)))}
6. {¬p(f(a)), p(f(f(a)))}

The subset F containing clauses 1, 3, 4, and 6 is a fully linked subset of S. In fact, F is the only fully linked subset of S in this case.

2.2 The Propositional Calculus Prover

After a round of hyper-linking, the set S of clauses available so far is given to the propositional calculus prover for an unsatisfiability test. First, all variables in S are replaced by a new individual constant \$ to obtain a set Gr(S) of ground clauses. Thus, the clause {¬p(X, Y), q(Y)} would be replaced by {¬p(\$, \$), q(\$)}, for example. Then Gr(S) is tested for unsatisfiability using the propositional calculus prover. If Gr(S) is unsatisfiable, then the ground instantiation problem has been solved and we know S is unsatisfiable. Otherwise, more rounds of hyper-linking and so on, are done.

The propositional calculus prover is a modification of the Davis-Putnam procedure [DaPu60]. It uses a technique similar to dependency-directed backtracking [dK86a][dK89] and McAllester's RUP [McA80][McA82] in the area of Artificial Intelligence. One disadvantage of the Davis-Putnam procedure is that it carries out two cases in a case analysis all the time, which results in unnecessary duplication by case analysis. The motivation of our propositional calculus prover is to remove such unnecessary duplication. Our propositional calculus prover always performs as well as the Davis-Putnam procedure on the problems tested, and for some problems it performs much better.

Although the satisfiability problem is NP-complete [Coo71], we have found that the time taken by the propositional calculus prover is usually much smaller than the time taken by the other steps of the system. Therefore, even if P = NP, this would not help most of the time. It is only for a few problems such as the "pigeonhole problems" [Pel86] that the propositional prover takes most of the time. This enables us to do the unsatisfiability test on the ground set of all the clauses instead of on the ground set of a fully linked subset of these clauses. It is surprising that such a simple method works so well. Also, since the only duplication by case analysis in the hyper-linking strategy prover occurs in the propositional calculus prover, the speed of the propositional calculus prover shows that this duplication is negligible most of the time.

2.2.1 Simplification Rules

We will first describe the Davis-Putnam procedure, and then our method. The following simplification rules are used by both methods:

1. A clause containing ¬FALSE can be deleted:

$$\frac{S_1 \wedge (\mathrm{A} \vee \neg \mathrm{FALSE})}{S_1}$$

2. A clause containing TRUE can be deleted:

$$\frac{S_1 \wedge (\mathrm{A} \vee \mathrm{TRUE})}{S_1}$$

3. ¬TRUE may be deleted from a clause containing ¬TRUE:

$$\frac{S_1 \wedge (\mathrm{A} \vee \neg \mathrm{TRUE})}{S_1 \wedge \mathrm{A}}$$

4. FALSE may be deleted from a clause containing FALSE:

$$\frac{S_1 \wedge (\mathrm{A} \vee \mathrm{FALSE})}{S_1 \wedge \mathrm{A}}$$

5. A clause containing both L and ¬L can be deleted:

$$\frac{S_1 \wedge (\mathrm{A} \vee \mathrm{L} \vee \neg \mathrm{L})}{S_1}$$

6. TRUE may be deleted from a set of clauses containing TRUE:

$$\frac{S_1 \wedge \mathrm{TRUE}}{S_1}$$

7. A set of clauses containing FALSE may be replaced by FALSE:

$$\frac{S_1 \wedge \mathrm{FALSE}}{\mathrm{FALSE}}$$

Thus, {{TRUE, p}, {FALSE, q}} simplifies to {{q}}.

2.2.2 The Davis-Putnam Procedure

The Davis-Putnam procedure basically performs unit simplification, pure literal deletion, and case analysis iteratively. We represent the Davis-Putnam procedure by a procedure DP(S) which takes as input a set of propositional clauses and outputs TRUE, if S is satisfiable, and FALSE otherwise. This procedure is fairly simple, and may be described as follows:

procedure DP(S)
    simplify S to S′ using simplification rules;

if S′ is TRUE or FALSE, return S′;
if S′ contains a unit clause C then % do unit simplification
  if C is {L} for positive literal L,
    replace L by TRUE in S′, and simplify S′ to S″ using simplification rules;
    return DP(S″);
  if C is {¬L} for positive literal L,
    replace L by FALSE in S′, and simplify S′ to S″ using simplification rules;
    return DP(S″);
if S′ contains a clause C containing a literal L and
  no other clause contains ¬L then % do pure literal deletion on L
  if L is positive, replace L by TRUE in S′,
    and simplify S′ to S″ using simplification rules;
    return DP(S″);
  if L is ¬M for some M,
    replace M by FALSE in S′, and simplify S′ to S″ using simplification rules;
    return DP(S″);
if none of the above holds, then % do case analysis
  pick a positive literal L in some clause C in S′;
  replace L by TRUE, and simplify S′ to S_1 using simplification rules;
  if DP(S_1) is TRUE then return TRUE
  else replace L by FALSE, and simplify S′ to S_2 using simplification rules;
      if DP(S_2) is TRUE, then return TRUE
      else return FALSE;
end DP.

If we draw a node to represent a case analysis on L and two edges out from this node to represent the subcases of L assigned TRUE and L assigned FALSE respectively, then we end up with a complete binary tree (with two edges out from a non-leaf node). We call this tree a *case tree*.

2.2.3 Our Method

Our propositional calculus prover basically performs unit simplification and case analysis iteratively. It is described as follows:

procedure PC(S)
  simplify S to S′ using simplification rules;
  If S′ is TRUE or FALSE, return S′;
  If S′ contains a unit clause C then % do unit simplification

```
      if C is {L} for positive literal L,
        replace L by TRUE in S′, and simplify S′ to S″ using simplification rules;
        return PC(S″);
      if C is {¬L} for positive literal L,
        replace L by FALSE in S′, and simplify S′ to S″ using simplification rules;
        return PC(S″);
    If none of the above holds, then
      if S′ does not contain a positive or a negative clause
      then return TRUE
      else let C be a smallest negative clause in S′;   % do case analysis
            pick a literal ¬L in C;
            replace L by FALSE, and simplify S′ to S1 using simplification rules;
            if PC(S1) is TRUE then return TRUE
            else if the assignment of FALSE to L was not
                    needed to show that S1 is unsatisfiable
                then return FALSE   % right cutoff
                else replace L by TRUE, and simplify S′ to S2 using simplification rules;
                    if PC(S2) is TRUE, then return TRUE
                    else if the assignment of TRUE to L was not
                            needed to show that S2 is unsatisfiable
                        then return FALSE   % left cutoff
                        else return FALSE;
end PC.
```

Our method is similar to the Davis-Putnam procedure except that there is no test for a pure literal, and the clause used for case analysis is chosen as a smallest negative clause, if one exists. However, the major difference is in the case analysis itself. If PC(S_1) is FALSE, we keep track of whether the assignment to L is used in this part of the proof. If not, we call this a *right cutoff* and return FALSE, since the same proof could be done if the other truth value were assigned to L. If the truth assignment to L was used in the first case, but in the second case, PC(S_2) is FALSE and the truth assignment to L is not used, we call this a *left cutoff*, and PC(S) returns FALSE. Right cutoffs correspond to cases in which our procedure does less case analysis than the Davis-Putnam procedure. Left cutoffs do not save work on the current case analysis, but may reduce the number of case analysis afterwards. We have observed a number of right cutoffs, especially on the pigeonhole problems. However, we have never observed a left cutoff.

Our method seems to be efficient. It always runs as fast as as the Davis-Putnam procedure on the problems tested, and for some problems it performs much better. We ran both methods on 76 problems which are ground sets of instances, obtained by hyper-linking,

no	problem	no. of input clauses	no. of prop. symbols	satisfiability	DP (sec)	PC (sec)
1	apabhp	112	88	sat	1.5	0.2
2	ex4t1	98	52	unsat	41.2	0.8
3	ex4t2	98	52	unsat	1.7	0.8
4	wos4	113	98	unsat	1.2	0.4
5	wos25	137	172	sat	2.5	0.7
6	wos33	88	120	sat	1.4	0.7
7	ls28	343	273	sat	13.1	8.6
8	ls108	173	215	unsat	4.6	0.7
9	ls112	354	425	unsat	18.9	2.4
10	latinsq	573	63	sat	4.1	2.5
	average	209	156		9.0	1.8

Table 2.1: Some sample problems on which PC runs faster than DP

of some first-order problems. Our procedure is significantly faster on ten of the problems, as shown in Table 2.1. Note that the test problems contain many unit clauses and pure literals. This favors the Davis-Putnam procedure. The advantage of our propositional calculus prover over the Davis-Putnam procedure is intuitively understandable since the case tree for our procedure does not necessarily have two edges out from a non-leaf node. Therefore a lot of work may be saved sometimes. We can construct a resolution proof of the empty clause from the run of our propositional calculus prover, as described in [Pla90].

The time taken by the propositional calculus prover is only 5.293 percent in average of the total time for the test problems listed in Appendix A. Since the only duplication by case analysis in the hyper-linking strategy prover occurs in the propositional calculus prover, the speed of the propositional calculus prover shows that this duplication is negligible most of the time. Therefore, the hyper-linking strategy prover is almost a duplication-free prover.

2.2.4 Features of Our Method

In this section, we present some features about our propositional calculus prover. First of all, we show that we can detect the input set of clauses satisfiable earlier than the Davis-Putnam procedure. Whenever the set being considered does not contain a positive or a negative clause, we conclude that the input set of clauses is satisfiable. Notice that the Davis-Putnam procedure can't conclude that the input set of clauses is satisfiable until an empty set is derived.

Lemma 2.1 If a set S of clauses doesn't contain a negative clause or a positive clause, then S is satisfiable.

Proof. Suppose P is the set of proposition symbols that appear in S. Assume S does not contain a negative clause. Then assigning every symbol in P TRUE would make S true since every clause in S has a positive literal. Therefore, P itself is a model for S. Similarly, if S does not contain a positive clause, then assigning every symbol in P FALSE would satisfy every clause in S.

Theorem 2.2 If the set to be considered by case analysis does not contain a positive or a negative clause, then the input set of clauses to the propositional calculus prover is satisfiable.

Proof. Suppose at some node N_i in the case tree, we find that the set S′ being considered does not contain a negative or a positive clause. From Lemma 2.1, we know that there is a model M′ for S′. Apparently, any literal in M′ is different from the literals attached to the edges along the path from the root node to N_i. Therefore, the union of M′ and these literals constitutes a model for the input set of clauses to the propositional calculus prover.

Another feature of our propositional calculus prover is that one subcase in a case analysis can be saved if right cutoff occurs. If the proof of the subcase with L assigned TRUE did not use L, then the proof also holds with L assigned FALSE. Therefore, the work on proving the subcase with L assigned FALSE is saved. This saving also implies that our propositional calculus prover results in less duplication by case analysis than the Davis-Putnam procedure.

One interesting point we would like to mention here. We have never observed a left cutoff in any test problems. If a left cutoff does not occur, then the work done for all cases is needed in the whole proof tree for the input set of clauses. In this case, our procedure runs in time polynomial in the size of its input and in the size of the proof it finds.

Finally, we don't need a special procedure to carry out pure literal deletion. Pure literal deletion is essential to the Davis-Putnam procedure. Suppose L is a pure literal and is not deleted. The Davis-Putnam procedure may choose L to do case analysis. Then both L assigned TRUE and L assigned FALSE are to be considered, which wastes time. Case analysis in our propositional calculus prover carries out pure literal deletion automatically. Suppose a pure literal p (p is negative) is picked in a case analysis. All the clauses containing occurrences of p in the set being considered are deleted. The remaining clauses do not contain occurrences of ¬p since p is a pure literal. So the proof of the subcase with p does not use p and we have a right cutoff. Therefore we don't need to try the other subcase. This essentially achieves the same effect done by pure literal deletion on p. However, we don't take any action on p if p is positive since p can't be picked by case analysis.

2.2.5 Dependency Rules

The biggest advantage of our propositional calculus prover is that we don't need to do the other subcase if a right cutoff occurs. How do we detect a right cutoff, or even a left cutoff? The idea is as follows. We keep track of dependency information along with each clause in a set. Whenever we find an empty clause in a subcase with L assigned TRUE, we check whether the dependency set of the empty clause contains L. If L is not contained in the set, then L is not used in this subcase and we have a right cutoff. Similarly, we can detect a left cutoff.

Let A:B represent a clause B with *dependency set* A. This means that A was used in the proof of B. Initially we have {}:C for non-unit clauses C and $\{l\}$:l for unit clauses l. We have the following dependency rules:

1. For unit simplification, we have the following rules with l a literal:

$$\frac{S_1 \wedge [D_1\text{:}(\mathrm{C} \vee \neg l)],\ [D_2\text{:}l] \in S_1}{S_1 \wedge [(D_1 \cup D_2)\text{:}\mathrm{C}]}$$

$$\frac{S_1 \wedge [D_1\text{:}(\mathrm{C} \vee l],\ [D_2\text{:}l] \in S_1}{S_1}$$

2. For case analysis, when doing case analysis on p, first we assume p is true. This is like adding {p}:p to S. When assuming p is false, we add {¬p}:¬p to S.

3. The dependency set of a case analysis is that of the first subcase if right cutoff occurs, or that of the other subcase if left cutoff occurs, or the union of dependency sets, with the two assumptions deleted, of the subcases if neither cutoff occurs. If an empty clause is derived in a subcase, then the dependency set of the subcase is the dependency set of the empty clause.

4. To see if a proof depends on a case analysis on p, we check if p appears in the dependency set of this case analysis.

For a case analysis on p, it is a right cutoff if p does not appear in the dependency set of the subcase with p assigned TRUE. It is a left cutoff if p appears in the dependency set of the subcase with p assigned TRUE, but ¬p does not appear in the dependency set of the other subcase with p assigned FALSE.

2.2.6 An Example

We give an example here to illustrate our idea.

Example 2.1 Consider S of the following clauses:

- {¬p, q}
- {¬p, ¬q}
- {p, q}
- {a, ¬b}
- {¬a, ¬b}
- {a, b}
- {¬a, b}

Initially, every clause has an empty dependency set. Suppose we choose ¬p in {¬p, ¬q} to do case analysis. Adding {¬p}:{¬p} to S and doing unit simplification on this, we have S_1:

- {¬p}:{q}
- {}:{a, ¬b}
- {}:{¬a, ¬b}
- {}:{a, b}
- {}:{¬a, b}

Applying unit simplification on {¬p}:{q}, S_1 becomes S_2:

- {}:{a, ¬b}
- {}:{¬a, ¬b}
- {}:{a, b}
- {}:{¬a, b}

Suppose we choose ¬a in {¬a, ¬b} to do case analysis. We try the subcase with ¬a assigned TRUE first. Adding {¬a}:{¬a} to S_2 and doing unit simplification on this, we have S_3:

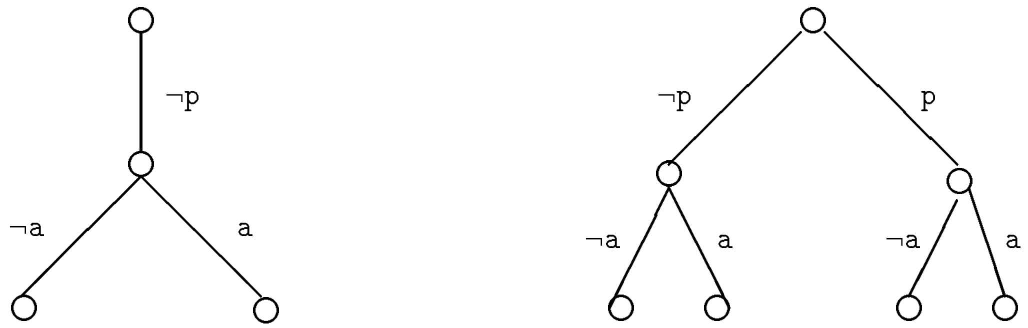

Figure 2.3: Case trees of our propositional calculus prover and the Davis-Putnam procedure

- {¬a}:{¬b}
- {¬a}:{b}

Applying unit simplification on {¬a}:{¬b}, we have {¬a}:{}. Since the dependency set of this empty clause contains ¬a, we have to try the other subcase with ¬a assigned FALSE. Adding {a}:{a} to S_2 and doing unit simplification on this, we have S_4:

- {a}:{¬b}
- {a}:{b}

Applying unit simplification on {a}:{¬b}, we have {a}:{}. Since these two subcases, ¬a assumed TRUE and ¬a assumed FALSE, depend on ¬a and a respectively, the dependency set of the case analysis on ¬a is {}. This dependency set is the dependency set of the subcase with ¬p assigned TRUE and does not contain ¬p, so we don't need to consider the subcase with ¬p assigned FALSE. Therefore, we conclude that S is unsatisfiable. The case tree is shown at the left-hand side of Figure 2.3. Notice that the case tree with the Davis-Putnam procedure for this example is more complicated, as shown at the right-hand side of Figure 2.3. Obviously, our propositional calculus prover can save much of the work for this simple example.

2.2.7 Built-In Facilities

Our propositional calculus prover has facilities that provide information about the input problem to the user. One is to find a model for the input set if the set is satisfiable. The other is to find a subset of unsatisfiable clauses from the input set if the set is unsatisfiable. This information may increase the confidence of the user in the prover.

2.2.7.1 Model Finding

Let M denote a collection of literals. Initially we set M to be {}. If the input set of ground clauses is satisfiable, then M is returned as a model for this set. The following are rules for building M:

1. For unit simplification, we have the following rules with l a literal:

$$\frac{\mathrm{M} \Rightarrow S_1 \wedge (\mathrm{C} \vee \neg l),\ l \in S_1}{\mathrm{M} \vee \{l\} \Rightarrow S_1 \wedge \mathrm{C}}$$

$$\frac{\mathrm{M} \Rightarrow S_1 \wedge (\mathrm{C} \vee l),\ l \in S_1}{\mathrm{M} \vee \{l\} \Rightarrow S_1}$$

$$\frac{\mathrm{M} \Rightarrow S_1,\ l \in S_1}{\mathrm{M} \vee \{l\} \Rightarrow S_1}$$

   M $\Rightarrow$ A means that the set of clauses is A and the collection of literals built up to this point is M.

2. For case analysis on p, we treat it like adding unit clauses p and $\neg$p in order.

3. Suppose M = M$'$ at a case analysis on p. If the subcase with p assigned TRUE is unsatisfiable, we have to reset M to M$'$ and start the other subcase with p assigned FALSE.

4. When we have M $\Rightarrow$ {}, M is a model for the input problem.

Notice that we don't stop until an empty set is derived. So we need a special treatment when a set derived is not empty but does not have either a positive or a negative clause. We apply case analysis and unit simplification until the set is reduced to the empty set. In this case, we lift the restriction that the clause selected for case analysis has to be negative.

Example 2.2 Suppose S contains the following clauses:

1. {$\neg$p(b, a), $\neg$p(b, f(b)), $\neg$p(f(b), b)}

2. {$\neg$p(a, a)}

3. {¬p(f(b), a), ¬p(f(b), b), ¬p(b, f(b))}
4. {p(b, f(b)), p(b, a)}
5. {p(f(b), b), p(b, a)}
6. {¬p(b, a), ¬p(b, b), ¬p(b, b)}
7. {p(f(a), a)}
8. {¬p(f(a), f(f(a))), ¬p(f(f(a)), f(a))}
9. {¬p(a, f(a))}

The propositional calculus prover detects S satisfiable and outputs the following model M:

- ¬p(f(b), a)
- p(f(b), b)
- p(b, f(b))
- ¬p(b, a)
- ¬p(f(a), f(f(a)))
- ¬p(a, f(a))
- p(f(a), a)
- ¬p(a, a)

2.2.7.2 Subset of Unsatisfiable Clauses

The idea is similar to dependency sets. However, we keep track of clauses instead of literals. Let A:B represent a clause B with *dependent clause set* A. Initially we have C:C for all clauses. We have the following dependency rules:

1. For unit simplification, we have the following rules with l a literal:

$$\frac{S_1 \wedge [D_1 : (\mathrm{C} \vee \neg l)],\ [D_2 : l] \in S_1}{S_1 \wedge [(D_1 \cup D_2) : \mathrm{C}]}$$

$$\frac{S_1 \wedge [D_1{:}(C \vee l)],\ [D_2{:}l] \in S_1}{S_1}$$

2. For case analysis, when doing case analysis on p, first we assume p is true. This is like adding {}:p to S. When assuming p is false, we add {}:¬p to S.

3. The dependent clause set of a case analysis is that of the first subcase if a right cutoff occurs, or that of the other subcase if a left cutoff occurs, or the union of dependent clause sets of the subcases if none of cutoffs occur. If an empty clause is derived in a subcase, then the dependent clause set of the subcase is the dependent clause set of the empty clause.

The dependent clause set of the top case analysis then contains an unsatisfiable subset of the input clauses. Sometimes, the subset obtained is a minimal one.

We used a technique, called delayed merging, for finding a subset of unsatisfiable clauses. A recursive expression of dependent clauses is created for each clause until an empty clause is found, where the expression of dependent clauses with this empty clause is evaluated to form an ordered set. For example, suppose X and Y are two expressions of dependent clauses. We use merge(X, Y, Z) to denote that Z is the resulting expression of the union of X and Y, where Z is a variable and X and Y are similar expressions. We don't evaluate Z until the clause with Z is an empty clause. The evaluation of an expression is done recursively. This saves the evaluation of merges until they are needed. Thus we are doing lazy merging.

Example 2.3 Suppose S contains the following clauses:

1. {g(f(a), a), g(a, a)}
2. {g(f(f(a)), f(a)), ¬g(a, f(a))}
3. {g(f(f(f(b))), f(f(b))), ¬g(f(b), f(f(b)))}
4. {g(f(f(b)), f(b)), ¬g(b, f(b))}
5. {g(f(b), f(f(b))), ¬g(f(f(b)), f(b))}
6. {g(b, f(b)), ¬g(f(b), b)}
7. {g(f(a), f(f(a))), ¬g(a, f(a))}
8. {g(f(f(b)), f(f(f(b)))), ¬g(f(b), f(f(b)))}

9. $\{g(f(b), f(f(b))), \neg g(b, f(b))\}$
10. $\{\neg g(f(a), a), \neg g(f(f(a)), f(a))\}$
11. $\{\neg g(f(a), a), \neg g(a, f(a))\}$
12. $\{\neg g(f(b), a), \neg g(f(f(b)), f(b))\}$
13. $\{\neg g(b, a), \neg g(f(b), b)\}$
14. $\{\neg g(a, a)\}$
15. $\{\neg g(f(f(b)), a), \neg g(f(b), f(f(b)))\}$
16. $\{\neg g(f(b), a), \neg g(b, f(b))\}$
17. $\{g(f(f(a)), f(a)), g(f(a), a)\}$
18. $\{g(f(f(b)), f(b)), g(f(b), a)\}$
19. $\{g(a, f(a)), g(a, a)\}$
20. $\{g(f(a), f(f(a))), g(f(a), a)\}$
21. $\{g(f(b), f(f(b))), g(f(b), a)\}$
22. $\{g(f(b), b), g(b, a)\}$
23. $\{g(b, f(b)), g(b, a)\}$
24. $\{g(f(b), b), \neg g(b, b)\}$
25. $\{g(b, f(b)), \neg g(b, b)\}$
26. $\{\neg g(b, a), \neg g(b, b)\}$

The propositional calculus prover detects S unsatisfiable and outputs the following subset of unsatisfiable clauses which happens to be minimal:

- $\{g(f(a), a), g(a, a)\}$ (clause 1)
- $\{g(f(f(a)), f(a)), \neg g(a, f(a))\}$ (clause 2)
- $\{\neg g(f(a), a), \neg g(f(f(a)), f(a))\}$ (clause 10)
- $\{\neg g(a, a)\}$ (clause 14)
- $\{g(a, f(a)), g(a, a)\}$ (clause 19)

2.3 Soundness and Completeness

The hyper-linking strategy is obviously sound, that is, the proof that is found logically follows the axioms. This is justified by the following universal instantiation rule:

$$\frac{\forall \text{ X F(X)}}{\text{F(c)}}$$

where F is a formula and c is some arbitrary constant in the domain.

We now prove the completeness of the hyper-linking strategy.

Definition. A mapping ϕ from clauses to clauses is an *instance mapping* if for all clauses C, C is an instance of ϕ(C).

Lemma 2.3 Let T be an unsatisfiable set of ground instances of set S of clauses. Suppose Gr(S) is satisfiable. Let

$$\phi : T \rightarrow S$$

be an instance mapping. Extend ϕ to map the literals of C to the corresponding literals of ϕ(C); for this, it may be necessary to distinguish the literals in different clauses. Then there are literals L, M in T that are complementary but Gr(ϕ(L)) and Gr(ϕ(M)) are not complementary.

Proof. Let I be a model of Gr(S). Suppose the lemma is false; then for all literals L, M of clauses in T, if L and M are complementary then Gr(ϕ(L)) and Gr(ϕ(M)) are complementary. Then I can be used to give a model I$'$ of T defined by I$' \models$ L if and only if I $\models$ Gr(ϕ(L)). This contradicts the assumption that T is unsatisfiable.

Definition. For a clause C, ||C|| is the sum of the number of occurrences of non-variable symbols in C, that is, individual, function, and predicate constants. ||L|| is defined similarly for a literal L. If

$$\phi : T \rightarrow S$$

is an instance mapping and T is $\{D_1, \ldots, D_n\}$ then $\|\phi\|$ is the sum over i of $\|\phi(D_i)\|$.

Definition. Suppose (L,M) is a link in S and L and M are contained in clauses C and D respectively. A *link operation* on S with (L,M) produces two instances Cθ and Dθ, where θ is the most general unifier of L and M.

Lemma 2.4 Let T be an unsatisfiable set of ground instances of set S of clauses. Let ϕ be as in the above lemma. Suppose Gr(S) is satisfiable. Then there are clauses C_1 and C_2 obtained by a link operation on S and there is an instance mapping

$$\phi' : T \to S \cup \{C_1, C_2\}$$

such that $\|\phi'\| > \|\phi\|$.

Proof. By Lemma 2.3, there must exist literals L, M in clauses D_1, D_2 of T such that L and M are complementary but Gr(ϕ(L)) and Gr(ϕ(M)) are not complementary. However, it must be that their most general unifier θ binds at least one variable to a non-variable term. Then,

$$\| \phi(L)\theta \| + \| \phi(M)\theta \| > \| \phi(L) \| + \| \phi(M) \|$$

Define ϕ' by ϕ'(D) = ϕ(D)θ for D = D_1 or D = D_2, and ϕ'(D) = ϕ(D) otherwise. Then $\|\phi'\| > \|\phi\|$.

Theorem 2.5 If S is an unsatisfiable set of clauses, then there is a set S$''$ of clauses obtained by a sequence of link operations on S such that Gr(S$''$) is unsatisfiable.

Proof. If Gr(S) is unsatisfiable, we are done. Suppose Gr(S) is satisfiable. We know by Herbrand's theorem that there is an unsatisfiable set T = $\{D_1, \ldots, D_n\}$ of ground instances of clauses of S. Let ϕ be an instance mapping from T to S. Now, $\|\phi\|$ is bounded by the sum over i of $\|D_i\|$. By Lemma 2.4, if Gr(S) is satisfiable then we can perform a link operation on two clauses of S obtaining S$'$ and

$$\phi' : T \to S'$$

with $\|\phi'\| > \|\phi\|$. If Gr(S$'$) is satisfiable, this can be repeated. Since the $\|\phi\|$ are bounded, this process must stop. Letting S$''$ be the set of instances at that time, Gr(S$''$) will be unsatisfiable by Lemma 2.4.

Corollary 2.6 If S is an unsatisfiable set of clauses, then there is a set S$''$ of clauses obtained by a sequence of hyper-link operations on S such that Gr(S$''$) is unsatisfiable.

Proof. From Theorem 2.5, we know there is a sequence of link operations producing a set of instances as desired. We convert this into a sequence of hyper-link operations. For this part of the proof we assume that T is a minimal unsatisfiable set of ground instances and R is the corresponding minimal sequence of link operations. Suppose a link operation between clauses C and D is in R. Suppose this link operation produces the instances Cθ and Dθ. We show that there is a hyper-link operation on C producing an instance of Cθ

suitable for our purposes, and similarly for D. Suppose C is $\{L_1, \ldots, L_m\}$. Now C must be $\phi(\mathrm{C}')$ for some ground clause $\mathrm{C}' = \{L'_1, \ldots, L'_m\}$ in T. For each L'_i, pick clause D_i in T and literal M_i in D_i such that $\mathrm{Gr}(L'_i)$ and $\mathrm{Gr}(M_i)$ are complementary. Such clauses must exist since T is minimal unsatisfiable (since clauses containing pure literals can be deleted without affecting satisfiability). Then if we work on the hyper-link $\{(L_1, \phi(M_1)), \ldots, (L_m, \phi(M_m))\}$ we will obtain an instance of Cθ. As in the proof of Theorem 2.5, we can construct a new instance mapping ϕ' and show that if a suitable set of ground instances has not been found, $\|\phi'\| > \|\phi\|$, hence we will eventually find a set of instances as desired.

Note the simplicity of the proof, which seems simpler than the proof of the completeness of resolution, and is directly related to Herbrand's theorem. This suggests that the completeness proofs for extensions and refinements of the hyper-linking strategy (including equality and paramodulation) will be simpler than for resolution. Note also that our strategy prover does not have a factoring operation on clauses; a factor of a clause is an instance with two literals unified.

2.4 Implementation

In practice, hyper-linking is implemented in Prolog using backtracking, to examine all possibilities of links for all literals in C. Each instance is asserted into the database. Note that if some literal in C is a ground literal, or becomes a ground literal due to previous unifications, then no backtracking is needed since linking will not instantiate this literal. Also, for efficiency, ground literals N_i are considered first. Note that we don't delete instances of clauses. Thus it is possible to have both a clause C and an instance D of C at the same time. This seems to be a disadvantage, but does not appear to cause problems. Also, instance deletion can be done in some cases, as explained later.

Given a problem, the hyper-linking strategy prover stops with one of the following results:

1. Unsatisfiable. The input clauses are contradictory.
2. Satisfiable. No new instances of clauses can be generated.
3. Time overflow. The time spent exceeds the specified time limit. In this case, we don't know whether or not the input clauses are contradictory.

We check time overflow after hyper-linking a nucleus and after a round of propositional unsatisfiability test. Thus the total time spent for a problem may be bigger than the specified time limit.

Since we don't store links as in connection graph resolution, duplicate instances occur in different rounds. It takes time to generate them and also takes time to check duplication. This may be helped by requiring that one of the electrons for a hyper-link be generated in the last hyper-linking round. Also, a breadth-first search is used for hyper-linking. This requires a lot of memory sometimes. We'll present techniques to avoid such problems later.

For some problems, only ground instances are needed. For example, consider the set of clauses $\{\{p(a)\}, \{\neg p(X), p(f(X))\}, \{\neg p(f(f(X)))\}\}$. The following seven clauses would be generated before a proof is found:

- $\{\neg p(a), p(f(a))\}$
- $\{\neg p(f(X)), p(f(f(X)))\}$
- $\{\neg p(f(a)), p(f(f(a)))\}$
- $\{\neg p(f(f(X))), p(f(f(f(X))))\}$
- $\{\neg p(f(f(a))), p(f(f(f(a))))\}$
- $\{\neg p(f(f(a)))\}$
- $\{\neg p(f(f(f(X)))), p(f(f(f(f(X)))))\}$

However, the non-ground instances are not necessary. If we restrict instances to be ground, then these non-ground instances are blocked. Our implementation provides an option for this.

Sometimes we found that restricting the number of literals in an instance is useful. Here we don't count ground literals. This restriction has two good effects. It reduces the size of the search space since some possible electrons have been discarded. It also reduce the number of instances from a nucleus since a long nucleus tends to produce many instances in different combinations. This restriction reflects the use of low-level heuristics, i.e. preferring clauses that are small, of human theorem proving ability.

2.5 Discussion

The hyper-linking strategy is an instance-based proving method. There are other instance-based proving methods. Among them Gilmore's procedure, the Prawitz procedure, and the linked conjunct procedure are major ones. However, our hyper-linking strategy prover is more efficient than any of them. We give a brief justification in the following.

1. Gilmore's procedure. In Gilmore's procedure, instances are generated by *blind substitution*. It replaces variables in a clause by elements of the Herbrand universe with level saturation. The *Herbrand universe* consists of all the variable free terms that we can make from the constants and functions in the given set of clauses. This results in exponential explosion of the search space. Consider the set S of the following two clauses

   - {p(X,g(X),Y,h(X,Y),Z,k(X,Y,Z))}
   - {¬p(U,V,e(V),W,f(V,W),X}

   The Herbrand universe of level 0 contains 1 element, level 1 contains 6 elements, ..., and level 5 contains about 10^{65} elements. Gilmore's procedure can't find a proof of S until level 5 in which about 10^{256} instances are generated at level 5. Furthermore, the unsatisfiability test is done using the multiplication method. Clauses are transformed to a disjunctive normal form before the test. This transformation is very inefficient. It results in exponential explosion of the number of conjuncts. It also combines parts of different clauses in a conjunct. In contrast, we adopt unification to find substitutions which results in fewer instances. For the above example, it generates only two instances from S. One instance is

   $$\{p(X,g(X),e(g(X)),h(X,e(g(X))),f(g(X),h(X,e(g(X)))),\\ k(X,e(g(X)),f(g(X),h(X,e(g(X))))))\}$$

   The other instance is the complement of this one. These two instances are contradictory. Therefore we have found a proof. We also use a propositional calculus prover which is more efficient and able to work on clauses directly.

2. The Prawitz procedure. The Prawitz procedure needs duplicate clauses. Each duplicate may have independent instantiations. For example, consider the following three clauses: {p(a)}, {p(b)}, and {¬p(X)}. Two copies of the third clause are required. One copy is instantiated to {¬p(a)}; the other copy is instantiated to {¬p(b)}. One problem with the Prawitz procedure is related to the determination of the number of duplicates for an input clause. It also uses the multiplication method; the use of disjunctive normal form inevitably produces the same difficulties as Gilmore's procedure.

3. The linked conjunct procedure. It also needs duplicates as the Prawitz procedure. It unifies literals in a clause one at a time, in contrast to our method which unifies all literals in a clause simultaneously. Due to this global unification, the instances generated by our method are fewer and more useful than those generated by the linked conjunct procedure. The linked conjunct procedure may have to consider a large number of cases. Suppose each literal has m possible mates (literals it can

complementarily unify with). If there are n literals, then there may be m^n cases to be considered. Even if each case can be done fast, m^n may be still a large number. This might involve a lot of duplication by case analysis since identical clauses may appear in many cases. In our method, identical clauses may also appear in different rounds, but the number of rounds is usually small (often 3 or 4). The linked conjunct procedure uses the Davis-Putnam procedure for unsatisfiability test which introduces more duplication by case analysis than our propositional calculus prover. Finally, the linked conjunct procedure performs the unsatisfiability test more often than our method.

Chapter 3

Refinements to the Hyper-Linking Strategy Prover

The efficiency of the hyper-linking strategy prover can be improved in two directions. The number of clauses generated can be reduced, and the unsatisfiability of the clauses retained can be detected before the propositional calculus prover detects it. In this chapter, we present some refinements. Unit simplification, ground unit generation, support sets, relevance confinement, UR resolution, and instance deletion help reduce the number of clauses to be kept. Unit simplification and small proof checking help detect unsatisfiability early.

3.1 Unit Simplification

After the hyper-linking phase, the clauses retained are simplified by a unit simplification phase which is analogous to operations in the Davis-Putnam procedure. If a unit clause {L} is derived and a clause C is obtained by hyper-linking such that C contains M, and M is an instance of the complement of L, then C is replaced by C - {M}. We can write this as an inference rule as follows:

$$\frac{S \cup \{C\} \cup \{\{L\}\},\ M \text{ in } C,\ M \text{ is an instance of } \neg L}{S \cup \{C - \{M\}\} \cup \{\{L\}\}}$$

This is called *unit literal deletion.* Also, if a unit clause {L} is derived, and clause C is derived, and C contains a literal M that is an instance of L, then C can be deleted. This is called *unit subsumption.* This may be written as an inference rule as follows:

$$\frac{S \cup \{C\} \cup \{\{L\}\},\ M \text{ in } C,\ M \text{ is an instance of } L}{S \cup \{\{L\}\}}$$

Note that unit literal deletion may produce more unit clauses, which may enable more unit literal deletions, and so on. Thus, a considerable amount of inference may occur in the unit simplification phase. As an example, suppose S is {{p}, {¬p, q}, {¬q, r}}. Then unit literal deletion with p produces {{p}, {q}, {¬q, r}}. There is now a new unit clause q, and unit literal deletion with q finally produces the set of clauses {{p}, {q}, {r}}. As an example of unit subsumption, if S is {{p}, {p, q}}, then unit subsumption with p causes {p, q} to be deleted, producing the set {{p}} of clauses.

We may perform unit literal deletion in a hyper-link operation. Suppose we are considering a hyper-link $\{(L_1, M_1), \ldots, (L_m, M_m)\}$ with nucleus $C = \{L_1, \ldots, L_m\}$ and substitution Θ. If electron M_i is a unit clause, then we delete $L_i\Theta$ from the instance CΘ. Therefore, the clause generated from this hyper-link operation is a subset of CΘ. We call this *embedded* unit literal deletion This embedding has many advantages. The clause generated is shorter, so we can save space in the database of clauses. Checking for duplication and other processing including unit simplification are simpler.

We now give an example which shows the use of embedded unit literal deletion. Suppose S is {{p(a)}, {¬p(X), q(f(X))}, {¬q(X), r(g(X))}, {¬r(X)}}. Let's call these clauses C_1, C_2, C_3, and C_4, respectively. Hyper-linking with nucleus C_2 and electrons {p(a)}, {¬q(X), r(g(X))}, we obtain the clause {¬p(a), q(f(a))} which is simplified to {q(f(a))} by embedded unit literal deletion. Hyper-linking with nucleus C_3 and electrons {q(f(a))}, {¬r(X)}, we obtain the clause {¬q(f(a)), r(g(f(a)))} which is reduced to {} by embedded unit literal deletion. This indicates that a contradiction has been found, and that S is unsatisfiable.

Unit simplification simplifies the set of clauses, in the sense that the size of the set of clauses is reduced, but proofs do not become harder to obtain. Moreover, unit simplification maintains the completeness of the hyper-linking strategy prover.

Unit simplification is very helpful for finding proofs. Many theorems can be proved by hyper-linking and unit simplification alone. For these proofs, the propositional calculus prover is not needed.

3.2 Ground Unit Generation

Suppose there is a clause C = $\{L_1, L_2, \ldots, L_m\}$ with ground literal L_i. Suppose $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$ are unifiable with unit clauses. Then we may replace C by $\{L_i\}$ in the database. This is essentially a UR resolution followed by unit subsumption. By UR resolution, a clause $\{L_i\}$ is generated which subsumes and deletes C from the database.

For example, consider the following set of clauses: {¬p(a)}, {p(a), q(X), ¬r(Y)}, {¬q(b)}, {r(c)}, {¬p(a), s(X)}, and {¬s(a)}. Clause {p(a), q(X), ¬r(Y)} can be replaced by {p(a)}, which simplifies {¬p(a), s(X)} to {s(X)}, which contradicts {¬s(a)}. Note that we have proven S unsatisfiable without applying any hyper-link operations.

Ground unit generation makes small proof checking more efficient because we don't need to consider non-unit electrons which contain ground literals.

3.3 Support Sets

3.3.1 Descriptions

In resolution, set-of-support strategies [WRC65] are widely used. Given a set S of clauses, a support set is a subset T of S such that S - T is satisfiable. Such strategies restrict the generation of irrelevant resolvents and seem essential for applications in which there are a large number of clauses. We also use a number of support strategies to restrict which hyper-links are performed. Thus reduces the number of instances generated. One of these support strategies simulates forward chaining, another simulates backward chaining, and another simulates set-of-support in resolution. We permit these support methods to be interleaved in arbitrary orders on successive hyper-linking rounds, which allows us to combine forward and backward chaining, for example. We have found such strategies to be effective in reducing the size of the search space. We also permit combinations of support strategies to be used on a given round.

Let S be the set of input clauses. We have the following definition.

Definition. A *forward support set* is the set of all positive clauses in S. A *backward support set* is the set of all negative clauses in S. A *user support set* is a set of clauses provided by the user. If T is the user support set then we assume that S - T is satisfiable.

We say the clauses in the support set are forward supported, backward supported, or user supported respectively. As the prover proceeds, the number of supported clauses increases. More and more clauses are supported according to the following criterion. Suppose C is a nucleus of a hyper-link and $L_1 \ldots L_m$ are the positive literals that are electrons for C and $M_1 \ldots M_n$ are the negative literals that are electrons for C. Thus C has m negative literals, linked by L_i, and n positive literals, linked by M_j. Let $A_1 \ldots A_m$ be the clauses containing $L_1 \ldots L_m$, respectively, and let $B_1 \ldots B_n$ be the clauses containing $M_1 \ldots M_n$. Let D be the instance of C obtained by this hyper-link. Then D is *forward supported* if m = 0 (C is all positive) or if all A_i are forward supported. D is *backward supported* if n = 0 (C is all negative) or if all B_j are backward supported. D is *user supported* if C was user supported or if some A_i or B_j was user supported. By specifying that an instance be user supported, forward supported, or user supported, the number of instances generated can be reduced.

User support seems necessary when there are large numbers of clauses. The user will typically specify the user support set to be those clauses that come from the theorem to be proven, and the other clauses will typically be axioms of some theory. User support restricts attention to clauses that are in some way related to the theorem. This appears to be necessary when there are large numbers of axioms, for otherwise many clauses will be derived purely from the axioms. The set of support restriction in resolution has a similar effect.

The use of support sets maintains completeness of the hyper-linking strategy prover as shown below.

Let I be an interpretation. We define nt(I) to be the number of propositions assigned TRUE by I.

Lemma 3.1 Suppose S is a minimal unsatisfiable set of ground clauses obtained by hyper-linking. For all interpretations I there is a clause C in S such that C is false in I and C is forward supported after some number of rounds of hyper-linking with forward support strategy.

Proof. The proof is done by induction on nt(I). Suppose nt(I) is 0. Then I assigns all propositions FALSE. Since S is unsatisfiable, some clause C in S is assigned FALSE by I. C must be positive, and therefore C is forward supported.

Suppose the lemma is true for $nt(\mathrm{I}) <$ n. We show it for $nt(\mathrm{I}) =$ n. Let $L_1, L_2, \ldots, L_n$ be propositions assigned TRUE by I. Let $I_1, I_2, \ldots, I_n$ be defined this way:

- I_1 assigns $L_2, L_3, \ldots, L_n$ TRUE, others FALSE
- I_2 assigns $L_1, L_3, \ldots, L_n$ TRUE, others FALSE
- $\vdots$

- I_n assigns L_1, L_2, ..., L_{n-1} TRUE, others FALSE

Thus $nt(I_j)$ = n - 1. Since S is unsatisfiable, none of I_j makes S true. By assumption, there is a clause C_j false in I_j. If C_j does not contains L_j then C_j is false in I also. Also, since $nt(I_j)$ = n - 1, by induction, we know that C_j is forward supported eventually. We pick C = C_j. So, if some C_j is false in I_j for some j, and C_j does not contain L_j, we are done. Suppose now that for all j, the clause C_j contains L_j. Let C be some clause of S that is false in I. The literals $\neg L_1$, $\neg L_2$, ..., $\neg L_n$ may appear in C or some subset of them may appear; no other negative literals may appear. After all the clauses C_j have been found to be forward supported, the electrons L_1, L_2, ..., L_n will all be available, and so another round of hyper-linking with forward support strategy will cause C to be forward supported.

Theorem 3.2 Suppose S is a minimal unsatisfiable set of ground clauses obtained by hyper-linking. Then all clauses in S will be forward supported after some number of rounds of hyper-linking with forward support strategy.

Proof. From Lemma 3.1, we know that for each interpretation I we get a forward supported clause C that is false in I. Let T be the set of all such C. Then T $\subseteq$ S and T is unsatisfiable. Since S is minimal unsatisfiable, we have T = S. Thus all clauses in S will be forward supported eventually.

Theorem 3.3 Suppose S is an unsatisfiable set of clauses and T is a minimal unsatisfiable set of ground instances of S. Then eventually every D in T will have some more general C in S that is forward supported.

Proof. Let ϕ be an instance mapping from T to S. We show by induction on nt(I) that if D is false in I then eventually some C will be produced by hyper-linking such that C is forward supported and C = ϕ(D).

Suppose nt(I) is 0. The D is positive as before. So C is positive too, and therefore forward supported.

Suppose the theorem is true for nt(I) < n. We show it for nt(I) = n. Suppose I assigns L_1, L_2, ..., L_n TRUE, other propositions FALSE. Let I_1, I_2, ..., I_n and D_1, D_2, ..., D_n be as before; D_j is false in I_j. If any D_j does not contain L_j then D_j is false in I too and we can let D be D_j. Suppose all D_j contain L_j. Let C_1, C_2, ..., C_n be $\phi(D_1)$, $\phi(D_2)$, ..., $\phi(D_n)$. Let D be some clause false in I. By induction, some such C_j will eventually be forward supported. Then electrons L'_1, L'_2, ..., L'_n will be available, where $L'_j \in C_i$ and $L'_1 = \phi(L_i)$. These electrons can be used to produce a forward supported instances of D by a hyper-link operation. This completes the proof.

Corollary 3.4 The hyper-linking strategy prover is complete with forward support strategy.

Proof. Suppose the set S of clauses is unsatisfiable. From Theorem 3.3, we know that eventually ϕ(D) is forward supported for all D. Thus any hyper-link between the clauses ϕ(D) is allowed. Since unrestricted hyper-linking strategy is complete, we will get a proof for S eventually.

Corollary 3.5 The hyper-linking strategy prover is complete with backward support strategy.

Proof. The proof is similar to that for forward support strategy. However, we replace nt(I) by nf(I) which means the number of propositions assigned FALSE by I.

Corollary 3.6 The hyper-linking strategy prover is complete with user support strategy if the clauses not contained in the user support set are satisfiable.

Proof. The proof is similar to that for forward support strategy if we change the signs of some atoms in the right way. Forward support is like user support if positive clauses are chosen as the support set. By changing signs of some atoms, we can get the effect of user support.

A round of hyper-linking can be restricted to be forward supported, backward supported, user supported, or any combination of the three. If forward support is specified, for example, then only forward supported hyper-links are performed. If a round is forward and user supported, then only instances which are both forward and user supported, are retained. Also, various support criteria can be interleaved. For example, [f, b] indicates that alternate hyper-linking rounds will be forward supported and backward supported.

We also have implemented a "throw away" strategy, which permits a nucleus to be deleted in certain cases. Suppose we are doing a round of forward support. Suppose a nucleus C is forward supported. Then C can be deleted after hyper-linking. The reason is that any instances needed for the proof, will be generated during that round, so that C is not needed.

3.3.2 Behavior of Different Support Strategies

The prover is sensitive to the support strategy it uses. Theoretically, user support is most goal-oriented if the support set is chosen correctly. Unfortunately, many instances become user-supported rapidly, since only one electron needs to be user-supported for an instance to be user supported. The use of user support set effectively controls the generation of level 1 instances, as mentioned in [Wos88], but fails to control the instances of levels greater than 1. Forward support is more restrictive because all the electrons for the negative literals have to be forward supported in order for an instance to be forward supported. However, it may have trouble when a problem contains many facts which are usually positive clauses, or when the number of negative literals in a nucleus is small. Backward support is somewhat goal-oriented since the negation of the theorem usually contains negative clauses. It is also restrictive; all the electrons for the positive literals are required to be backward support. However, it may have difficulty when a problem contains many negative clauses, or when the number of positive literals in a nucleus is small. Also, different support strategies may have different search patterns for the proof. The number of rounds needed and the size of the search space may vary.

Example 3.1 Consider the following set S of clauses which is a reduced version of "wos15" [Sti88]:

1. C_1: {o(a)}
2. C_2: {o(b)}
3. C_3: {o(Z), ¬p(X, g(Y), Z), ¬o(X), ¬o(Y)}
4. C_4: {p(a, b, c)}
5. C_5: {p(e, X, X)}
6. C_6: {p(X, e, X)}
7. C_7: {p(g(X), X, e)}
8. C_8: {p(X, g(X), e)}
9. C_9: {p(X, Y, f(X,Y))}
10. C_{10}: {p(X, V, W), ¬p(X, Y, U), ¬p(Y, Z, V), ¬p(U, Z, W)}
11. C_{11}: {p(U, Z, W), ¬p(X, Y, U), ¬p(Y, Z, V), ¬p(X, V, W)}
12. C_{12}: {¬o(c)}

Let's show S unsatisfiable with several support strategies. For simplicity, we only list the clauses needed in the proof. In fact, the search space for this problem is very large without further control. All the proofs are obtained with sliding priority which will be discussed later.

1. Forward support.
   (a) In the first round, {p(e, X, g(g(X)))} is generated from C_{11} and {o(e)} is generated from C_3.

(b) In the second round, {p(a, g(g(b)), c} is generated from C_{10} and {o(g(b))} is generated from C_3.

(c) In the third round, {o(c)} is generated from C_3. This contradicts C_{12}.

2. Backward support.

(a) In the first round, {¬p(a, g(Y), c), ¬o(X)} is generated from C_3.

(b) In the second round, {p(a, g(Y), c), ¬p(e, b, g(Y))} is generated from C_{10} and {o(g(b)), ¬o(e)} is generated from C_3.

(c) In the third round, {p(e, b, g(g(b)))} is generated from C_{11}, {o(e)} is generated from C_3, and {¬p(a, g(g(b)), c), ¬o(g(b))} is generated from {¬p(a, g(Y), c), ¬o(X)}.

(d) In the fourth round, {p(a, g(g(b)), c} is obtained from {p(a, g(Y), c), ¬p(e, b, g(Y))}.

The clauses {p(a, g(g(b)), c}, {¬p(a, g(g(b)), c), ¬o(g(b))}, {o(e)}, {p(e, b, g(g(b)))}, and {o(g(b)), ¬o(e)} are unsatisfiable.

3. User support. Suppose the initial user support set contains C_1, C_2, and C_{12}. The number of rounds and the proof are identical to those with backward support.

The performance comparison of the prover with forward, backward, and user support strategies can be seen from Table 3.1.

As mentioned previously, user support is goal-oriented but less restrictive, while forward support is more restrictive but does not restrict attention to the theorem. We have a method of combining the advantages of both kinds of support, although this has not yet been implemented. Given two types of support x and y, where x and y may be user, forward, or backward, we define x:y support as follows: Suppose C is a nucleus of a hyper-link and D is the instance of the hyper-link. Then D is x:y supported if either C is not x supported and D is x supported, or if C is x supported and D is both x and y supported. The restriction to x:y support is complete if x and y are chosen arbitrarily from user support, forward support, or backward support. However, the following four combinations seem most interesting: user:forward, user:backward, forward:backward, and backward:forward. For example, user:forward first generates user supported instances, and then restricts attention to their instances that are both user and forward supported. This is complete because eventually all clauses needed for the proof will be user supported, and after that, forward support can be used. Thus we restrict attention to instances related to the theorem, and also have the advantages of forward support.

Without using a support strategy, the hyper-linking strategy prover concludes that the input problem is satisfiable if no new clauses can be generated. We have to be careful if

problem	forward support (sec)	backward support (sec)	user support (sec)
burstall	5.317	16.100	5.767
prim	9.100	71.717	258.983
ex4t1	144.017	704.600	362.300
ls17	11.800	36.150	91.717
ls36	59.283	325.717	73.867
ls65	58.750	190.933	53.966
ls75	11.217	11.517	11.400
wos1	29.083	211.033	71.100
wos2	21.617	21.667	20.833
wos6	39.133	219.550	103.667
wos10	45.783	200.800	138.417
wos11	58.783	246.283	86.950
chang_lee_8	9.817	28.317	185.250
chang_lee_9	5.183	7.933	9.417
salt	29.483	31.900	22.917
average	35.891	154.947	99.770

Table 3.1: Performance Comparison of different support strategies

support strategies are used. The user may provide an inappropriate user support set which causes the prover to make a wrong conclusion even though the input problem is unsatisfiable. However, the prover always make right decisions if the user uses forward support strategy, backward support strategy, or user support strategy with the right support set.

3.4 Relevance Confinement

We also define distances for each kind of support. Thus a clause may have a *forward distance*, a *backward distance*, and a *user distance*. These are defined as follows: Let D be an instance as above. Then the forward distance of D is zero, if D is all positive, otherwise it is 1 + max{forward distances of A_i}. The backward distance of D is zero, if D is all negative, otherwise it is 1 + max{backward distances of B_j}. The user distances of all clauses are initially infinity, except clauses in the user support set, which have a user distance of zero. The user distance of D is 1 + min{user distances of A_i and B_j}. These distances measure how closely related D is to various support sets. The larger the distance, the less related D is. We have a priority measure that can be made to depend on these distances, and prefer more "relevant," or closely related, clauses.

Consider the following clauses: {p(a)}, {¬p(X), ¬p(f(X))}, and {¬p(f(f(a)))}. Clauses {p(a)}, {¬p(a), ¬p(f(a))}, {¬p(f(a)), ¬p(f(f(a)))}, and {¬p(f(f(a)))} are instances obtained from hyper-linking operations. Let x:y mean that the forward distance is x and the backward distance is y. Then {p(a)} has distances 0:3, {¬p(a), ¬p(f(a))} has distances 1:2, {¬p(f(a)), ¬p(f(f(a)))} has distances 2:1, and {¬p(f(f(a)))} has distances 3:0.

Distances can be used in two ways. The user can set a relevance bound to discard any clause whose distance is larger than the relevance bound. Apparently, we may set the relevance bound 3 for the above example. Distances can also be used in the sliding priority that will be discussed later.

There is a difficulty in distance computation. For the above example, what is the backward distance of {¬p(a), ¬p(f(a))} in the first round? We don't know yet that there is a path from ¬p(f(a)) through {¬p(f(a)), ¬p(f(f(a)))} to {¬p(f(f(a)))}. We can't say the backward distance is infinity, otherwise {¬p(a), ¬p(f(a))} would be deleted. We solve this difficulty by using incremental computation. Initially, all forward supported clauses have forward distance 0 and all clauses that are not forward supported have forward distance 1. All backward supported clauses have backward distance 0 and all clauses that are not backward supported have backward distance 1. Similarly, all user supported clauses have user distance 0 and all clauses that are not user supported have user distance 1. Thus we record a lower bound on the support distances.

Our distance measure is an attempt to restrict the proof size. However, we are really measuring the depth of the proof, not its size. This different measure may explain some differences in performance between CLIN and the iterative deepening prover of Stickel [Sti86] as well as the modified problem reduction format prover [Pla88]. Both of those provers measure the size rather than the depth of a proof. Sometimes it may be preferable to use one measure, and sometimes it may be preferable to use another.

3.5 UR resolution

We note that the UR resolution strategy [WOLB84], though incomplete, is compatible with the philosophy of our system, namely, that literals from different clauses should never be combined. UR means "unit resulting," and this strategy only generates resolvents that are unit clauses. This strategy is often used by the Argonne group [McC89]. *UR resolution* is an inference rule which produces a unit clause (UR resolvent) from a set of clauses one of which is a nonunit clause (nucleus) while the remaining are unit clauses (electrons). One application of UR resolution which generates a UR resolvent is called a *UR resolution step*. We have also found that this strategy is effective in our system, not surprisingly. UR resolution can be viewed as a kind of hyper-linking operation, in which all but one of the electrons is a unit clause, and the other electron is absent. For example, if the nucleus is

{¬p(X), ¬q(X), ¬r(X)} and the electrons for the first and third literals are p(a) and r(a), then the instance {¬p(a), ¬q(a), ¬r(a)} is generated. The electron for the second literal ¬q(X) is missing here. The literals ¬p(a) and ¬r(a) are immediately deleted by embedded unit literal deletion, resulting in the unit clause ¬q(a). Thus we have obtained a unit clause ¬q(a) by resolving the given clause with two unit clauses; this is essentially the same as UR resolution.

Example 3.2 Let's use UR resolution to solve the following problem "chang_lee_8" [Sti88]:

- C_1: {l(1, a)}
- C_2: {d(X, X)}
- C_3: {¬d(X, a), ¬p(X)}
- C_4: {d(X, Z), ¬d(Y, Z), ¬d(X, Y)}
- C_5: {p(f(X)), ¬l(1, X), ¬l(X, a)}
- C_6: {d(f(X), X), ¬l(1, X), ¬l(X, a)}
- C_7: {d(g(X), X), p(X)}
- C_8: {l(1, g(X)), p(X)}
- C_9: {l(g(X), X), p(X)}

In the first round, only the clause {¬p(a)} is generated from the nucleus C_2. In the second round, {l(g(a),a)} is generated from C_9, {l(1,g(a))} is generated from C_8, and {d(g(a),a)} is generated from C_7. In the third round, {d(f(g(a)),g(a))} is generated from C_6, {p(f(g(a)))} is generated from C_5, and {¬p(g(a))} is generated from C_3. These 16 clauses are detected unsatisfiable by small proof checking which will be discussed later. Notice that UR resolution can be very efficient for this problem since all the clauses generated are needed in the proof.

UR resolution is a very efficient strategy. We have obtained many proofs with it. It is particularly useful if the top-level supervisor, which will be discussed later, is used. It can be used together with support strategies. For example, [UR, f] means UR resolution and forward support are used interleavingly. We have also noted that user support strategy together with UR resolution can be very useful for some hard problems, such as "wos15". The search space may be reduced a lot by requiring that one of the unit clauses be user supported. This combination is represented by [uUR] or [URu].

3.6 Instance Deletion

Since our hyper-linking strategy prover does not delete instances of clauses, it may keep more clauses than a resolution prover. First, we note that not deleting instances can be an advantage, because it means that duplicate checking can be done much faster. Hashing or a discrimination net can be used for this purpose, in time essentially independent of the number of instances. However, partially because of our use of Prolog as the implementation language, we have not fully exploited this possibility.

We have experimented with methods of instance deletion, and have obtained some improvement on a few problems. For example, it is complete to only delete instances of non-forward supported clauses. The reason is that eventually all the instances needed will be forward supported, and therefore retained. Also, if the set of input clauses is a Horn set [HW74], then unit resolution is complete, and all instances can be deleted. In fact, instance deletion is complete whenever unit resolution suffices to obtain a proof, although this may not be known in advance. We have obtained significant improvements on some problems using these ideas, but significant slowdowns on others. It seems that retaining instances is not usually a major problem for our hyper-linking strategy prover.

Instance deletion can be made more efficient in the following way: Hyper-linking is performed by creating a list of all the literals that appear in clauses in S and using these literals as electrons in the hyper-links. If all instances are deleted, then we can delete literals in this list that are instances of other literals in this list, because such literals will produce clauses that are instances of clause produced by more general literals. A more restrictive but similar idea can be used if instances of non-forward supported clauses are deleted.

3.7 Small Proof Checking

Our hyper-linking strategy prover uses a breadth-first search to generate instances. The breadth-first search is complete. However, its storage requirement is proportional to the nodes that have already been searched successfully in the search tree. The prover may fail to solve a problem due to the use of large amount of memory. In order to reduce storage requirements, we have to make the stored search tree small. There are two ways to achieve this goal. One is to make the stored tree thin. The other is to make it shallow.

In this section, we propose small proof checking as a way to make the stored search tree shallow. The problem of making it thin will be dealt with later. Small proof checking is performed after the propositional unsatisfiability test. The basic idea of small proof checking is to find "early termination" due to a "small proof." It detects a proof as early as possible so that the nodes of further levels in the search tree need not be stored. In theorem proving, unit clauses are usually very important and can be processed fast. We would like to take advantage of them. A clause is picked and examined if it can resolve with unit clauses directly or indirectly. The given set of clauses is unsatisfiable if such clause exists. For example, consider the set S of clauses {p(a)}, {¬p(X), q(Y)}, and {¬q(b)}. {¬p(X), q(Y)} resolves with unit clauses {p(a)} and {¬q(b)}. So S is unsatisfiable. This test can be done without storing clauses and faster than by performing another round of hyper-linking, unit simplification, and propositional unsatisfiability testing.

Small proof checking is designed so that the storing of the last two levels of the search tree may be avoided. For a search tree of N levels with average branching factor B (for us, the branching factor of a node is the number of clauses generated from this node by hyper-linking), the total amount of storage required for this tree is proportional to $\frac{B^N-1}{B-1}$. Therefore, the ratio of space needed for two search trees of same branching factor B but with levels N and $N-2$ respectively is $\frac{B^N-1}{B^{N-2}-1}$. For example, if B is 3 and N is 5, then the small proof checking can save almost 90% of the storage.

Small proof checking may cause a combinatorial problem due to duplication by combination. However, we have ways to control this problem. An iterative deepening on proof size is developed so that a small proof with the shortest proof size is always found. A time control mechanism is designed. This time control also provides a nice trade-off between time and space considerations.

The small proof checking is a general idea. It can be applied to any theorem proving method using breadth-first search strategy, such as the resolution method or Gilmore's procedure.

3.7.1 Descriptions

Some provers gain speed by looking for pairs L and M of unit clauses such that L and the complement of M are unifiable. If such are found, then a proof of unsatisfiability is immediate. Searching directly for such pairs may be much faster than performing many inferences until a proof of unsatisfiability is found. The simple small proof checking is a generalization of this "early termination" idea. Instead of looking for pairs of unit clauses, we look for a hyper-link in which all the electrons are unit clauses. If this is found, then we immediately know that the clauses are unsatisfiable. For example, if the nucleus of the hyper-link is {¬p(X), ¬q(X), r(X)} and the electrons are unit clauses p(a), q(a), and ¬r(a), then the instance {¬p(a), ¬q(a), r(a)} is generated. This will be simplified to the empty clause by unit literal deletion using the electrons. Searching for such hyper-links is much faster than doing a round of hyper-linking, since only unit electrons are used, and since the instances are not saved unless an empty clause is derived, which stops the search. We call this *small proof checking* because such a hyper-link corresponds to a small proof that the set of clauses is unsatisfiable. This simple small proof checking does not combine parts of different clauses, so the problem of duplication by combination does not occur.

A more sophisticated small proof checker allows searching for small proofs of a more complicated structure. Instead of looking for a hyper-link in which all of the electrons are unit clauses already existing in the set of instances, we look for a hyper-link in which all of the electrons are either existing unit clauses, or derivable by one UR-resolution step. This allows us to find larger proofs, and sometimes terminate the search even earlier than with the simple method described above.

An operational definition of a small proof is given below.

Definition. Suppose (S, E) is a connection graph. Suppose C is a clause $\{L_1, \ldots, L_m\}$ in S and a hyper-link H of C in E is $\{(L_1, M_1), \ldots, (L_m, M_m)\}$. H is a *small proof* if there are clauses D_i such that M_i is derivable by UR resolution from D_i.

Theorem 3.7 If a small proof is found for a set S of clauses, then S is unsatisfiable.

Proof. Suppose such a small proof exists. From the operational definition, we know that $(D_i - M_i)\alpha_i$ hyper-links unit clauses with unifier β_i where α_i is the most general unifier of L_i and M_i. Therefore, $D_i\alpha_i$ is resolved to $\{M_i\theta_i\}$ by these unit clauses. We also know that $L_i\theta_1 \cdots \theta_{i-1}\alpha_i$ is identical to $(\neg M_i)\alpha_i$ where $\theta_i = \alpha_i\beta_i$. So $L_i\theta_1 \cdots \theta_{i-1}\alpha_i\beta_i$ is identical to $(\neg M_i)\alpha_i\beta_i$. So $L_i\theta_1 \cdots \theta_{i-1}\theta_i$ is identical to $(\neg M_i)\theta_i$. Therefore, the clause $\{L_1, \ldots, L_m\}$ is resolved to the empty clause by unit clauses $\{M_i\theta_i\}$. This completes the proof.

Definition. The size of a small proof is defined as

$$d_1 + d_2 + \ldots + d_m$$

where d_i is defined as follows:

1. If $L_i\theta_1 \cdots \theta_{i-1}$ is ground then d_i is 0, or

2. If M_i is a unit clause then d_i is 0, or

3. Otherwise, d_i is one less than the number of literals in D_i.

The complicated small proof checking may save the storage for the last two levels of the search tree. We may think that the last level concerns the generation of the empty clause and the next to last level concerns the generation of unit clauses $\{M_i\theta_i\}$. However, checking does combine parts of different clauses if the electrons being considered are not unit clauses. This combination may cause combinatorial explosion, so we have to control it. Several control mechanisms are developed and will be discussed later. These mechanisms include iterative deepening on proof size, a time control, and restrictions on nuclei and electrons.

3.7.2 Iterative Deepening on Proof Size

The size of the search space of small proof checking increases dramatically as the allowable proof size increases. This can be seen from the following illustration. Suppose S contains N clauses, of which there are N_1 unit clauses and N_2 two-literal clauses. Suppose each clause has L literals on the average. For simplicity, we assume that all literals N_i are non-ground. If we restrict a small proof to have proof size 0, then the search space would be of the order ${N_1}^2(1 + N_2)$. If we restrict a small proof to have proof size 1, then the search space would be of the order ${N_1}^2(1 + N_2 + 2{N_2}^2)$.

Because of the combinatorial explosion of the search space, it is reasonable to explore a way of obtaining a small proof with smallest proof size. The iterative deepening method [Kor85, ST85] is used for this purpose. The idea is as follows. Suppose B_S is the specified proof size bound. We start with allowable size bound 0. If a small proof is found, then we are done. Otherwise we increase the allowable size bound by 1 and try again. If the allowable size bound is greater than B_S, we stop and fail to find a small proof for S. The iterative deepening method is guaranteed to find a small proof with minimum proof size that is not greater than B_S if such small proof exists.

3.7.3 A Time Control

We don't know how big the proof size is even if there exists a small proof for S. If we set the proof size bound small, we may miss a small proof. If we set the proof size bound large, the time spent on small proof checking may be enormous. To handle this problem, we set the initial proof size bound 0 and let the prover increase the proof size bound until either a small proof is found or the time spent on small proof checking exceeds a certain amount of time. A *controlling time* is defined as a function of the time spent in the last hyper-linking round. If the small proof checking spends more time than the controlling time, we stop it. This time control makes sense because of the following reasons:

1. In general, the number of clauses generated is proportional to the time spent in the hyper-linking. Therefore, it makes sense to allow more time for small proof checking

if more time is spent in hyper-linking since more clauses are involved.

2. It provides a nice trade-off between time and space considerations. In general, if we miss a small proof with proof size D_S in round R, there exists a small proof with size D'_S in the next round such that $D'_S \leq D_S$.

This control takes the burden off users of setting an exact proof size bound for various problems. The proof size bound can be set to infinity. The small proof checking will try to find an easiest proof within the controlling time in the current round. If it fails, it will wait and try again in the next round.

3.7.4 Restrictions on Nuclei or Electrons

The search space with small proof checking can be reduced by restricting nuclei or electrons to be considered.

Theorem 3.8 Suppose there is a small proof P′ with nucleus C′. If C′ is an instance of C, then there is also a small proof P with nucleus C.

Proof. Suppose C′ = $\{L'_1, \ldots, L'_m\}$ and P′ is $\{(L'_1, M_1), \ldots, (L'_m, M_m)\}$. Suppose D_i are the electrons containing M_i for $1 \leq$ i $\leq$ m. Let $C' =$ Cδ and C = $\{L_1, \ldots, L_m\}$. From the operational definition of the small proof, we know

$$L'_1\theta'_1 \cdots \theta'_{i-1}\alpha'_i = L_1\delta\theta'_1 \cdots \theta'_{i-1}\alpha'_i = (\neg M_i)\alpha'_i$$

Since there are no common variables between D_i and C, we have

$$L_1\theta'_1 \cdots \theta'_{i-1}\delta\alpha'_i = (\neg M_i)\delta\alpha'_i$$

If we define α_i to be $\delta\alpha'_i$, we have

$$L_1\theta_1 \cdots \theta_{i-1}\alpha_i = (\neg M_i)\alpha_i$$

where θ_i is $\delta\theta'_i$. Note that $(D_i - \{M_i\})\alpha_i$ is identical to $(D_i - \{M_i\})\alpha'_i$ and $(D_i - \{M_i\})\alpha_i$ hyper-links the unit clauses that $(D_i - \{M_i\})\alpha'_i$ does and with same unifiers β_i. Therefore, P is also a small proof where P is the hyper-link $\{(L_1, M_1), \ldots, (L_m, M_m)\}$. This completes the proof.

Notice that the proof size of P may be bigger than that of P′ since the number of ground literals in C′ may be less than that in C.

Theorem 3.9 Suppose there is a small proof P$'$ = $\{(L_1, M'_1), \ldots, (L_m, M'_m)\}$ with nucleus C = $\{L_1, \ldots, L_m\}$ and electrons $M'_1(D'_1), \ldots, M'_m(D'_m)$. Suppose each $M'_i(D'_i)$ is an instance of $M_i(D_i)$. Then there is also a small proof P with nucleus C and electrons $M_1(D_1), \ldots, M_m(D_m)$.

Proof. Let D_i be $D'_i\delta_i$. We know

$$L_1\theta'_1 \cdots \theta'_{i-1}\alpha'_i = (\neg M'_i)\alpha'_i = (\neg M_i)\delta_i\alpha'_i$$

Since there are no common variables between D_i and C, we have

$$L_1\theta'_1 \cdots \theta'_{i-1}\delta_i\alpha'_i = (\neg M_i)\delta_i\alpha'_i$$

If we define α_i to be $\delta_i\alpha'_i$. we have

$$L_1\theta_1 \cdots \theta_{i-1}\alpha_i = (\neg M_i)\alpha_i$$

where θ_i is $\delta_i\theta'_i$. Note that $(D_i - \{M_i\})\alpha_i$ is identical to $(D'_i - \{M'_i\})\alpha'_i$ and $(D_i - \{M_i\})\alpha_i$ hyper-links the unit clauses that $(D'_i - \{M'_i\})\alpha'_i$ does and with same unifiers β_i. Therefore, P is also a small proof where P is the hyper-link $\{(L_1, M_1), \ldots, (L_m, M_m)\}$. This completes the proof.

Notice that the proof size of P may be bigger than that of P$'$ since the literals in C has a higher chance to be ground by D'_i than by D_i.

Corollary 3.10 Suppose S is a set of clauses. Delete all the instances from S and let the resulting set be S$'$. A small proof can be found in S if and only if a small proof can be found in S$'$.

In fact, we can only consider input clauses to be nuclei. The reason is that every clause is either an instance of an input clause or with some literals deleted from the instance. The first case is proved by Theorem 3.8. Consider the second case. A deleted literal is either unifiable with or subsumed by a unit clause. By the lifting lemma [CL73], if a small proof can be found with this clause as nucleus then a small proof can also be found with the input clause from which it was generated as nucleus. Of course, more electrons are needed for the input clause. Electrons can be further reduced by the following theorem.

Theorem 3.11 In small proof checking, we only need to consider the following two kinds of electrons: unit clauses or clauses containing no ground literals.

Proof. A clause belongs to one of the following classifications:

1. unit clause
2. containing only one ground literal
3. containing two or more ground literals
4. containing no ground literals

We prove the theorem by showing that it is impossible to find a small proof with electrons picked from the second and third categories.

- If an electron C containing the ground literal G is not a unit clause and if it is an electron of a small proof, then G must be M_i in a link (L_i, M_i) since if G is not M_i then it must have been deleted by unit literal deletion. Now the rest of the literals must be unifiable with unit clauses. However, in the ground unit generation phase (see section 3.2), a unit clause {G} must have been generated and C must have been replaced by {G}.
- If an electron C contains two or more ground literals, then one of the ground literals must be unifiable with a unit clause. This ground literal must have been deleted by unit literal deletion.

Therefore, we conclude that clauses that belong to the second and third classifications can't be electrons of a small proof.

3.7.5 Implementation

We have implemented both the simple small proof checking and the complicated small proof checking in our system CLIN. The implementation of the simple small proof checking is easy. We pick a nucleus and link all of its literals with unit clauses. Time control is not used in the simple small proof checking. The complicated small proof checking is implemented as follows. We first search for small proofs of size 0, then of size 1, then of size 2, et cetera, until some specified time bound is exceeded. We take care to avoid repeated work, so that the same work will not be performed for more than one size bound. To search for a small proof with specified size bound, we follow the operational definition of the small proof defined before. Suppose C is the nucleus to be considered. We pick one literal L each time. Then we pick some clause D, unify a literal of D with the complement of L, and try to unify the other literals of D with complements of unit clauses. At each step we keep track of the total size of the proof constructed so far. We also check the total time used so far before and after unifying the literals of each chosen clause D. This ensures that the gaps between time checks are not too large. If all clauses D can be found for all literals in C, and the other literals of D can be unified with complements of unit clauses as specified, then we have a small proof and we know S is unsatisfiable.

We avoid backtracking for ground literals, as in hyper-linking. For efficiency, we pick ground literals of C first since we don't backtrack for ground literals. We also order the clauses in a certain way. Nuclei are ordered by number of non-ground literals; electrons are ordered by number of literals. With this ordering, a simplest small proof is always found first if it exists.

3.7.6 Two Examples

We give two examples to illustrate how the small proof checking works.

Example 3.3 The following problem is "ls100" [Sti88]:

1. C_1: {e(b, c)}
2. C_2: {m(a, b)}
3. C_3: {m(Z_3, Y_3), ¬s(X_3, Y_3), ¬m(Z_3, X_3)}
4. C_4: {m(f(X_4, Y_4), X_4), s(X_4, Y_4)}
5. C_5: {s(X_5, Y_5), ¬m(f(X_5, Y_5), Y_5)}
6. C_6: {s(X_6, Y_6), ¬e(X_6, Y_6)}
7. C_7: {s(Y_7, X_7), ¬e(X_7, Y_7)}
8. C_8: {e(X_8, Y_8), ¬s(X_8, Y_8), ¬s(Y_8, X_8)}
9. C_9: {¬m(a, c)}

For convenience, we have already renamed the variables so that no common variables shared between clauses. The small proof checking fails with clauses having 1 and 2 literals. However, it succeeds with C_3. Let's see how it succeeds with C_3.

- Consider the first literal, m(Z_3, Y_3), of C_3. m(Z_3, Y_3) and the complement of ¬m(a, c) of C_9 are unifiable with the most general unifier θ_1 as $\{Z_3 \leftarrow \text{a}, Y_3 \leftarrow \text{c}\}$. d_1 is 0 since C_9 is a unit clause. Let C'_3 be $C_3\theta_1$, that is,

$$\{\text{m(a, c)}, \neg\text{s}(X_3, \text{c}), \neg\text{m(a}, X_3)\}$$

- Consider the second literal, $\neg$s(X_3, c), of C'_3. $\neg$s(X_3, c) and the complement of first literal, s(X_6, Y_6), of C_6 are unifiable. The unifier α_2 is $\{X_6 \leftarrow X_3,\ Y_6 \leftarrow \text{c}\}$. ($C_6$ - {s(X_6, Y_6)})α_2 hyper-links C_1 with unifier $\beta_2 = \{X_3 \leftarrow \text{b}\}$. d_2 is 1 since C_6 has two literals. Now θ_2 is $\{X_6 \leftarrow \text{b},\ Y_6 \leftarrow \text{c},\ X_3 \leftarrow \text{b}\}$. Let C''_3 be $C'_3\theta_2$, that is,

$$\{\text{m(a, c)}, \neg\text{s(b, c)}, \neg\text{m(a, b)}\}$$

- Consider the third literal, $\neg$m(a, b), of C''_3. $\neg$m(a, b) and the complement of m(a, b) of C_2 are unifiable with empty unifier θ_3. d_3 is 0 since C_2 is a unit clause.

Therefore, we have obtained a small proof for "LS100". The small proof is {(m(Z_3, Y_3), $\neg$m(a, c)), ($\neg$s(X_3, Y_3), s(X_6, Y_6)), ($\neg$m(Z_3, X_3), m(a, b))}. The proof size is $0 + 1 + 0 = 1$. Therefore, "LS100" is unsatisfiable.

Example 3.4 The following problem is "exx5", a verification condition from Hoare's FIND program [Ble77]:

1. C_1 = {lt(j, i)}
2. C_2 = {le(m, p)}
3. C_3 = {le(p, q)}
4. C_4 = {le(q, n)}
5. C_5 = {$\neg$le(b(p), b(q))}
6. C_6 = {le(X_6, Y_6), lt(Y_6, X_6)}
7. C_7 = {le(b(X_7), b(Y_7)), $\neg$le(m, X_7), $\neg$le(Y_7, j), $\neg$le(X_7, Y_7)}
8. C_8 = {le(b(X_8), b(Y_8)), $\neg$le(i, X_8), $\neg$le(Y_8, n), $\neg$le(X_8, Y_8)}
9. C_9 = {le(b(X_9), b(Y_9)), $\neg$le(m, X_9), $\neg$lt(X_9, i), $\neg$lt(j, Y_9), $\neg$le(Y_9, n)}

We can't find a small proof without any new clauses added. After one round of hyper-linking, the following two clauses are added into the database:

1. C_{10} = {$\neg$le(i, p)}
2. C_{11} = {$\neg$le(q, j)}

problem	no. of input clauses	cpu(sec)
wos27	30	45.133
wos28	74	1859.683
wos27_s	42	17.533
wos28_s	86	571.367

Table 3.2: The effect of clause splitting

Now, we can find a small proof with nucleus C_9. The electrons are C_5, C_2, C_6, C_6, and C_4. All d_i's are 0. Notice that C_{10} and C_{11} are required to generate UR resolvents M_3 and M_4 from the two occurrences of C_6 respectively. Therefore we have found a small proof for "EXX5" with proof size 0 after the first round. The small proof is {(le(b(X_9), b(Y_9)), ¬le(b(p), b(q))), (¬le(m, X_9), le(m, p)), (¬lt(X_9, i), le(X_6, Y_6)), (¬lt(j, Y_9), le(X_6, Y_6)), (¬le(Y_9, n), le(q, n))}.

3.8 Clause Splitting

The time to hyper-link a large clause can be exponential in the number of literals. By splitting we may reduce this. However, it may need more rounds to find the proofs. A clause can be split into 2 clauses by introducing a new predicate symbol. Suppose we have a clause C:

$$\{L_1, L_2, \ldots, L_m, L_{m+1}, L_{m+2}, \ldots, L_n\}$$

and new_pred is a new predicate symbol. C can be split into the following two clauses:

- $\{L_1, L_2, \ldots, L_m, \text{new\_pred}(\overline{X})\}$
- $\{\neg\text{new\_pred}(\overline{X}), L_{m+1}, L_{m+2}, \ldots, L_n\}$

where $\overline{X}$ represents a vector of variables that appear in both the part $L_1, L_2, \ldots, L_m$ and the part $L_{m+1}, L_{m+2}, \ldots, L_n$. By applying splitting repeatedly, a large clause can be split into several short clauses.

The splitting idea helps sometimes. Table 3.2 shows two examples. The first two problems are original problems; they contain some clauses of 4 or 5 literals. The last two problems in the table contains split clauses. All clauses of 4 or 5 literals are split into clauses containing 3 literals.

Chapter 4

Predicate Replacement

For some theorems, finding their proofs requires a huge search space if no guidance is provided from the user. With appropriate guidance, the search space may be much reduced and the prover is less likely to get lost.

One typical example of this kind is set theory [BLM+86]. Some "natural deduction" system theorem provers can prove set theory theorems quickly by simulating a human's capability of replacing predicates by definitions using explicit quantifiers. Unfortunately, these definitions with explicit quantifiers cannot be used directly in CLIN since it is a quantifier-free system. An idea of term rewriting with a quantifier-free theorem prover "sprfn" [Pla88] results in an efficient approach in proving these theorems [PP88]. However, the approach is less natural. A special tautology tester is written for "sprfn" solely for this purpose.

In this section, we discuss a method, called predicate replacement, which accepts the user's guidance and helps CLIN do search efficiently. This method is similar to the idea of [PP88] but fits in more naturally in our system. It allows the same effect of replacing predicates by their definitions in a Skolemized, quantifier-free setting. With this technique, our system can prove set theory theorems easily. Furthermore, the predicate replacement, similar to hyper-linking operations, does not incur any duplication by combination.

Our method has another advantage. It avoids exponential increase in length of clauses by direct substitutions. This has the same virtue as the structure preserving translation described in [PG86]. The use of predicate replacement is not restricted to definition replacements. It can be used to replace predicates by predicates in any clause. In this way, the user may convey his or her intuition or knowledge about how to prove the underlying problem to the system through replace rules. The rules then direct the system to do guided search.

4.1 Descriptions

In more general terms, predicate replacement permits us to add instances without general search. It permits information to propagate more quickly than by hyper-linking. We may consider predicate replacement like learned responses, while hyper-linking is like general search or unguided thought used in the absence of specific knowledge of what to do. The replace rules used by predicate replacement may be viewed as clause production rules, that is, production rules in a clausal setting. Another way to look at it is to view predicate replacement as focusing attention on objects that have already been constructed. Predicate replacement usually permits a large number of inferences that do not construct new objects. Only when these are exhausted, are new terms constructed by hyper-linking. This seems similar to the general philosophy of our prover, namely, to separate the process of combining literals from the process of reasoning about the instances already obtained, and also seems to correspond to human problem solving methods. Sometimes predicate replacement constructs new terms, but this can be done in a controlled way.

Suppose we have defined a predicate p(X) as (q(X) $\wedge$ r(X)). We would like to replace p(X) by (q(X) $\wedge$ r(X)) everywhere. We would also like to do this if p appears negated in a clause. Thus $\neg$p(a) should be replaced by $\neg$(q(a) $\wedge$ r(a)). Note that the definition of p(X) may be expressed by the formula

$$\forall X\,(p(X) \equiv (q(X) \wedge r(X)))$$

This corresponds to the following clauses:

1. {$\neg$p(X), q(X)}
2. {$\neg$p(X), r(X)}
3. {p(X), $\neg$q(X), $\neg$r(X)}

Suppose the clause {$\neg$s(b), p(f(b))} is generated. We get the effect of replacing p(f(b)) by its definition, by adding the instances {$\neg$p(f(b)), q(f(b))} and {$\neg$p(f(b)), r(f(b))} to the set of clauses. These clauses logically imply the desired result ($\neg$s(b) $\vee$ (q(f(b)) $\wedge$ r(f(b))), which is obtained by replacing p(f(b)) in the clause {$\neg$s(b), p(f(b))} by its definition q(f(b)) $\wedge$ r(f(b)). Similarly, if a clause containing $\neg$p(c) is generated, then we get the effect of replacing p(c) by its definition, by adding the instance {p(c), $\neg$q(c), $\neg$r(c)}. Then it may be necessary to further replace q(c) and r(c) by their definitions, if they exist. We call the literals containing predicate symbols p or $\neg$p in clauses 1, 2, and 3 above the *distinguished* *literals* since they are the literals being defined. They are treated differently than the other literals in the clause.

Definition. If C = $\{L_1, \ldots, L_m\}$ is a clause in a set S of clauses, and $\{L_1, \ldots, L_k\}$ are the distinguished literals of C where $1 \leq k \leq m$, then a *replacement* of C in S is a set $\{(L_1, M_1), \ldots, (L_k, M_k)\}$ of links in S such that there exists a substitution Θ such that $L_i\Theta$ and $M_i\Theta$ are complementary for all i, $1 \leq i \leq k$. We may have to rename the variables in L_i or M_i so that L_i and M_i have no common variables. Θ allows the variables in M_i to be substituted only by individual constants. A most general such Θ is called the *substitution* of the replacement and CΘ for this Θ is called the *instance* of the replacement. We call C the *nucleus* of the replacement and we call the M_i (or clauses D_i containing M_i) the *electrons* of the replacement. We represent C by the following notation

$$M_1, M_2, \ldots, M_k \rightarrow L_{k+1}, \ldots, L_m$$

where M_i is $\neg L_i$, for $1 \leq i \leq k$. We call this representation a *replace rule*. Thus $\rightarrow$ represents implication and the distinguished literals are to the left of the arrow.

Note that our definition of a replacement allows more than one literal to be distinguished. In fact, we perform unit literal deletion for replacements as follows. Let C' represent CΘ, and $C' = \{L'_1, \ldots, L'_m\}$. If $\neg L'_i$ is a unit clause or something more general, then L'_i is deleted from C'.

Suppose we have the following replace rules for equality:

1. X = Y $\rightarrow$ X $\subseteq$ Y

2. X = Y $\rightarrow$ Y $\subseteq$ X

3. $\neg$(X = Y) $\rightarrow$ $\neg$(X $\subseteq$ Y), $\neg$(Y $\subseteq$ X)

Let's call them R_1, R_2, and R_3 respectively. Suppose we have a clause

$$\{\neg((a \cup b) = (b \cup a))\}$$

retained in the database. There are no replacements of R_1 and R_2, but one replacement of R_3. The replacement of R_3 is the set of links

$$\{(X = Y, \neg((a \cup b) = (b \cup a)))\}.$$

The substitution Θ of this replacement is

$$\{X \leftarrow (a \cup b),\ Y \leftarrow (b \cup a)\}.$$

The instance of the replacement is

$$\{(a \cup b) = (b \cup a),\ \neg((a \cup b) \subseteq (b \cup a)),\ \neg((b \cup a) \subseteq (a \cup b))\}$$

which is simplified to

$$\{\neg((a \cup b) \subseteq (b \cup a)),\ \neg((b \cup a) \subseteq (a \cup b))\}$$

by unit literal deletion.

There is also a technicality that restricts the use of replace rules. Namely, when we apply a replace rule {L, M, N} with distinguished literal L, generating an instance {L′, M′, N′}, then the literal L′ may not be used as an electron in further replacements. This is because this literal represents something that is being replaced, not a part of a definition, and we only want the definitions themselves to be available for future replacement.

Sometimes a definition involves existential quantifiers. If p(X) is defined as (∃Y q(X, Y)), then we can express this definition by the formula

$$\forall X\,(p(X) \equiv (\exists Y\, q(X, Y)))$$

which after conversion to clause form yields the clauses

1. $\{\neg p(X),\ q(X, f(X))\}$
2. $\{p(X),\ \neg q(X, Y)\}$

where f is a Skolem function. These would be represented in our notation as the rules

1. $p(X) \rightarrow q(X, f(X))$
2. $\neg p(X) \rightarrow \neg q(X, Y)$

The literals containing p or ¬p are the distinguished literals. If a literal p(g(a)) appears in a clause, then the first clause results in the instance {¬p(g(a)), q(g(a), f(g(a)))} being added by replacement. If a literal ¬p(c) appears in a clause, then a replacement with the second clause as a nucleus results in the instance {p(c), ¬q(c, Y)} being added to the set of clauses.

Suppose we have a definition. Here are the steps for converting the definition into replace rules:

1. Transform the definition to clauses. The definition is generally written in first-order formula. We may do skolemization to take care of existential quantifiers. For more details about the transformation of first-order formula to clause form, see [CL73, Lov78].
2. For each clause C, we pick distinguished literals, negate them, and place them at the left-hand side of $\rightarrow$. The remaining literals are placed at the right-hand side of $\rightarrow$. Usually, literals containing the predicate being defined are made into distinguished literals.

Although we have used definitions to explain the idea of replace rules, their use is not restricted to definition replacements. The replace rules can be used to replace predicates by predicates in any clause. Using this idea, the user may convey his or her intuition or knowledge about how to prove the underlying theorem by selecting appropriate literals to be distinguished literals, negating them, and putting them at the left-hand side of $\rightarrow$. The remaining literals are put at the right-hand side of $\rightarrow$. These distinguished literals are then used to direct CLIN to do the search the user anticipates.

4.2 Context Literals

Consider the following situation. Suppose we have the replace rule

$$\mathrm{X} \geq \mathrm{Z},\ \mathrm{Y} \geq \mathrm{Z} \rightarrow \mathrm{X} \geq \mathrm{Y},\ \mathrm{Y} \geq \mathrm{X}$$

and the following clauses:

- C_1: {p ≥ t, t ≥ s}
- C_2: {r ≥ t, t ≥ q}
- C_3: {¬(p ≥ r)}
- C_4: {¬(r ≥ p)}

From the replace rule, we obtain the following clause:

- $\{\neg(p \geq t), \neg(r \geq t), p \geq r, r \geq p\}$

This clause and clauses C_1 through C_4 are satisfiable. This does not make sense since it is true that either $p \geq r$ or $r \geq p$ and this fact contradicts clauses C_3 and C_4. Since

$$X \geq Y \vee Y \geq X$$

is a tautology, we can delete the instances of the distinguished literals. If we do so, we obtain the desired clause from the replace rule, that is

$$p \geq r \vee r \geq p$$

The distinguished literals $X \geq Z$ and $Y \geq Z$ in this case are called *context literals*. In general, a context literal is a distinguished literal that may be deleted from an instance of a replace rule.

By adding context literals appropriately, finding the instances for all distinguished literals could be more restricted. Therefore, the number of instances generated from a replace rule may be much reduced. We will use context literals in proving temporal logic theorems and some hard theorems.

4.3 Termination of Replacement

We have not addressed the issue of termination. It is possible that the rules we have chosen cause the generation of instances to fail to terminate. This problem can be addressed with a priority system that will be described later. However, we do give some general conditions guaranteeing termination.

Here we apply some ideas similar to those used to show termination of term rewriting systems [Der82] to prove termination of replacements.

Definition. An ordering $\succeq$ on literals is called a *replacement ordering* if this ordering is reflexive and transitive and has the property that if $L \succeq M$ then for all substitutions Θ, $L\Theta \succeq M\Theta$. Also, we require that for any literal L, the set of M such that $L \succeq M$, must be finite.

For example, we can say that L $\succeq$ M if the size of L is at least as large as that of M, and if each variable in M occurs at least as many times in L as in M. Thus p(X, f(X)) $\succeq$ q(X, X) but we don't have p(X, f(Y)) $\succeq$ q(X, X). The reason is that the substitution of a large term for X could make q larger than p.

Definition. A set R of replace rules satisfies the *termination condition* for a replacement ordering $\succeq$ if for all rules r in R, for all literals L in r, if L is not a distinguished literal of r then there is a distinguished literal M of r such that M $\succeq$ L.

Theorem 4.1 Let R be a set of replace rules and $\succeq$ be a replacement ordering such that R satisfies the termination condition for $\succeq$. Then replacement using R always terminates.

Proof. Let S be the (finite) set of clauses to which replacement is applied. Whenever an instance r$'$ of a replace rule r is generated, for every literal L$'$ in r$'$, there exists a literal L in S such that L $\succeq$ L$'$. This can be shown by induction on the number of replacements. Since for each L in S, the set of L$'$ such that L $\succeq$ L$'$, is finite, there are only finitely many literals that can appear in instances r$'$ obtained by replacements. Thus the number of instances r$'$ is finite, and replacement will eventually stop.

Note that replacement may terminate in cases where the usual replacement of predicates by their definitions would not. For example, if a predicate p(X, Y) is defined to be p(Y, X), then replacement of p(a, b) by its definition would successively yield p(b, a), and p(a, b), and p(b, a), ad infinitum. However, the replace rules would only generate the instances {¬p(a, b), p(b, a)} and {¬p(b, a), p(a, b)}, and would then stop.

In general, a set of replace rules will terminate if the distinguished literals are larger than or equal to the other literals of each rule, in some reasonable ordering. Rules like p(X, Y) $\rightarrow$ p(Y, X) do not cause a problem; it is only rules like p(X) $\rightarrow$ p(f(X)) that cause a problem, because the non-distinguished literal is larger than the distinguished literal in any replacement ordering.

4.4 An Example

We give an example here to illustrate the use of replace rules. Consider the following set theorem

$$\forall X, Y ((X \cup Y) = (Y \cup X))$$

which states that the union operation is commutative. The replace rules we need are listed below:

1. $X = Y \to X \subseteq Y$
2. $X = Y \to Y \subseteq X$
3. $\neg(X = Y) \to \neg(X \subseteq Y), \neg(Y \subseteq X)$
4. $X \subseteq Y \to \neg(Z \in X), Z \in Y$
5. $\neg(X \subseteq Y) \to f(X, Y) \in X$
6. $\neg(X \subseteq Y) \to \neg(f(X, Y) \in Y)$
7. $Z \in (X \cup Y) \to Z \in X, Z \in Y$
8. $\neg(Z \in (X \cup Y)) \to \neg(Z \in X)$
9. $\neg(Z \in (X \cup Y)) \to \neg(Z \in Y)$

These rules are obtained easily from the definitions of $=$, $\subseteq$, and $\cup$. We transform the negation of the theorem to the clause

- C_1: $\{\neg((a \cup b) = (b \cup a))\}$

From C_1, we derive

- C_2: $\{\neg((a \cup b) \subseteq (b \cup a)), \neg((b \cup a) \subseteq (a \cup b))\}$

From C_2, we derive the following clauses in sequence:

- C_3: $\{(a \cup b) \subseteq (b \cup a), f((a \cup b), (b \cup a)) \in (a \cup b)\}$
- C_4: $\{(a \cup b) \subseteq (b \cup a), \neg(f((a \cup b), (b \cup a)) \in (b \cup a))\}$
- C_5: $\{(b \cup a) \subseteq (a \cup b), f((b \cup a), (a \cup b)) \in (b \cup a)\}$
- C_6: $\{(b \cup a) \subseteq (a \cup b), \neg(f((b \cup a), (a \cup b)) \in (a \cup b))\}$

From C_3, we derive

- C_7: $\{\neg(f((a \cup b), (b \cup a)) \in (a \cup b)), f((a \cup b), (b \cup a)) \in a, f((a \cup b), (b \cup a)) \in b\}$

From C_4, we derive

- C_8: $\{f((a \cup b), (b \cup a)) \in (b \cup a), \neg(f((b \cup a), (a \cup b)) \in b\}$
- C_9: $\{f((a \cup b), (b \cup a)) \in (b \cup a), \neg(f((b \cup a), (a \cup b)) \in a\}$

From C_5, we derive

- C_{10}: $\{\neg(f((b \cup a), (a \cup b)) \in (b \cup a)), f((b \cup a), (a \cup b)) \in b, f((b \cup a), (a \cup b)) \in a\}$

From C_6, we derive

- C_{11}: $\{f((b \cup a), (a \cup b)) \in (a \cup b), \neg(f((b \cup a), (a \cup b)) \in a\}$
- C_{12}: $\{f((b \cup a), (a \cup b)) \in (a \cup b), \neg(f((b \cup a), (a \cup b)) \in b\}$

The propositional calculus prover decides that the above 12 clauses, C_1 through C_{12}, are unsatisfiable. Therefore, we have proved the theorem. That is, the union operator is commutative.

4.5 Implementation

CLIN performs predicate replacement after each round of hyper-linking, and once before the first round of hyper-linking. We implemented two kinds of replace rules: repeated replace rules and once replace rules. Instances of the repeated replace rules are generated repeatedly until no new instances can be generated. Then once replace rules are applied only once. Then instances of the repeated replace rules are generated repeatedly until no new instances can be generated.

The separation of replace rules provides better control over the number of instances to be generated. For example, we may treat the replace rules converted from predicate definitions as repeated replace rules. It makes sense to allow these replace rules to be applied as far as possible. We may also use a rule whose distinguished literals are larger than the literals at the right-hand side as a repeated replace rule. However, we may want a replace rule concerning transitivity to be a once replace rule. This replace rule may generate a lot of instances if it is allowed to be applied iteratively. We may also use a rule whose distinguished literals are smaller than the literals at the right-hand side as a once replace rule.

We avoid backtracking for ground distinguished literals. We also match ground distinguished literals of a replace rule first since we don't backtrack for them.

The user is allowed to choose one of the following three options to further cut down the search space:

1. Ground substitution. The electrons to be linked with distinguished literals are to be ground. However, this does not apply to context literals.
2. Ground matched literals. The literals contained in the instance of a replace rule and corresponding to distinguished literals are required to be ground.
3. Ground restriction. The instance of a replace rule has to be ground.

The ground substitution is very useful. With appropriate replace rules, many proofs can be obtained by ground substitution.

4.5.1 Set Theorems

The predicate replacement yields spectacular improvements in efficiency in set theory problems. We list some proved set theorems in Table 4.1. Most of the theorems in this table can't be proved without using replace rules. In the table, $\bar{\cap}$ means disjoint, $\neg$ means complement, $m(X)$ means X is an element, $meq(X,Y)$ means X and Y are identical elements, $< X, Y >$ means an ordered pair, $opp(X)$ means that X is an ordered pair predicate. The others are conventional symbols in set theory. Most of the replace rules used in proving these set theorems are for definition replacements. Replace rules for non-definition replacements are used for proving some hard theorems and temporal logic theorems that will be discussed later.

no	problem	cpu(sec)
1	$\forall X, Y(X \cup Y = Y \cup X)$	16.650
2	$\forall X, Y(X \cap Y = Y \cap X)$	19.233
3	$\forall X(X \cup X = X)$	15.817
4	$\forall X(X \cap X = X)$	14.233
5	$\forall X(X \cup \neg X = universe)$	14.883
6	$\forall X(X \cap \neg X = \emptyset)$	15.417
7	$\neg universe = \emptyset$	13.167
8	$\neg \emptyset = universe$	13.083
9	$\forall X(\neg\neg X = X)$	16.317
10	$\forall X(X \cup \emptyset = X)$	15.883
11	$\forall X(X \cap universe = X)$	14.783
12	$\forall X(X \cup universe = universe)$	13.383
13	$\forall X(X \cap \emptyset = \emptyset)$	13.483
14	$\forall X, Y, Z((X \cup Y) \cup Z = X \cup (Y \cup Z))$	18.083
15	$\forall X, Y, Z((X \cap Y) \cap Z = X \cap (Y \cap Z))$	27.417
16	$\forall X, Y(X \subseteq Y \to X \cap Y = X)$	14.567
17	$\forall X, Y(\neg(X \cup Y) = \neg X \cap \neg Y)$	20.917
18	$\forall X, Y(\neg(X \cap Y) = \neg X \cup \neg Y)$	18.550
19	$\forall X, Y((X \cup Y) \cap (X \cup \neg Y) = X)$	19.500
20	$\forall X, Y(X - Y = X \cap \neg Y)$	19.133
21	$\forall X, Y, Z(X \cap (Y \cup Z) = X \cap Y \cup X \cap Z)$	22.250
22	$\forall X, Y, Z(X \cup Y \cap Z = (X \cup Y) \cap (X \cup Z))$	21.767
23	$\forall X(\emptyset \subseteq X)$	0.500
24	$\forall X, Y, Z(X \subseteq Y \wedge Y \subseteq Z \to X \subseteq Z)$	19.217
25	$\forall X, Y(X \subseteq Y \to X \in 2^{Y})$	0.683
26	$\forall X, Y(X \bar{\cap} Y \to X \cap Y = \emptyset)$	21.217
27	$\forall X, Y(X \subseteq (X \cup Y))$	1.700
28	$\forall X, Y((X - Y) \subseteq X)$	1.733
29	$\forall X, Y, Z(X \subseteq (Y \cap Z) \to X \subseteq Y \wedge X \subseteq Z)$	13.350
30	$\forall X, Y(2^{X \cap Y} = 2^{X} \cap 2^{Y})$	25.733
31	$\forall X, Y, Z(X \times (Y \cap Z) \subseteq ((X \times Y) \cap (X \times Z)))$	21.900
32	$\forall X, Y, Z, W((X \subseteq Y) \cap (Z \subseteq W) \to X \times Z \subseteq Y \times W)$	40.033
33	$\forall X, Y, Z, W(meq(a, b) \wedge meq(c, d) \to < a, c > = < b, d >)$	1.083
34	$\forall X, Y, Z(X = < Y, Z > \to opp(X))$	4.017
35	$\forall X, Y(m(X) \wedge m(Y) \to \{X\} \subseteq \{X, Y\})$	21.933
36	$\forall X, Y(m(X) \wedge m(Y) \to \{X, Y\} = \{Y, X\})$	0.867

Table 4.1: Times for sample set theorems

Chapter 5

The Sliding Priority Method

As we mentioned earlier, the hyper-linking strategy prover may require too much memory. We also pointed out that there are two remedies to this problem: making the stored search tree thin or shallow. We have presented the small proof checking as a way to make the stored tree shallow. Now we turn to the problem of how to make it thin.

Making the search tree thin is equivalent to reducing the branching factor of nodes in the search tree. For a search tree of N levels with average branching factor B, the total amount of storage required for this tree is proportional to $\frac{B^{N}-1}{B-1}$. Therefore, the ratio of space needed for storing two search trees of N levels but with average branching factors B_1 and B_2 respectively is $\frac{{B_1}^{N}-1}{{B_2}^{N}-1} \times \frac{B_2-1}{B_1-1}$. If B_1 is 6 and B_2 is 5 and N is 5, for example, the second tree saves about 50% of the storage used by the first tree.

Three strategies, based on priorities assigned to each node, are commonly used in the area of Artificial Intelligence for the reduction of the branching factor. The ordering strategy considers more important nodes first, and delays less important ones. The restriction strategy prohibits certain nodes to be produced. The pruning strategy deletes selected existing nodes. In order to make these strategies work, one has to set a priority bound. Two difficulties occur. The priority bound may be too small to solve the given problem. We call this property *incompleteness*. On the other hand, the priority bound may be too large. Too many nodes are produced and use up the storage. We call this property *overwhelmingness*. Furthermore, a priority bound has no relationship with the space to be used. We have no idea of how many nodes are going to be generated when we set the priority bound. Therefore, we have no control over how much space is used.

We propose a sliding priority method to solve both difficulties of incompleteness and overwhelmingness. We don't have to set a priority bound; it is set automatically. The search is performed under the permitted space quota. If the problem can't be solved with the current space quota, the space quota is increased automatically. In this way, we always solve a given problem with, in some sense, minimum space. A similar idea has been proposed in another context in [NP90]. The same effect could also be obtained by best-first search [OMW76], but our method permits a simpler implementation.

The idea of sliding priority has been proven very effective. It enabled us to get a number of proofs that couldn't be obtained without sliding priority. The idea also makes CLIN more self-guiding, requiring less input and insight from the user. Furthermore, the idea can be easily applied to any other prover using breadth-first search strategies, such as the resolution method [Rob65] and Gilmore's procedure [Gil60].

5.1 Descriptions

5.1.1 Basic Idea

Suppose we decide in advance that we only want to store b clauses at one time. We call b the *work bound*. Each clause has a priority, which is an integer giving its size and/or some other measure of complexity. We assume that for each integer n, there are only finitely many clauses having priority bounded by n. Suppose we want to set a priority bound so that no more than b clauses will be retained. Clauses with a priority that is too large will be deleted. However, we do not know in advance how large the priority bound should be. We assume the priority bound to be infinity at the beginning. Suppose a clause C is generated. If the number of clauses stored is less than b, we store C. If the number of clauses stored is not less than b and the priority value of C is greater than or equal to the priority bound, we discard C. If the number of clauses stored is not less than b and the priority value of C is less than the priority bound, we delete one clause having largest priority value, store C, and set the priority bound to be the largest priority value of the stored clauses. If more than one clause having largest priority value, we may use other criteria, such as preferring clauses having fewer distinct variables, to delete one of them. During this generation of clauses, we try to find a proof from these stored clauses periodically. If a proof is found, we are done. Otherwise we keep on generating more clauses. Suppose at some point we can't obtain any new clauses; we call this a *session* with work bound b. In this case the work bound is increased and a new session is started. This is called sliding priority because the priority bound keeps decreasing, or "sliding," in each session. Eventually, the work bound will be large enough to obtain a proof, if the strategy is complete. Notice that the system is automatically setting the priority to insure that the work bound is not exceeded. The number of clauses that are generated and not deleted right away in a session is called the *actual work* for work bound b. If a clause is deleted when it is generated, we don't count it in the actual work.

Let's illustrate the above idea by an example. Suppose the work bound is 3. Suppose we generate clauses of priority value 4, 3, 1, and 1. After these four clauses are generated, we know that all clauses of size greater than 3 cannot be retained, since there are three clauses with size 3 or less. Therefore, the priority bound is 3. Next, we generate clauses of size 6 and 5, which are deleted because their priorities are larger than 3. Finally, we generate a clause of priority 2. The priority bound is now 2, and the clause of priority 3 is deleted. Suppose there are no more new clauses generated. We end up with a session with work bound 3. The total number of clauses generated is 7, and the actual work is 5. The clauses with priority value 6 and 5 have never been stored; they were deleted right away.

5.1.2 A Refinement

The idea we presented previously is attractive. However, there is a problem with sliding priority, namely, that there is no strong relationship between the work bound and the actual work for a session of search. For example, if the first clause generated has priority 100, the second has priority 99, the third has priority 98, ..., and so on, the actual work can be 100 for a work bound of 3 (or even 1). If the first clause has priority 10000, the second clause has priority 9999, and so on, the actual work can be 10000 for a work bound of 1. There is no upper bound to the actual work for a given work bound. Furthermore, the work bound does not reflect the space available for use. to be used.

We solve this problem by controlling the sum of the priorities of the clauses stored, rather than the number of clauses stored. Presumably it takes more work to generate and store a clause of size 100 than a clause of size 10. If the work bound is 3, then a clause of size 100 (or 10) would be deleted immediately. We redefine the work bound to be the sum of the priorities of the clauses that we want to store at one time, and the actual work to be the sum of the priorities of the clauses ever stored. We set the priority bound P_b to be the work bound b initially. Let W be the sum of the priorities of the clauses currently stored. Suppose a clause C with priority value P_c is generated. If P_c is not greater than P_b and P_c + W $\leq$ b we store C. If P_c is greater than P_b, we discard C. If P_c is not greater than P_b and P_c + W $>$ b, we store C, delete one clause with the largest priority value, and set P_b to be the largest priority value of the stored clauses.

With this approach, we can give a good upper bound for the actual work for a given work bound. Suppose the work bound is b. Then the worst case for actual work will first generate a clause of size b, then a clause of size b-1, then a clause of size b-2, ..., then 2 clauses of size b/2, then 2 clauses of size b/2 - 1, ...then 3 clauses of size b/3, et cetera. The total number of clauses stored is then about b/2 + 2(b/2 - b/3) + 3(b/3 - b/4) + ...+

(b - 1)(b/(b-1) - b/b). This simplifies to b/2 + b/3 + b/4 + ...+ b/b, or, b(1/2 + 1/3 + 1/4 + ...+ 1/b), which is proportional to $b \log b$. Thus there is a reasonable upper bound for the total number of clauses stored in a round. The sum of the priorities of these clauses, that is, the actual work, is easily seen to be proportional to b^2.

This approach makes more sense. It makes the space to be used proportional to the work bound if the priority function is proportional to clause size. Therefore, it better controls the use of memory. In this way, we always solve a given problem with minimum space plus the deviation of the amount that the work bound is increased each time.

Let's work out the previous example with this refined idea. The priority bound is 3 at the beginning. The clause with priority value 4 is discarded. After the first four clauses are generated, the priority bound is still 3. We discard the next two clauses with priority 6 and 5 respectively. Finally, we store the clause with priority 2. The number of clauses stored is 4. The actual work is $3 + 1 + 1 + 2$, which is 7.

5.1.3 A Generalization

It is possible to have a more general sliding priority than this. Suppose there are two parameters for a clause, such as its size and its depth of nesting of function symbols. Suppose we want to set bounds on the size and depth to insure that a work bound is not exceeded. However, we do not restrict the relationship between size and depth. This gives a kind of two-dimensional sliding priority. Something can still be done in this case. For example, suppose the work bound is 3. To make the illustration simpler, suppose that we only count each clause as one unit of work, regardless of its size and depth. Suppose 3 clauses have been generated, and all have size less than 10 and depth less than 3. Then a clause of size 11 and depth 5 cannot be retained. The reason is that this clause can only be retained if the size and depth bounds are at least 11 and 5, respectively, and in this case, four clauses will be retained, exceeding the work bound. In the same way, any clause with size at least 10 and depth at least 3 should be deleted.

In general, we may have a partial ordering $\prec$ on clauses such that if C $\preceq$ D, then if D is retained, C should be also. For example, if C has smaller size and depth than D, then C $\preceq$ D in our above example. Or, there may be 3 measures of complexity of a clause, and we say C $\preceq$ D if all measures for D are at least as large as for C. Thus this formalism generalizes one, two, and higher dimensional sliding priorities, and even priorities computed as some linear sum of measures of C. In this case, we can delete a clause D if the number of clauses C with C $\preceq$ D, is at least b. Or, if we associate a non-unit work w(C) with each clause C, we can delete D if the sum of w(C), for C $\preceq$ D, is at least b. However, for this system we need to consider the issue of whether $\preceq$ is efficiently computable, and whether the condition for deleting a clause is efficiently computable.

5.2 Implementation

We have the refined idea described in section 5.1.2 implemented in CLIN. For simplicity, we implement it in a slightly different way. Suppose a new clause C with priority value P_c is generated. If P_c is not greater than P_b and $P_c + W$ is not greater than b, we store C. If P_c is greater than P_b, we discard C. If P_c is not greater than P_b and $P_c + W$ is greater than b, we store C and discard all the clauses having priority value P_b and decrease the priority bound P_b by 1. This avoids the complexity of selecting a clause having priority P_b to be deleted, but slows down the sliding rate of the priority bound.

The initial work bound can be either assigned by users or computed by the system as default. We assume that it is reasonable to set the initial work bound to be proportional to the size of input clauses. The size of a clause is the number of symbols in the clause when it is written as a character string. We double the work bound for each new session. We also delete all the instances generated in the previous session when we start a new session. This means work of the previous session has to be duplicated.

The priority function incorporates four priority measures: size, depth, distance, and number of literals. The size may be the size of the clause or the size of the largest literal in the clause. The size of a literal is the number of symbols in the literal when it is written as a character string. The coefficient for each priority measure can be set by the user. The function can be the linear sum of its priority measures or the maximum of these measures. Let c_i be positive coefficients and p_i be various complexity measures of the clause. The maximum idea is to take the priority to be the maximum of $c_i \times p_i$. The linear sum idea is to take the priority to be the sum of $c_i \times p_i$.

A weighting scheme is also provided. The user may assign terms different weights. For example, sometimes a problem can be solved much easier when only clauses with a small number of variables are used. Therefore it may make sense to allow the user to weight variables more highly than constant symbols when computing the size of a term. The weighting scheme is especially useful when the user has intuition about what terms are to be involved in the proof. In this case, assigning these terms low weights may help a lot.

We order the nuclei by number of non-ground literals: the clauses having smallest number of non-ground literals are used first. This has an advantage that as many nuclei as possible can be used before the time limit is reached since the time spent on a nucleus in hyper-linking is in general proportional to the number of non-ground literals it contains.

When the time limit is exceeded, we try to find a proof from those instances generated so far.

5.3 An Example

We give a real example here for illustration. This example is "ls103" [Sti88]. The axioms are in the following:

1. {un(a, a, b)}
2. {m(Z, Y), ¬s(X, Y), ¬m(Z, X)}
3. {m(f(X, Y), X), s(X, Y)}
4. {s(X, Y), ¬m(f(X, Y), Y)}
5. {s(X, Y), ¬e(X, Y)}
6. {s(Y, X), ¬e(X, Y)}
7. {e(X, Y), ¬s(X, Y), ¬s(Y, X)}
8. {m(W, X), ¬un(X, Y, Z), ¬m(W, Z), m(W, X)}
9. {m(W, Z), ¬un(X, Y, Z), ¬m(W, X)}
10. {m(W, Z), ¬un(X, Y, Z), ¬m(W, Y)}
11. {m(g(X, Y, Z), Z), m(g(X, Y, Z), X), m(g(X, Y, Z), Y), un(X, Y, Z)}
12. {un(X, Y, Z), ¬m(g(X, Y, Z), X), ¬m(g(X, Y, Z), Z)}
13. {un(X, Y, Z), ¬m(g(X, Y, Z), Y), ¬m(g(X, Y, Z), Z)}
14. {¬e(b, a)}

A proof contains the following ground clauses:

1. {¬s(b,a),¬s(a,b)}
2. {s(b,a),¬m(f(b,a),a)}
3. {s(a,b),¬m(f(a,b),b)}
4. {m(f(a,b),b),¬m(f(a,b),a)}

5. {m(f(b,a),a),¬m(f(b,a),b)}

6. {m(f(b,a),b),s(b,a)}

7. {m(f(a,b),a),s(a,b)}

CLIN keeps 133 clauses and uses 1905 units of space without sliding priority. If we use sliding priority and define the priority function to be the size of the clause without counting the symbol ¬, CLIN keeps only 46 clauses and uses 465 units of space with initial work bound 600. The final priority bound set by CLIN is 12, a little bit higher than the optimum 10.

5.4 Complexity Analysis

The precise analysis of the sliding priority method is hard. Many factors are problem dependent, and there is no way to decide these factors by just looking at the syntactic structure of the input clauses. However, we can have some insights from an analysis based on simplified assumptions.

Given a problem P, we assume:

1. It takes T_1 seconds to generate a unit size of clauses. If a clause has size S, then ST_1 of time is needed for generating this clause.
2. It takes T_2 seconds to assert a unit size of clauses.
3. It takes T_3 seconds to delete a unit size of clauses.
4. The priority function is defined to be the size of a clause. With this, the work bound is equal to the available space. This assumption can be easily generalized to any priority function that is associated with the size of a clause.
5. In a session with work bound b, the search space is a function of b, say, f(b).
6. If P can be solved without sliding priority, the space consumed is U.
7. If P can be solved with sliding priority, the initial work bound is V and N sessions are required.
8. The time spent by the contradiction detector on a unit size of clauses is T_4 seconds.

5.4.1 Space

The amount of space needed by CLIN is trivial. It is U without sliding priority, and 2^{N-1}V with sliding priority.

5.4.2 Time

The time complexity for CLIN without sliding priority is

$$U(T_1 + T_2 + T_4)$$

The time complexity for CLIN with sliding priority can be derived as follows. Since there are N sessions, the work bound for each session is V, 2V, ..., 2^{N-2}V, and 2^{N-1}V respectively. The total time needed is then

$$\sum_{n=0}^{N-1}[f(2^n V)T_1 + (2^n V)^2 T_2 + ((2^n V)^2 - 2^n V)T_3 + 2^n V T_4]$$

The above formula is derived from the fact that total units $f(2^n V)$ of clauses are generated in session n, of which $(2^n V)^2$ are asserted, $(2^n V)^2 - 2^n V$ are deleted, and $2^n V$ are presented to the contradiction detector.

More assumptions can be made to simplify the equations.

1. Let T_1 be T.
2. Let $f(b) = Ab^2$.
3. Let $T_2 = BT$.
4. Let $T_2 = T_3$. This is reasonable. In general, assertions are faster than deletions since deletions need indexing which is a kind of searching. However, in our prover, we check duplicates before a clause is asserted, which needs searching too.
5. Let $T_4 = CT$.

Then the time complexity without sliding priority is simplified to

$$UT(1 + B + C)$$

and the time complexity with sliding priority is simplified to

problem	without priority (sec)	with priority (sec)
ex4t1	132.817	4223.300
ex4t2	133.450	3725.716
ex6t1	43.750	562.333
ex6t2	43.917	562.967

Table 5.1: Examples on which the sliding priority method has difficulty

$$V^2T(A+2B)\sum_{n=0}^{N-1} 2^{2n} + VT(C-B)\sum_{n=0}^{N-1} 2^n$$

This can be evaluated to

$$V^2T(A+2B)\frac{2^{2N}-1}{3} + VT(C-B)(2^N-1)$$

which can be reasonably approximated to

$$V^2T(A+2B)2^{2N}/3$$

This time complexity looks bad. However, in practice the actual work is usually much smaller than b^2 for work bound b. Therefore, the term V^2 may be replaced by V^x where x is close to 1.

5.5 Discussion

The sliding priority idea helps CLIN to solve a number of problems. Several problems in our test set that can't be solved without sliding priority are solved with sliding priority. Most of the problems that can be solved without sliding priority can also be solved equally well with sliding priority.

The sliding priority may have trouble if a proof requires big terms to be generated. Some examples are listed in Table 5.1. These problems need big terms for the proof. For example, the largest input clause in "ex4t1" has size of 17, whereas the largest clause that appears in the proof has size of 72. The big terms force the work bound to be updated many times before a proof is found since the initial work bound, which is estimated syntactically from the input clauses, is small. In this case, the term 2^{2N} is big in the formula of time complexity. However, the sliding priority method may run better for certain problems, as shown in Table 5.2. For these problems, the search space is cut down significantly by the

problem	without priority (sec)	with priority (sec)
wos10	884.650	46.200
wos22	111.016	73.383
ls87	190.866	69.400

Table 5.2: Examples on which the sliding priority method runs better

problem	strategy	maximum literal (sec)	clause size (sec)
wos4	[b]	169.650	36.267
wos10	[f, b]	46.250	133.850
wos17	[b, f]	72.000	1003.500
wos22	[UR]	2828.167	73.200
wos26	[b]	1287.500	126.500
wos28	[UR]	1866.783	> 5646.217
ls36	[b, f]	257.750	57.017
ls108	[b]	490.033	931.417
ls112	[b]	1244.283	> 7086.250
schubert	[UR]	428.433	135.017

Table 5.3: Comparison of two measures of size for priority

sliding priority method and the clauses thrown away do not make any contribution to the proof. In addition, some problems can't be solved unless sliding priority is used, see Table A.1 in the Appendix A.

We have found that the sliding priority method is very sensitive to the choice of priority measure. For example, some problems that are easily solved with clause size may be hard for the maximum literal size, and vice versa. See Table 5.3. We have also used number of literals in a clause and the relevance as priority measures. They help some cases but hurt other cases. In some tests, we have compared whether the maximum of a collection of priority measures is a better measure than the linear sum. The former seems better for most cases.

The sliding priority idea can be used to control predicate replacement. Some replace rules may not terminate. To handle this, we can use sliding priority and let each instance of a replace rule contribute to the work for the prover. When the work bound is exceeded, the replace rule instances with a high priority will be deleted. Since there are only a finite number of instances with a bounded priority, the replacement round will eventually stop.

The low-level heuristics that a human uses are applied in the sliding priority method.

The priority function is defined to be related to the size of the clause. The user is allowed to weight variables more highly than constant symbols when computing the size of a term. These imply the preference of small terms and terms containing a small number of variables respectively. The effectiveness of the sliding priority method gives evidence that these low-level heuristics help in finding the proof. This might suggest that CLIN provides a rational reconstruction of human theorem proving activity.

For some problems we found an amusing dependence of the time taken to find a proof on the time limit allocated to the system. They were proven more quickly with a small time bound than with a large one. This is a benefit from the ordering of the nuclei used for hyper-linking. For these problems, the instances generated from the nuclei with many non-ground literals do not contribute to the proof.

Chapter 6

The Top-Level Supervisor

It is important that a theorem prover be usable by users who have little knowledge of theorem proving strategies. In this chapter, we present a top-level supervisor which frees the user from the necessity of understanding our system or the merits of different strategies in order to use it. This supervisor embodies much of our expert knowledge about how to prove theorems. All the user has to do is input his problem and time bounds; the supervisor will direct CLIN to find a good proof for the user by following the expert knowledge. We have found that this top-level supervisor not only eliminates the need for tedious bookkeeping by a human in trying a sequence of strategies, but also often reduces the total time required to find a proof.

6.1 Motivation

A theorem prover may be quite complex. It may use many different strategies for the sake of efficiency. Some strategies may not even be complete. Therefore, it is not uncommon for a theorem prover to accommodate a variety of options for users. This causes many difficulties which limit the usefulness of a theorem prover. First, users are required to be familiar with techniques of theorem proving. Otherwise, it is impossible for them to select from options. This intimidates the users outside the theorem proving area. Second, an incomplete strategy may be very efficient if it is applicable. However, there is no way to know if it is applicable for a given problem before trying it. This inhibits the user from using such strategies. Third, the user can't concentrate on the problem they want to solve. Fourth, performance of different strategies may vary to a large extent. It is hard to choose the right options in order to obtain good proofs.

A friendly top-level supervisor is the solution we propose to the above difficulties.

The supervisor serves as an interface between the user and CLIN. This interface accepts the user's problems and directs the system to find proofs without intervention from the user. This autonomy is achieved by giving expert knowledge about how to prove theorems to the supervisor.

6.2 Implementation of CLIN

We give a brief description about the implementation of CLIN here. Then we will describe how the top-level supervisor interacts with CLIN later.

Given a problem stated in proper format, CLIN first translates the statements in the input file which contains the problem to internal forms. The translation procedure performs

1. syntactical checking. It aborts the problem if a syntax error in the input is detected.
2. option settings. It executes commands to set up the environment desired.
3. calculation of desired data structures for clauses. Clauses are stored in a form convenient for duplicate checking, grounding, and occurs check.
4. assignment of priority values to all clauses.
5. set-up of support sets. Clauses are classified to be forward supported, backward supported, or user supported appropriately.

After the translation procedure, CLIN performs a pre-check. The check carries out the following in order:

1. some trivial terminating conditions including
   - empty input set. In this case, the input problem is satisfiable.
   - empty clause. In this case, the problem is unsatisfiable.
   - positive and negative clauses. If the input problem does not contain either a positive or a negative clause, then the problem is satisfiable.
2. unit simplification.
3. propositional unsatisfiability test.
4. small proof checking.

This pre-check is to find an “early termination” without any hyper-link operations.

Then CLIN performs the sequence of hyper-linking, unit simplification, propositional unsatisfiability test, and small proof checking iteratively until either the problem is detected satisfiable or unsatisfiable, or the time limit is exceeded.

6.3 Implementation of the Top-Level Supervisor

6.3.1 Expert Knowledge

The expert knowledge given to the supervisor is obtained from our various experimentations with CLIN. From our experience, we found that the combination of user support strategy and UR resolution is pretty good in general. It is goal-oriented and only generates unit clauses. UR resolution itself is also good, especially if the user does not specify the user support set. These two strategies are not complete but do well if they are applicable. Backward chaining and forward chaining interleaved is a complete strategy. It is able to take care of a problem that can't be proved by an incomplete strategy. However, we have to control it so that it won't hurt too much the other problems that can be solved with incomplete strategies. Similarly for the forward chaining and backward chaining interleaved strategy. User support strategy focuses attention on the theorem, so it is worthwhile to try it. However, it may generate too many clauses. Often, a theorem has the form $P \rightarrow Q$. The negation of the theorem thus consists of P and $\neg Q$. It may help cut down the search space if we use $\neg Q$ only in the user support set. Unit clauses are very useful in theorem proving. We also consider a strategy to use the set of unit clauses as a user support set.

It may help to restrict the number of literals in a clause generated from hyper-linking. This encourages embedded unit literal deletions, which implies a preference for unit clauses. However, this restriction is not a complete strategy.

Conceptually, we provide the following rules in the knowledge base for the top-level supervisor. These rules reflects our experience stated above.

1. For UR resolution:
   -1:1:[uUR]
   -1:1:[UR]
2. For unlimited literal bound:
   -1:1:[f, b]
   -1:1:[b, f]
   -1:0.5:[n]

-1:1:[u]
-1:0.5:[o]

3. For literal bound 2:
   2:0.5:[f, b]
   2:0.5:[b, f]
   2:0.25:[n]
   2:0.5:[u]
   2:0.25:[o]

Each rule is a triplet x:y:z which specifies literal bound x, time coefficient y, and support pattern z. Literal bound x specifies that the number of literals in a clause to be retained should be less than or equal to x. In particular, we use "-1" to mean no literal bound. Time coefficient y is to scale the time limit allocated for a run of certain strategy. If it is 0.5 and the time limit is 200 seconds for a run then the actual time bound is 100 seconds for this run. The use of a time coefficient is intuitively reasonable. For example, a strategy with literal bound 2 may use less time than the same strategy without literal bound since the search space with literal bound 2 is not larger than the search space without literal bound. Support pattern z specifies what support strategies are to be used in a certain run. We use "u" to represent user support strategy, "UR" to represent UR resolution strategy, "b" to represent backward support strategy, "f" to represent forward support strategy, "o" to represent user support strategy with all unit clauses as user support set, and "n" to represent user support strategy with negative clauses in a user support set to be actual user support set. Two strategies separated by a comma means they are interleaved. Two strategies concatenated together means they are in effect at the same time.

We may add more knowledge or take some knowledge away without any difficulties. This knowledge is used as data, not procedural codes.

Some options are fixed in the top-level supervisor. For example, we use the sliding priority method since it restricts the amount of space to be used without losing completeness. We fix the priority function to be dependent only on size of clauses. We also set the proof size bound to infinity for the small proof checking.

6.3.2 Time Allocation

The user is allowed to specify the least time bound, the greatest time bound, and the time limit multiple to the top-level supervisor. The defaults are 200 seconds, 800 seconds, and 2 respectively. The time limit starts with the least time bound, increases by multiples of the time limit multiple each time, and should not exceed the greatest time bound. The actual time limit is the time limit scaled by the time coefficient. The actual time bound is used to control each run of a problem.

6.3.3 The Algorithm

Given a problem P, the top-level supervisor tries all strategies by following in order one rule in the knowledge base after another. If P is detected unsatisfiable, then P is solved. Otherwise the supervisor increases the time limit and tries all strategies again. The algorithm is described as follows:

```
supervisor(P)
    set the time limit T_L to be the least time bound;
    set terminated as FALSE;
    while terminated is FALSE
        while terminated is FALSE and there are rules left
            read next rule x:y:z;
            set literal bound x;
            set actual time bound T_A to be T_L×y;
            set support strategy z;
            try to prove P;
            if P is detected unsatisfiable
            then assign TRUE to terminated;
        if terminated is FALSE
        then double T_L;
            if T_L is greater than the greatest time bound
            then assign TRUE to terminated
            else prepare to read the first rule;
end supervisor(P).
```

6.4 Improvements

The algorithm in the previous section can be improved. First of all, we want to stop if the problem P is detected satisfiable. Note that we are able to make a correct conclusion about the input problem if a complete support strategy is used. We provide a list of complete strategies in the knowledge base something like

- complete_strategy([f, b])

- complete_strategy([b, f])

If the problem P is detected satisfiable and the underlying support strategy is complete, then we stop the supervisor.

Another improvement can be done with an incomplete support strategy R. If CLIN detects satisfiable for the problem P in a time limit T with support strategy R, then we don't need to try P with support strategy R in any time limit T′ that is greater than T. Also, we delete all instances if the set of input clauses is a Horn set since unit resolution is complete for this case.

A rule can be skipped if the specified support set is not provided. For example, if user support set is not provided then we don't try [uUR] or [u] at all. Similarly, we don't try [n] if there are no negative clauses in a user support set. We also don't try [o] if there are no unit clauses.

We only need to do the pre-check once, although the given problem P may be tried many times. The reason is that the pre-check concerns input clauses only.

6.5 Batch Mode

We also have a batch mode for the top-level supervisor, in which a list of problems can all be run together. In this mode, the first strategy with the first time bound is tried on all problems, then the second strategy with the first time bound is tried on all problems not already solved, and so on. In this way, the theorems that can be proven quickly are solved early, and the theorems that are difficult do not delay finding the easy proofs.

Basically, we maintain two data structures. One is for keeping the problems yet to be tried. The other records which strategies may be skipped for which problems. The algorithm is as follows, where L is a list of problems:

```
supervisor(L)
        set the time limit T_L to be the least time bound;
        set terminated as FALSE;
        while terminated is FALSE
                while terminated is FALSE and there are rules left
                        read next rule x:y:z;
                        set literal bound x;
                        set actual time bound T_A to be T_L×y;
```

```
            set support strategy z;
            prepare to read the first problem in L;
            while there are unsolved problems left in L
                read next problem P;
                try to prove P;
                if P is detected unsatisfiable
                then delete P from L;
                if P is detected satisfiable and the underlying strategy is complete
                then delete P from L;
                if P is detected satisfiable and the underlying strategy is incomplete
                then do not try P with this strategy any more;
            if L is empty
            then assign TRUE to terminated
            else prepare to read the first problem in L;
        if terminated is FALSE
        then double T_L;
            if T_L is greater than the greatest time bound
            then assign TRUE to terminated
            else prepare to read the first rule;
end supervisor(L).
```

6.6 Test

The top-level supervisor has been proven very useful. Of 120 problems in the test set, 104 are solved by the supervisor as shown in Table A.5 in Appendix A. The test was done in batch mode by putting all 120 problems together and with 50 seconds, 800 seconds, and 4 for the least time bound, the greatest time bound, and the time limit multiple respectively.

The top-level supervisor often reduces the total time required to find a proof. We found that some problems were solved with UR resolution strategy much faster than our best previous time using complete strategies. The inconvenience of trying an incomplete strategy inhibited us from using it on these problems, but the top-level supervisor removed that inconvenience. Therefore we feel that this idea is a significant enhancement of our system, eliminating the need for tedious bookkeeping by a human in trying a sequence of strategies.

Chapter 7

Comparison with Other Provers

In this chapter, we compare CLIN with some other major theorem provers. First, we show that duplication of instances of clauses is a real problem for theorem proving. Second, we want to show that instance-based proving method is good in two ways:

1. CLIN is better than the other provers for propositional and near-propositional problems.

2. Even for other problems, CLIN can prove more of them than the other provers. This shows that, especially for hard problems, the advantanges of an instance-based approach are starting to appear.

The other provers considered are OTTER [McC89], "sprfn" [Pla88][Nie89][NP90], and PTTP [Sti88]. All these provers were published in the past two years. We don't consider Prolog in the comparison for the following reasons. The inference rule, SLD resolution, that Prolog uses is only complete for Horn clauses; it is not complete for first-order clauses. Unification without occurs check and the implementation of "not" by negation as failure make Prolog unsound even for Horn clauses. The use of depth-first search makes Prolog incomplete; a path chosen may not terminate and maybe there is another path which succeeds. Because of these deficiencies, Prolog is not able to solve most of the problems in the test set.

In the following, we start with a brief introduction to OTTER, "sprfn", and PTTP. Then we use some propositional and near-propositional problems to show the significance of removing duplications. We then give statistics of some other problems to demonstrate the competitiveness of our system.

7.1 Introduction to Other Provers

7.1.1 OTTER

OTTER [McC89], developed in the Argonne National Laboratory, is a fast resolution-style theorem prover for first-order logic with equality. It is written in C and uses discrimination nets for fast look-up of potential unifications. OTTER provides many inference rules that can be chosen by the user, such as binary resolution, UR resolution, and hyperresolution. Hyperresolution applies to a set of clauses one of which must be a negative clause or a mixed clause, the remaining must be positive clauses and their number must be equal to the number of negative literals in the negative or mixed clause. The result of successful application must be a positive clause or the empty clause. Consider the following clauses:

1. $\{\neg\text{married}(X, Y), \neg\text{mother}(X, Z), \text{father}(Y, Z)\}$
2. $\{\text{married}(\text{thelma}, \text{pete}), \text{olderthan}(\text{thelma}, \text{pete})\}$
3. $\{\text{mother}(\text{thelma}, \text{steve})\}$

If we apply hyperresolution to these clauses, then we have

$$\{\text{father}(\text{pete}, \text{steve}), \text{olderthan}(\text{thelma}, \text{pete})\}$$

as a result.

7.1.2 sprfn

"sprfn" is based on the modified problem reduction format (MPRF) [Pla88]. MPRF uses the subgoaling paradigm. It basically derives $false$ from inference rules and axioms. Clauses are converted to inference rules in two steps. First, all clauses are converted into Horn-like clauses. That is, for non-Horn clauses, one of the positive literals is chosen as the head literal, and the other literals are negated and added to the body. Only one such representation is chosen for each non-Horn clause. Also, $false$ is added as the head literal for negative clauses. Second, all Horn-like clauses, $L : -L_1, \ldots, L_n$ in S, with possibly some of the L_i being negative, are converted to the following inference rule:

$$\frac{\Gamma_0 \rightarrow L_1 \Rightarrow \Gamma_1 \rightarrow L_1, \quad \Gamma_1 \rightarrow L_2 \Rightarrow \Gamma_2 \rightarrow L_2, \quad \ldots, \quad \Gamma_{n-1} \rightarrow L_n \Rightarrow \Gamma_n \rightarrow L_n}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_n \rightarrow L}$$

MPRF also has the following case analysis rule:

$$\frac{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1, \neg M \rightarrow L, \quad \Gamma_1, M \rightarrow L \Rightarrow \Gamma_1, M \rightarrow L}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1 \rightarrow L}$$

Γ and Γ_i are sets of literals. In a rule

$$\frac{A_1,\ A_2,\ \ldots,\ A_n}{\mathrm{A}}$$

if all A_i are derived, then we can derive A. MPRF has the following assumption axioms:

$$\begin{array}{ll} \Gamma \rightarrow L \Rightarrow \Gamma \rightarrow L, \text{ if } L \in \Gamma & L \text{ is a literal} \\ \Gamma \rightarrow \neg L \Rightarrow \Gamma, \neg L \rightarrow \neg L & L \text{ is a positive literal} \end{array}$$

MPRF is complete in the sense that S is unsatisfiable if and only if the sequent $\rightarrow false$ can be derived using the axioms and rules mentioned above.

Let's see how MPRF works for the clauses {¬p}, {p, ¬q}, {p, q}. First we convert the clauses into the following Horn-like clauses:

1. *false* :- q
2. p :- q
3. p :- ¬q

These clauses are then converted into the following inference rules:

$$\frac{\Gamma_0 \rightarrow \mathrm{p} \Rightarrow \Gamma_1 \rightarrow \mathrm{p}}{\Gamma_0 \rightarrow false \Rightarrow \Gamma_1 \rightarrow false}$$

$$\frac{\Gamma_0 \rightarrow \mathrm{q} \Rightarrow \Gamma_1 \rightarrow \mathrm{q}}{\Gamma_0 \rightarrow \mathrm{p} \Rightarrow \Gamma_1 \rightarrow \mathrm{p}}$$

$$\frac{\Gamma_0 \to \neg q \Rightarrow \Gamma_1 \to \neg q}{\Gamma_0 \to p \Rightarrow \Gamma_1 \to p}$$

Also we have the case analysis rule and assumption axioms given above. The proof obtained with MPRF is expressed as follows.

$$\frac{\dfrac{\dfrac{\to \neg q \Rightarrow \neg q \to \neg q}{\to p \Rightarrow \neg q \to p} \qquad \dfrac{q \to q \Rightarrow q \to q}{q \to p \Rightarrow q \to p}}{\to p \Rightarrow \to p}}{\to false \Rightarrow \to false}$$

"sprfn" has been implemented by Xumin Nie and David Plaisted [Pla88][Nie89][NP90]. It uses depth-first iterative deepening and caching of answers to avoid repeated solution of the same subgoal.

7.1.3 PTTP

PTTP is based on model elimination (ME) [Lov78][Sti88][MW88]. ME is a form of SL resolution that restricts one of the clauses at each resolution step to be the resolvent from the previous resolution step. The resolvent generated is called the *center* clause. The most recent resolvent is called the current center clause. The clause other than the center clause that enters the resolution step is called the *side clause.* Model elimination requires that the side clause at each step is either an input clause from S or a progenitor of the current center clause. Clause D is a progenitor of C if one of the following conditions holds:

1. The literal L of C resolved on was introduced previously in the resolvent of D and some side clause.
2. D is a progenitor of E and E is a progenitor of C.

Model elimination is complete for S if all contrapositives of the input clauses are added to S as the input set.

The information of all progenitors of a center clause can be represented in the clause itself with framed literals. When a literal is resolved on, the model elimination procedure does not eliminate it from the resolvent, but instead retains it as a framed literal. The list of framed literals to the right of a literal in a center clause is just the list of all the progenitors of the clause. Suppose we have a current center clause C. Leftmost framed literals in C can be removed immediately. If the literal to be resolved on in C is ever the complement of a framed literal, the literal can be just removed; the resulting clause is a resolvent of C and one progenitor of C.

Consider the following clauses: $\{\neg p\}$, $\{p, \neg q\}$, and $\{p, q\}$. Let's start with $\{\neg p\}$. By applying ME, we obtain the following proof:

- $\{\neg q, [\neg p]\}$ % resolve with $\{p, \neg q\}$
- $\{p, [\neg q], [\neg p]\}$ % resolve with $\{p, q\}$
- $\{[\neg q], [\neg p]\}$ % delete p
- $\{\}$ % delete leftmost framed literals

PTTP actually uses the MESON procedures [Lov78] which is a refinement of Model Elimination. It uses depth-first iterative-deepening search to make search complete. It also uses the same data structures and optimization techniques as for a Prolog implementation, to obtain very fast rates of inference, on the order of thousands or tens of thousands of inferences per second. The version we used for our comparison is a Prolog version that is not as fast as that reported in [Sti88], and does not have all of the optimizations of the faster version.

7.2 Propositional and Near-Propositional Problems

In this section, we want to show that CLIN performs better than other provers for propositional and near-propositional problems. Propositional problems contains no variables. All clauses contained in them are ground. Near-propositional problems do not contain many variables; the search space is usually finite. Since clauses of these problems are mainly propositional, the instance generation strategy has little effect on the performance of the provers. Therefore, the performance of the provers on these problems reveals the effect of duplication during the proof.

We use 12 problems for performance comparison. The times spent on these problems for different proving systems are listed in Table 7.1. All the times are obtained on a SUN3/60M workstation with 12 MB memory. The prolog system used for running PTTP, "sprfn", and CLIN is ALS-Prolog (Version 0.60) of Applied Logic System, Inc. [ALS88]. The

problem	no. of input clauses	OTTER (sec)	PTTP (sec)	sprfn (sec)	CLIN (sec)
ph4	22	63.620	—	88.833	0.600
ph5	45	38606.760	>128711	7145.450	1.800
ph6	81	—	—	—	7.300
ph7	133	—	—	—	39.600
ph8	204	—	—	—	276.467
ph9	297	—	—	—	2266.617
example	6	142.360	—	9.766	21.067
latinsq	16	—	—	—	85.567
liar	8	0.320	76.483	8.530	7.600
salt	44	1541.140	—	5728.383	28.016
jobs	45	—	—	—	639.883
zebra	128	—	—	—	1370.966

Table 7.1: Performance comparison on some propositional and near-propositional problems

first six problems, "ph4" through "ph9" are propositional problems, and the last six problems are near-propositional problems. The strategies used for OTTER are hyperresolution and UR resolution together, plus forward and backward subsumptions.

It is obvious that CLIN performs much better than the other provers for most of the cases. Since our system intends to eliminate duplication of instances of clauses, the result of the comparison indicates that such duplication might be a real problem that is hindering the other proving methods.

7.3 Other Problems

In the last section, we demonstrated that CLIN performs better for propositional and near-propositional problems. In this section, we want to show that CLIN is competitive with other provers on the other problems.

We list the times of some sample problems in Table 7.2. An entry with "—" indicates we stopped the run on the problem in the entry after the time spent is bigger, usually many times bigger, than those of the other entries for the same problem. All the times are obtained on a SUN3/60M workstation with 12 MB memory. The prolog system used for running PTTP, "sprfn", and CLIN is ALS-Prolog (Version 0.60) of Applied Logic System, Inc. [ALS88]. The statistics for OTTER are taken from [Nie89]. The strategies used are hyperresolution and UR resolution together, plus forward and backward subsumptions. A correct set of support clauses have to be supplied to OTTER for many problems, marked by "*" in the table, in order to get the proof. The statistics for "sprfn" are also taken from

no	problem	no. of input clauses	OTTER (sec)	PTTP (sec)	sprfn (sec)	CLIN (sec)
1	prim	9	0.740	15.100	2.180	9.150
2	haspartst1	8	0.380	1.000	1.300	6.333
3	haspartst2	8	0.400	278.620	3.800	18.700
4	ances2	7	0.140	0.020	2.930	0.133
5	num1	7	0.220	0.070	0.620	0.717
6	ew2	5	0.100	0.020	0.370	0.183
7	ew3	9	0.140	0.030	1.230	0.150
8	qw	3	0.280	10.000	0.750	1.383
9	mqw	5	0.260	1.830	0.630	4.950
10	dbabhp	14	198.700	1.250	6.150	12.200
11	apabhp	18	*13.540	—	—	208.166
12	ex4t1	7	1482.380	—	590.180	132.817
13	ex4t2	7	1610.440	4158.400	126.020	133.450
14	ex5	14	2932.540	—	+231.780	289.917
15	ex6t1	8	*1070.620	58968.620	23.180	43.750
16	ex6t2	8	*1067.860	184.400	23.580	43.917
17	wos4	25	*7818.760	5107.833	656.170	36.367
18	wos15	23	*36.980	—	6045.230	3509.150
19	wos19	33	*11.400	23.720	62.580	6328.533
20	wos20	33	—	—	—	#45003.100
21	wos22	34	—	—	—	73.383
22	wos26	24	—	—	—	30.766
23	wos27	30	*1.700	0.170	18.080	45.133
24	wos28	74	—	—	—	1859.683
25	wos29	40	*4.980	22.680	45.170	12.400
26	wos31	23	*103.190	—	+7742.220	#54992.967
27	wos33	26	185.360	—	+7.330	8.783
28	ls5	4	0.150	0.050	0.520	0.483
29	ls17	12	0.540	0.820	3.320	11.800
30	ls37	18	—	—	—	13311.517
31	ls65	20	*1.520	42.000	164.200	39.667
32	ls75	16	*13.240	19.130	25.480	10.317
33	ls87	22	*19.460	279.830	—	69.400
34	ls103	14	0.740	8.480	5.500	43.683
35	ls108	16	*293.920	—	1557.380	114.266
36	ls112	23	—	—	—	318.883
37	ls116	16	*3.540	37.350	10.630	24.583
38	chang_lee_2	7	0.520	2.016	5.470	3.883
39	chang_lee_8	9	0.700	15.100	2.300	9.183
40	chang_lee_9	8	0.480	0.680	2.820	5.183
41	schubert	30	3.920	2483.080	112.970	130.583

Table 7.2: Performance comparison on some other problems

[Nie89]. Contrapositives have to be added in the input clauses of some problems, marked by "+" in the table, in order to get the proof. Two problems for CLIN use user support; they are marked by "-".

The following observations can be made from Table 7.2:

1. A prover may do well for some problems, but may have trouble for other problems. For example, OTTER is very good for problems "apabhp", "wos15", "wos19", "wos31", and "schubert", but is bad for problems "ex4t1", "ex4t2", "ex5", "ex6t1", and "ex6t2".

2. CLIN is the only one that solves all the problems listed in the table. Of the 41 problems, OTTER can't solve 6 problems, PTTP can't solve 13 problems, and "sprfn" can't solve 8 problems.

3. There are 27 problems that can be solved by all the provers. The average times for these problems are: 437.497 seconds for OTTER, 2643.780 seconds for PTTP, 48.442 seconds for "sprfn", and 258.400 seconds for CLIN. Notice that "ex6t1" and "wos19" contribute mostly to the average times for PTTP and CLIN respectively. If we take the worst problem out and only consider the other 26 problems for each prover, then the average times are: 153.604 seconds for OTTER, 477.454 seconds for PTTP, 25.068 seconds for "sprfn", and 24.933 seconds for CLIN. One point we should mention here. When small proof checking is done slightly differently, our system can prove wos15 and wos19 very fast and would improve the average times significantly.

The input files of these 41 problems for CLIN are listed in Appendix B.

Chapter 8

Proving Hard Theorems

We have mentioned in Chapter 4 that predicate replacement allows the user to convey his or her knowledge about how to prove the underlying theorem to CLIN through replace rules. The replace rules direct the system to do guided search. In this chapter, we show the power of the replacement technique by proving some hard problems that are beyond the reach of most theorem provers due to the huge search space. These proofs relied on the use of replace rules to focus attention on objects that have already been constructed. Some guidelines of using the replace rules we followed are listed below:

- Prefer ground substitution. Ground substitution provides a strong restriction on electrons. Therefore, the number of instances generated is greatly reduced.

- The distinguished literals are picked to make ground substitution possible. The variables which appear at the right-hand side of a rule also appear at the left-hand side. Otherwise, context literals have to be used.

- Prefer backward-chaining approach. Backward-chaining focuses on the statement to be proved.

- We may switch backward-chaining to forward-chaining and vice versa at the point where ground substitution can't apply.

- Prefer rules whose left-hand side literals are not bigger than right-hand side literals. This prevents cycles. If we must use rules otherwise, we usually either treat them as once replace rules or add context literals to control them.

- A literal with many ground arguments is encouraged to be a distinguished literal.

- Treat a replace rule that may generate many instances as a once replace rule.

We should point out that the solutions given in this chapter are the results of many experiments. If an expected lemma was not generated, we examined clauses generated and found out what the problems were. Then we fixed the problems, changed option settings if necessary, and tried again. As we mentioned earlier, one use of replace rules is to convey the intuition or knowledge about how to prove the underlying theorem in order to guide the search efficiently. The merit of replace rules is fully appreciated in proving these hard theorems.

8.1 The Intermediate Value Theorem

8.1.1 Solution 1

Following are first-order axioms for the intermediate value theorem, which says that if a continuous function f is negative at a and positive at b, and a $\leq$ b, then there is a point X between a and b such that f(X) = 0. Here p(X, Y) means X $\leq$ Y.

- % two end points
  {p(a, b)}
  {p(f(a), 0)}
  {p(0, f(b))}

- % least upper bound axioms
  {p(X, l), ¬p(X, b), ¬p(f(X), 0)}
  {p(l, X), p(f(g(X)), 0)}
  {p(l, X), p(g(X), b)}
  {p(l, X), ¬p(g(X), X)}

- % inequality axioms
  {p(X, X)}
  {p(X, Z), ¬p(X, Y), ¬p(Y, Z)}
  {p(X, Y), p(Y, X)}

- % interpolation axioms
  {p(X, Y), ¬p(X, q(Y, X))}
  {p(X, Y), ¬p(q(Y, X), Y)}

- % continuity axioms
  {p(f(X), 0), ¬p(a, X), ¬p(X, b), ¬p(X, h(X))}
  {¬p(f(Z), 0), p(Z, h(X)), ¬p(a, X), ¬p(X, b), ¬p(Z, X), p(f(X), 0)}

$\{p(0, f(X)), \neg p(a, X), \neg p(X, b), \neg p(k(X), X)\}$
$\{\neg p(0, f(Z)), p(k(X), Z), \neg p(a, X), \neg p(X, b), \neg p(X, Z), p(0, f(X))\}$

- % negation of the theorem
  $\{\neg p(f(X), 0), \neg p(0, f(X))\}$

The least upper bound axioms assert that l is the least upper bound of the set of X such that $f(X) \leq 0$ and $X \leq b$. Call this set A. The interpolation axioms state that there is a real number between any two real numbers. The continuity axioms state that if $f(X) < 0$ then there is an interval following X such that $f(Z) < 0$ for all Z in this interval, and if $f(X) > 0$ then there is an interval preceding X such that $f(Z) > 0$ for all Z in this interval. The proof goes like this: Suppose $f(l) < 0$. Then there is some interval following l such that $f(Z) < 0$ for Z in this interval. This interval cannot include b since $f(b) \geq 0$. Then there is some Z in this interval, $Z \leq b$, such that $f(Z) < 0$, by the interpolation axioms. This contradicts the fact that l is the least upper bound of A. Suppose $f(l) > 0$. Then there is some interval preceding l such that $f(l) > 0$. Then any Z in this interval is an upper bound of A, hence l is not the least upper bound of A. This completes the proof.

One would think that such a simple proof would be easy for a computer. However, most provers cannot obtain this proof. One of the few that can is the prover of Bledsoe and Hines [BH80] at Austin, which uses special inference rules for dense linearly ordered sets. Note that most of the inference steps involve reasoning about terms that have already been constructed, and not many involve constructing new terms, except in certain reasonable ways. We can formalize this by converting the clauses to replace rules, and then the above proof can be entirely obtained by the application of replace rules and ground substitution. Since the goal is not ground, the replace rules are set up for forward-chaining derivation in the following:

1. $p(X, b), p(f(X), 0) \rightarrow p(X, l)$
2. $\neg p(l, X) \rightarrow p(f(g(X)), 0)$
3. $\neg p(l, X) \rightarrow p(g(X), b)$
4. $\neg p(l, X) \rightarrow \neg p(g(X), X)$
5. $\neg p(f(g(X)), 0) \rightarrow p(l, X)$
6. $\neg p(g(X), b) \rightarrow p(l, X)$
7. $p(g(X), X) \rightarrow p(l, X)$
8. $\neg p(Y, X) \rightarrow p(X, Y)$

9. $p(a, X) \rightarrow \neg p(X, b), p(f(X), 0), \neg p(X, h(X))$

10. $p(a, X), p(X, b), \neg p(f(X), 0), p(Z, X), \neg p(Z, h(X)) \rightarrow \neg p(f(Z), 0)$

11. $p(a, X) \rightarrow \neg p(X, b), p(0, f(X)), \neg p(k(X), X)$

12. $p(a, X), p(X, b), \neg p(0, f(X)), p(X, Z), \neg p(k(X), Z) \rightarrow \neg p(0, f(Z))$

13. $p(a, X), p(X, b), p(X, Z), \neg p(0, f(X)) \rightarrow p(k(X), Z), \neg p(0, f(Z))$

14. $p(f(X), 0) \rightarrow \neg p(0, f(X))$

15. $p(0, f(X)) \rightarrow \neg p(f(X), 0)$

16. $p(X, Y), p(Y, Z) \rightarrow p(X, Z)$

17. $\neg p(X, Y) \rightarrow \neg p(X, q(Y, X))$

18. $\neg p(X, Y) \rightarrow \neg p(q(Y, X), Y)$

The last 3 rules are once replace rules. The transitivity rule may generate many instances, and the interpolation rules generate new terms. They are controlled by treating them as once replace rules. Replace rules 5, 6, and 7 are inverted version of rules 2, 3, and 4. We do this because the instance of a distinguished literal is not allowed to be an electron. The replace rules above are very natural. Rule 1 states that if $X \leq b$ and $f(X) \leq 0$, then $X \leq l$. This is the definition of least upper bound l. The least upper bound l is defined in terms of whether p(X, l) and p(l, X) are true for arbitrary X. Hence it is natural to make the literal containing p(X, l) or p(l, X) the distinguished literal, as in rules 2, 3, and 4. Rules 9 and 11 are used for generating p(l, b) once p(a, l) has been derived, and for instantiating the variable in the goal to l. These are necessary since we use ground substitution. Rules 10 and 12 are used for finding the polarity of a point in a known interval. Rule 13 is used for finding the other end point of a desired interval. Rules 17 and 18 are for interpolation; they generate a particular point in an interval. The rest of the rules are trivial.

Notice that clauses {p(a, b)}, {p(f(a), 0)}, and {p(0, f(b))} are axioms. The proof using these replace rules and axioms is described as follows. Applying rule 1 on p(a, b) and p(f(a), 0), we get

- C_1: {p(a, l)}

Applying rule 9 on p(a, l), we get

- C_2: $\{\neg p(l, b), p(f(l), 0), \neg p(l, h(l))\}$

Applying rules 2, 3, and 4 on ¬p(l, h(l)) respectively, we have

- C_3: {p(l, h(l)), p(f(g(h(l))), 0)}
- C_4: {p(l, h(l)), p(g(h(l)), b)}
- C_5: {p(l, h(l)), ¬p(g(h(l)), h(l))}

Applying rule 3 on ¬p(l, b) then rule 7 on p(g(b), b), we have

- C_6: {p(l, b), p(g(b), b)}
- C_7: {p(g(b), b), p(l, b)}

Applying rule 1 on p(g(h(l)), b) and p(f(g(h(l))), 0), we have

- C_8: {¬p(g(h(l)), b), ¬p(f(g(h(l))), 0), p(g(h(l)), l)}

Applying rule 11 on p(a, l), we have

- C_9: {¬p(l, b), p(0, f(l)), ¬p(k(l), l)}

Applying rule 15 on p(0, f(l)), we get

- C_{10}: {¬p(0, f(l)), ¬p(f(l), 0)}

Applying rule 10 on p(a, l), p(l, b), ¬p(f(l), 0), p(g(h(l)), l), and ¬p(g(h(l)), h(l)), we have

- C_{11}: {¬p(l, b), p(f(l), 0), ¬p(g(h(l)), l), p(g(h(l)), h(l)), ¬p(f(g(h(l))), 0)}

Applying rule 14 on p(f(l), 0), we have

- C_{12}: {¬p(f(l), 0), ¬p(0, f(l))}

Applying rule 13 on p(a, l), p(l,b), ¬p(0, f(l)), and then unit simplified by {p(0, f(b))}, we have

- C_{13}: {¬p(l,b), p(0, f(l)), p(k(l), b)}

Applying rules 17 and 18 on ¬p(k(l), l) respectively, we get

- C_{14}: {p(k(l), l), ¬p(k(l), q(l, k(l)))}
- C_{15}: {p(k(l), l), ¬p(q(l, k(l)), l)}

Note that rules 17 and 18 are once replace rules. Applying rule 8 on ¬p(k(l), q(l, k(l))) and ¬p(q(l, k(l)), l) respectively, we have

- C_{16}: {p(k(l), q(l, k(l))), p(q(l, k(l)), k(l))}
- C_{17}: {p(q(l, k(l)), l), p(l, q(l, k(l)))}

Applying rule 12 on p(a, l), p(l, b), ¬p(0, f(l)), p(l, q(l, k(l))), and ¬p(k(l), q(l, k(l))), we have

- C_{18}: {¬p(l, b), p(0, f(l)), ¬p(l, q(l, k(l))), p(k(l), q(l, k(l))), ¬p(0, f(q(l, k(l))))}

Applying rule 8 on ¬p(0, f(q(l, k(l)))), we have

- C_{19}: {p(0, f(q(l, k(l)))), p(f(q(l, k(l))), 0)}

This ends the first round of replacement since no new instances can be generated from repeated replace rules. Now we start the second round. Applying rule 16 on p(q(l, k(l)), k(l)) and p(k(l), b), we have

- C_{20}: {¬p(q(l, k(l)), k(l)), ¬p(k(l), b), p(q(l, k(l)), b)}

Note that rule 16 is a once replace rule. Applying rule 1 on p(q(l, k(l)), b) and p(f(q(l, k(l))), 0), we have

- C_{21}: {¬p(q(l, k(l)), b), ¬p(f(q(l, k(l))), 0), p(q(l, k(l)), l)}

The instances generated so far are contradictory. In fact, 19 clauses constitute an unsatisfiable subset. They are C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9, C_{10}, C_{11}, C_{13}, C_{14}, C_{15}, C_{16}, C_{17}, C_{18}, C_{19}, C_{20}, and C_{21}. The justification is easy. From C_6 and C_7, we know p(l, b). Then we do case analysis on C_{10}. Suppose ¬p(f(l), 0) is true. By applying unit literal deletion iteratively, we obtain the following unit clauses from C_2, C_3, C_4, C_5, C_8, and C_{11} respectively: {¬p(l, h(l))}, {p(f(g(h(l))), 0)}, {p(g(h(l)), b)}, {¬p(g(h(l)), h(l))}, {p(g(h(l)), l)}, {¬p(f(g(h(l))), 0)}. The second and sixth unit clauses are contradictory. On the other hand, suppose ¬p(0, f(l)) is true. By applying unit literal deletion iteratively, we obtain the following unit clauses from C_9, C_{13}, C_{14}, C_{15}, C_{16}, C_{17}, C_{18}, C_{19}, C_{20}, and C_{21} respectively: {¬p(k(l), l)}, {p(k(l), b)}, {¬p(k(l), q(l, k(l)))}, {¬p(q(l, k(l)), l)}, {p(q(l, k(l)), k(l))}, {p(l, q(l, k(l)))}, {¬p(0, f(q(l, k(l))))}, {p(f(q(l, k(l))), 0)}, {p(q(l, k(l)), b)}, {p(q(l, k(l)), l)}. The fourth and tenth unit clauses are contradictory. CLIN spends 1467.216 seconds on this theorem. The proof is detected by the propositional calculus prover.

8.1.2 Solution 2

The first solution needs 2 rounds of replacement. The reason is that we have interpolation rules and the transitivity rule as once replace rules. If we can change either interpolation rules or the transitivity rule to repeated replace rules, then the proof can be obtained in the first round. We chose to change the interpolation rules. This was achieved by adding context literals to reduce the number of instances generated from these rules.

It is not hard to figure out what context literals are needed. It is reasonable to expect that a desired interval is generated from the continuity axiom. This leads the following two rules:

- p(a, X), p(X, b), ¬p(f(X), 0) → ¬p(Z, X), p(Z, h(X)), ¬p(f(Z), 0)
- p(a, X), p(X, b), ¬p(0, f(X)) → ¬p(X, Z), p(k(X), Z), ¬p(0, f(Z))

These two rules are added to repeated replace rules. Let's consider the first rule. The interval generated is $h(X) < Z \leq X$. This interval corresponds to the negation of $(\neg(Z \leq X) \vee Z \leq h(X))$. Therefore, the context literals are $\neg(Z \leq X)$ and $Z \leq Y$ for $Y < X$. For the second rule, the interval generated is $X \leq Z < k(X)$. Therefore, the context literals are $\neg(Y \leq Z)$ and $X \leq Z$. We may replace Z by a new constant c since Z does not appear at the right-hand side of the rules. Therefore we use the following 4 rules for the interpolation property:

- ¬p(X, Y), p(c, Y), ¬p(c, X), → ¬p(X, q(Y, X))
- ¬p(X, Y), p(c, Y), ¬p(c, X), → ¬p(q(Y, X), Y)
- ¬p(X, Y), ¬p(Y, c), p(X, c) → ¬p(X, q(Y, X))
- ¬p(X, Y), ¬p(Y, c), p(X, c) → ¬p(q(Y, X), Y)

With these changes to the replace rules, the theorem is proved in 137.783 seconds.

8.2 The Theorem of Greatest Common Divisor

The theorem of greatest common divisor (gcd) says that if Z is gcd(X, Y) then Z×W is gcd(X×W, Y×W). Following are axioms for this theorem:

- {g(U, V, W), ¬d(W, U), ¬d(W, V), ¬d(h(U, V, W), W)}
- {d(U, f(W, V)), ¬d(U, f(V, W))}
- {d(U, f(V, W)), ¬d(V, U), ¬d(q(U, V), W)}
- {d(U, W), ¬d(U, V), ¬d(V, W)}
- {d(W, U), ¬g(V, X, U), ¬d(W, V), ¬d(W, X)}
- {d(V, f(W, U)), ¬d(V, W)}
- {d(q(W, V), U), ¬d(W, f(V, U)), ¬d(V, W)}
- {d(V, k(W, U)), ¬d(V, U), ¬d(V, W)}
- {d(h(U, V, W), U), ¬d(W, U), ¬d(W, V), g(U, V, W)}
- {d(h(U, V, W), V), ¬d(W, U), ¬d(W, V), g(U, V, W)}
- {d(f(U, V), f(U, W)), ¬d(V, W)}
- {d(f(V, U), f(W, U)), ¬d(V, W)}
- {d(U, q(W, V)), ¬d(f(U, V), W), ¬d(V, W)}
- {d(W, U), ¬g(U, V, W)}
- {d(W, V), ¬g(U, V, W)}
- {¬g(f(a, c), f(b, c), f(e, c))}
- {d(V, f(U, V))}
- {d(U, f(U, V))}
- {d(k(U, V), V)}
- {d(k(V, U), V)}
- {g(a, b, e)}

- $\{d(q(U, U), V)\}$
- $\{d(U, U)\}$

In the axioms, g(X, Y, Z) means gcd(X, Y) is Z, d(X, Y) means X divides Y, f(X, Y) means $X \times Y$, k(X, Y) represents gcd(X, Y), q(X,Y) means $X \div Y$. The goal statement are axioms $\{\neg g(f(a, c), f(b, c), f(e, c))\}$ and $\{g(a, b, e)\}$. Since the goal is ground, we set up the replace rules for backward-chaining, as shown in the following:

1. $\neg g(U, V, W) \rightarrow \neg d(W, U), \neg d(W, V), \neg d(h(U, V, W), W)$
2. $\neg d(U, f(W, V)) \rightarrow \neg d(U, f(V, W))$
3. $\neg d(U, f(V, W)) \rightarrow \neg d(V, U), \neg d(q(U, V), W)$
4. $\neg d(U, W), d(U, V) \rightarrow \neg d(V, W)$
5. $\neg d(W, U), g(V, X, U) \rightarrow \neg d(W, V), \neg d(W, X)$
6. $\neg d(V, f(W, U)) \rightarrow \neg d(V, W)$
7. $\neg d(q(W, V), U) \rightarrow \neg d(W, f(V, U)), \neg d(V, W)$
8. $\neg d(V, k(W, U)) \rightarrow \neg d(V, U), \neg d(V, W)$
9. $\neg g(U, V, W) \rightarrow d(h(U, V, W), U), \neg d(W, U), \neg d(W, V)$
10. $\neg g(U, V, W) \rightarrow d(h(U, V, W), V), \neg d(W, U), \neg d(W, V)$
11. $\neg d(f(U, V), f(U, W)) \rightarrow \neg d(V, W)$
12. $\neg d(f(V, U), f(W, U)) \rightarrow \neg d(V, W)$
13. $\neg d(U, q(W, V)) \rightarrow \neg d(f(U, V), W), \neg d(V, W)$
14. $g(U, V, W) \rightarrow d(W, U)$
15. $g(U, V, W) \rightarrow d(W, V)$
16. $d(V, U), d(V, W) \rightarrow d(V, k(W, U))$

The last one is the only once replace rule. These replace rules are very natural for backward-chaining. The only rule that is not backward-chaining is the last rule, which is used for forward-chaining due to the restriction of ground substitution.

We describe the proof below. Notice that clauses $\{\neg g(f(a, c), f(b, c), f(e, c))\}$ and $\{g(a, b, e)\}$ are axioms. Applying rule 1 on $\neg g(f(a, c), f(b, c), f(e, c))$, we have

- C_1: {¬d(f(e, c), f(a, c)), ¬d(f(e, c), f(b, c)), ¬d(h(f(a, c), f(b, c), f(e, c)), f(e, c))}

Applying rule 12 on ¬d(f(e, c), f(a, c)) and ¬d(f(e, c), f(b, c)) respectively, we have

- C_2: {d(f(e, c), f(a, c)), ¬d(e, a)}
- C_3: {d(f(e, c), f(b, c)), ¬d(e, b)}

Applying rules 14 and 15 on g(a, b, e) respectively, we have

- C_4: {d(e, a)}
- C_5: {d(e, b)}

Applying rule 2 on ¬d(h(f(a, c), f(b, c), f(e, c)), f(e, c)), we have

- C_6: {d(h(f(a, c), f(b, c), f(e, c)), f(e, c)), ¬d(h(f(a, c), f(b, c), f(e, c)), f(c, e))}

Applying rules 9 and 10 on ¬g(f(a, c), f(b, c), f(e, c)) respectively, we have

- C_7: {d(h(f(a, c), f(b, c), f(e, c)), f(a, c)), ¬d(f(e, c), f(a, c)), ¬d(f(e, c), f(b, c))}
- C_8: {d(h(f(a, c), f(b, c), f(e, c)), f(b, c)), ¬d(f(e, c), f(a, c)), ¬d(f(e, c), f(b, c))}

Applying rule 16 on d(h(f(a, c), f(b, c), f(e, c)), f(b, c)) and d(h(f(a, c), f(b, c), f(e, c)), f(a, c)), we have

- C_9: {¬d(h(f(a, c), f(b, c), f(e, c)), f(b, c)), ¬d(h(f(a, c), f(b, c), f(e, c)), f(a, c)), d(h(f(a, c), f(b, c), f(e, c)), k(f(a, c), f(b, c)))}

Note that rule 16 is a once replace rule. Applying rule 4 on ¬d(h(f(a, c), f(b, c), f(e, c)), f(c, e)) and d(h(f(a, c), f(b, c), f(e, c)), k(f(a, c), f(b, c))), we have

- C_{10}: {d(h(f(a, c), f(b, c), f(e, c)), f(c, e)), ¬d(h(f(a, c), f(b, c), f(e, c)), k(f(a, c), f(b, c))), ¬d(k(f(a, c), f(b, c)), f(c, e))}

Applying rule 3 on ¬d(k(f(a, c), f(b, c)), f(c, e)), we have

- C_{11}: {d(k(f(a, c), f(b, c)), f(c, e)), ¬d(c, k(f(a, c), f(b, c))), ¬d(q(k(f(a, c), f(b, c)), c), e)}

Applying rule 8 on ¬d(c, k(f(a, c), f(b, c))) and then unit simplified by {d(V, f(U, V))}, we have

- C_{12}: {d(c, k(f(a, c), f(b, c)))}

Applying rule 5 on ¬d(q(k(f(a, c), f(b, c)), c), e) and g(a, b, e), we have

- C_{13}: {d(q(k(f(a, c), f(b, c)), c), e), ¬d(q(k(f(a, c), f(b, c)), c), a), ¬d(q(k(f(a, c), f(b, c)), c), b)}

Applying rule 7 on ¬d(q(k(f(a, c), f(b, c)), c), a) and then unit simplified by C_{12}, we have

- C_{14}: {d(q(k(f(a, c), f(b, c)), c), a), ¬d(k(f(a, c), f(b, c)), f(c, a))}

Applying rule 7 on ¬d(q(k(f(a, c), f(b, c)), c), b) and then unit simplified by C_{12}, we have

- C_{15}: {d(q(k(f(a, c), f(b, c)), c), b), ¬d(k(f(a, c), f(b, c)), f(c, b))}

Applying rule 2 on ¬d(k(f(a, c), f(b, c)), f(c, a)) and then unit simplified by {d(k(V, U), V)}, we have

- C_{16}: {d(k(f(a, c), f(b, c)), f(c, a))}

Applying rule 2 on ¬d(k(f(a, c), f(b, c)), f(c, b)) and then unit simplified by {d(k(U, V), V)}, we have

- C_{17}: {d(k(f(a, c), f(b, c)), f(c, b))}

Clauses C_1 through C_{17} are unsatisfiable. The unsatisfiability is obtained by unit simplification. Starting from unit clauses C_4, C_5, C_{16}, and C_{17}, after several rounds of unit literal deletions, C_{11} is simplified to {¬d(c, k(f(a, c), f(b, c)))}, which is complementary to C_{12}. CLIN spends 74.516 seconds on this theorem.

8.3 LCM

The theorem of least common multiple (lcm) says that if Z is gcd(X, Y), then the least common multiple lcm(X, Y) of X and Y is X times Y divided by Z. Following are the axioms for this problem:

- $\{\neg \mathrm{lm}(a, b, q(f(a, b), c))\}$
- $\{\mathrm{lm}(X, Y, U), \neg d(X, U), d(Y, U), \neg d(U, k(Y, X, U))\}$
- $\{d(X, q(Y, Z)), \neg d(f(X, Z), Y)\}$
- $\{d(q(X, Y), Z), \neg d(Y, X), \neg d(X, f(Y, Z))\}$
- $\{g(f(X, Z), f(Y, Z), f(U, Z)), \neg g(X, Y, U)\}$
- $\{d(V, U), \neg g(X, Y, U), \neg d(V, X), \neg d(V, Y)\}$
- $\{d(U, f(X, Y)), \neg d(U, f(Y, X))\}$
- $\{d(f(X, Y), f(X, Z)), \neg d(Y, Z)\}$
- $\{d(f(Y, X), f(Z, X)), \neg d(Y, Z)\}$
- $\{d(X, f(Y, Z)), \neg d(X, Y)\}$
- $\{d(U, Y), \neg g(X, Y, U)\}$
- $\{d(U, X), \neg g(X, Y, U)\}$
- $\{g(a, b, c)\}$
- $\{d(X, k(Y, X, U)), \neg d(X, U), \neg d(Y, U), \mathrm{lm}(X, Y, U)\}$
- $\{d(Y, k(Y, X, U)), \neg d(X, U), \neg d(Y, U), \mathrm{lm}(X, Y, U)\}$
- $\{d(X, X)\}$
- $\{d(X, f(X, Y))\}$
- $\{d(q(X, X), Y)\}$
- $\{d(X, f(Y, Z)), \neg d(q(X, Y), Z), \neg d(Y, X)\}$
- $\{d(h(Y, X, U), X), g(X, Y, U), \neg d(U, X), \neg d(U, Y)\}$
- $\{d(h(Y, X, U), Y), g(X, Y, U), \neg d(U, X), \neg d(U, Y)\}$
- $\{g(X, Y, U), \neg d(h(Y, X, U), U), \neg d(U, X), \neg d(U, Y)\}$

- $\{d(X, Z), \neg d(X, Y), \neg d(Y, Z)\}$

The notations used here are the same as those in gcd except lm(X, Y, Z) means Z is lcm(X, Y).

We use the following axioms

1. $\neg\{lm(a, b, q(f(a, b), c))\}$
2. $\{g(a, b, c)\}$
3. $\{d(X, X)\}$
4. $\{d(X, f(X, Y))\}$
5. $\{d(q(X, X), Y)\}$

and the following replace rules

1. $g(X, Y, U), \neg g(V, W, f(U, Z)) \rightarrow g(f(X, Z), f(Y, Z), f(U, Z))$
2. $\neg lm(X, Y, U) \rightarrow \neg d(X, U), \neg d(Y, U), \neg d(U, k(Y, X, U))$
3. $\neg d(X, q(Y, Z)) \rightarrow \neg d(f(X, Z), Y)$
4. $\neg d(q(X, Y), Z) \rightarrow \neg d(Y, X), \neg d(X, f(Y, Z))$
5. $\neg d(V, U) \rightarrow \neg g(X, Y, U), \neg d(V, X), \neg d(V, Y)$
6. $\neg d(V, U), g(X, Y, U) \rightarrow \neg d(V, X), \neg d(V, Y)$
7. $\neg d(U, f(X, Y)) \rightarrow \neg d(U, f(Y, X))$
8. $d(U, f(Y, X)) \rightarrow d(U, f(X, Y))$
9. $\neg d(f(X, Y), f(X, Z)) \rightarrow \neg d(Y, Z)$
10. $\neg d(f(Y, X), f(Z, X)) \rightarrow \neg d(Y, Z)$
11. $\neg d(X, f(Y, Z)) \rightarrow \neg d(X, Y)$
12. $g(X, Y, U) \rightarrow d(U, Y)$
13. $g(X, Y, U) \rightarrow d(U, X)$
14. $\neg lm(X, Y, U) \rightarrow \neg d(X, U), \neg d(Y, U), d(X, k(Y, X, U))$

15. $\neg lm(X, Y, U) \rightarrow \neg d(X, U), \neg d(Y, U), d(Y, k(Y, X, U))$

16. $\neg g(X, Y, U) \rightarrow \neg d(h(Y, X, U), U), \neg d(U, X), \neg d(U, Y)$

17. $d(q(X, Y), Z) \rightarrow \neg d(Y, X), d(X, f(Y, Z))$

18. $\neg g(X, Y, U) \rightarrow d(h(Y, X, U), Y), \neg d(U, X), \neg d(U, Y)$

19. $\neg g(X, Y, U) \rightarrow d(h(Y, X, U), X), \neg d(U, X), \neg d(U, Y)$

20. $\neg d(X, Z), d(X, Y) \rightarrow \neg d(Y, Z)$

All are repeated replace rules. These rules are similar to gcd and are for backward-chaining. The second distinguished literal of rule 1 is a context literal. Rule 1 is used for generating a big literal, so it needs context literal to control it. Also, since we use ground substitution, the context literal helps to instantiate the variable Z at the right-hand side. This rule says, if we know g(X, Y, U) and are looking for g(V, W, f(U, Z)), then consider gcd(f(X, Z), f(Y, Z), f(U, Z)). This is a very natural rule. The proof is obtained totally by replacement and ground substitution in 13.700 seconds.

8.4 AM8

This theorem states that a continuous function f in a closed real interval [a, b] attains its minimum (or maximum) in this interval [WB87]. Following are the axioms for this problem:

- % facts
  $\{p(a, l)\}$
  $\{p(l, b)\}$
- % inequality axioms
  $\{p(X, X)\}$
  $\{p(X, Y), p(Y, X)\}$
  $\{p(X, Z), \neg p(X, Y), \neg p(Y, Z)\}$
  $\{p(f(X), f(Y)), \neg p(X, Y), \neg p(Y, X)\}$
- $\{p(f(l), f(X)), \neg p(a, X), \neg p(X, l)\}$
- % continuity axiom
  $\{p(a, h(X)), \neg p(a, X), \neg p(X, b)\}$
  $\{p(h(X), b), \neg p(a, X), \neg p(X, b)\}$
  $\{p(f(h(X)), f(X)), \neg p(a, X), \neg p(X, b)\}$

$\{p(h(X), Y), \neg p(a, X), \neg p(X, b), \neg p(a, Y), \neg p(Y, b), \neg p(f(Y), f(X))\}$
$\{p(h(X), Y), \neg p(a, X), \neg p(X, b), \neg p(a, Y), \neg p(Y, b), \neg p(f(Y), f(X))\}$

- % least upper bound axiom
  $\{p(X, l), p(a, q(X)), \neg p(a, X), \neg p(X, b)\}$
  $\{p(X, l), \neg p(a, X), \neg p(X, b), \neg p(f(X), f(q(X)))\}$
  $\{p(X, l), p(q(X), X), \neg p(a, X), \neg p(X, b)\}$

- % negation of the theorem
  $\{p(a, k(X)), \neg p(a, X), \neg p(X, b)\}$
  $\{p(k(X), b), \neg p(a, X), \neg p(X, b)\}$
  $\{\neg p(a, X), \neg p(X, b), \neg p(f(X), f(k(X)))\}$

p(X, Y) still means $X \leq Y$.

We use the following axioms

1. $\{p(a, l)\}$
2. $\{p(l, b)\}$
3. $\{p(X, X)\}$

Since the goal statement is not ground, we use forward-chaining replace rules mostly as in the following:

1. $\neg p(Y, X) \rightarrow p(X, Y)$
2. $p(Y, Z), \neg p(X, Z) \rightarrow \neg p(X, Y)$
3. $p(X, Y), \neg p(X, Z) \rightarrow \neg p(Y, Z)$
4. $p(X, Y), \neg p(f(X), f(Y)) \rightarrow \neg p(Y, X)$
5. $p(a, X), \neg p(f(l), f(X)) \rightarrow \neg p(X, l)$
6. $p(a, X), p(X, b) \rightarrow p(h(X), b)$
7. $p(a, X), p(X, b) \rightarrow p(f(h(X)), f(X))$
8. $p(a, X), p(X, b), p(a, Y), p(Y, b), p(f(Y), f(X)) \rightarrow p(h(X), Y)$
9. $p(a, X), p(X, b), \neg p(X, l) \rightarrow p(a, q(X))$
10. $p(a, X), p(X, b), \neg p(X, l) \rightarrow \neg p(f(X), f(q(X)))$

11. $p(a, X), p(X, b), \neg p(X, l) \rightarrow p(q(X), X)$
12. $p(a, X), p(X, b) \rightarrow p(k(X), b)$
13. $p(a, X), p(X, b) \rightarrow \neg p(f(X), f(k(X)))$
14. $p(X, Y), p(Y, Z) \rightarrow p(X, Z)$
15. $p(a, X), p(X, b) \rightarrow p(a, h(X))$
16. $p(a, X), p(X, b) \rightarrow p(a, k(X))$

The last three are once replace rules. The proof is obtained in 148.066 seconds, at the third round of replacement, with ground substitution.

Chapter 9

Applications

As described in the previous chapters, CLIN generates instances and uses the propositional calculus prover as one of the facilities to test if the instances are unsatisfiable. With the capability of model-finding of the propositional calculus prover, CLIN is able to find solutions for a given problem. This application enables CLIN to act as a problem solver, not just a theorem prover.

This is an important application. In practice, a problem is described as a set of rules or constraints. Such problems are often called constraint satisfaction problems (CSPs) in the area of Artificial Intelligence. We don't know what conclusions can be drawn from these constraints. Namely, the solutions of the problem are unknown. Therefore, it is hard to get help from a normal theorem prover in this case. CLIN provides a means of solving constraint satisfaction problems declaratively.

This problem solving capability also offers applications to other related fields. CLIN can be used as a back-end processor of a deductive database system. All possible conclusions of a deductive database are drawn for retrieval of a front-end query processor. CLIN might also help debug axioms. One of the difficulties in theorem proving concerns the adequacy or correctness of axioms for a given problem. We provide some help for this difficulty.

Special proving systems have been presented for proving temporal logic theorems. However, the techniques used in these systems are not human-oriented. The facility of predicate replacement in CLIN provides a new paradigm for proving temporal logic theorems. Replace rules are used to replace logical and modal operators by their definitions. The proofs obtained by our system are very similar to those obtained by human beings.

9.1 Problem Solving

Given a problem in terms of constraints or rules, an ideal problem solver is expected to find solutions for this problem by satisfying the following criteria:

1. The problem representation the solver accepts is fully declarative.
2. The solutions obtained are sound.
3. The problem solver is complete so that all possible solutions can be obtained.
4. The use of negative information is allowed.

Satisfying these criteria is not a trivial task. In fact, people have been trying hard to find a qualified problem solver. Prolog is one of the most famous solvers ever presented. However, it fails to satisfy any of the four criteria given above.

We claim that CLIN is a problem solver that meets all the criteria given above. The problem representation it accepts is in first order logic which is fully declarative. The solutions CLIN obtains are logical consequence of the constraints, so they are sound. CLIN is a complete system. Finally, negative information are processed correctly.

9.1.1 CLIN as a Problem Solver

The idea is basically as follows. Suppose we have a set A of axioms which are derived from the rules or constraints of a given problem. CLIN generates the set T of all instances of A, then calls the propositional calculus prover on Gr(T). Since A is satisfiable, the propositional calculus prover will find a model of Gr(T), that is, a truth assignment that makes all clauses in Gr(T) true. This model will give the solution to the problem. Once a model is found, a clause can be added which negates this truth assignment, and CLIN can be run again. In this way, more than one solution to the problem can be found. When all such solutions have been found, CLIN will detect that the clauses are unsatisfiable, indicating that there are no more solutions. This procedure is shown in Figure 9.1.

This technique works well if the number of instances generated by hyper-linking is finite. In practical applications, a problem usually involves finite domains. Such a problem can be represented without function symbols, and is guaranteed to have a finite number of instances. In this case, CLIN works as a decision procedure; it always stops and outputs a solution if one exists.

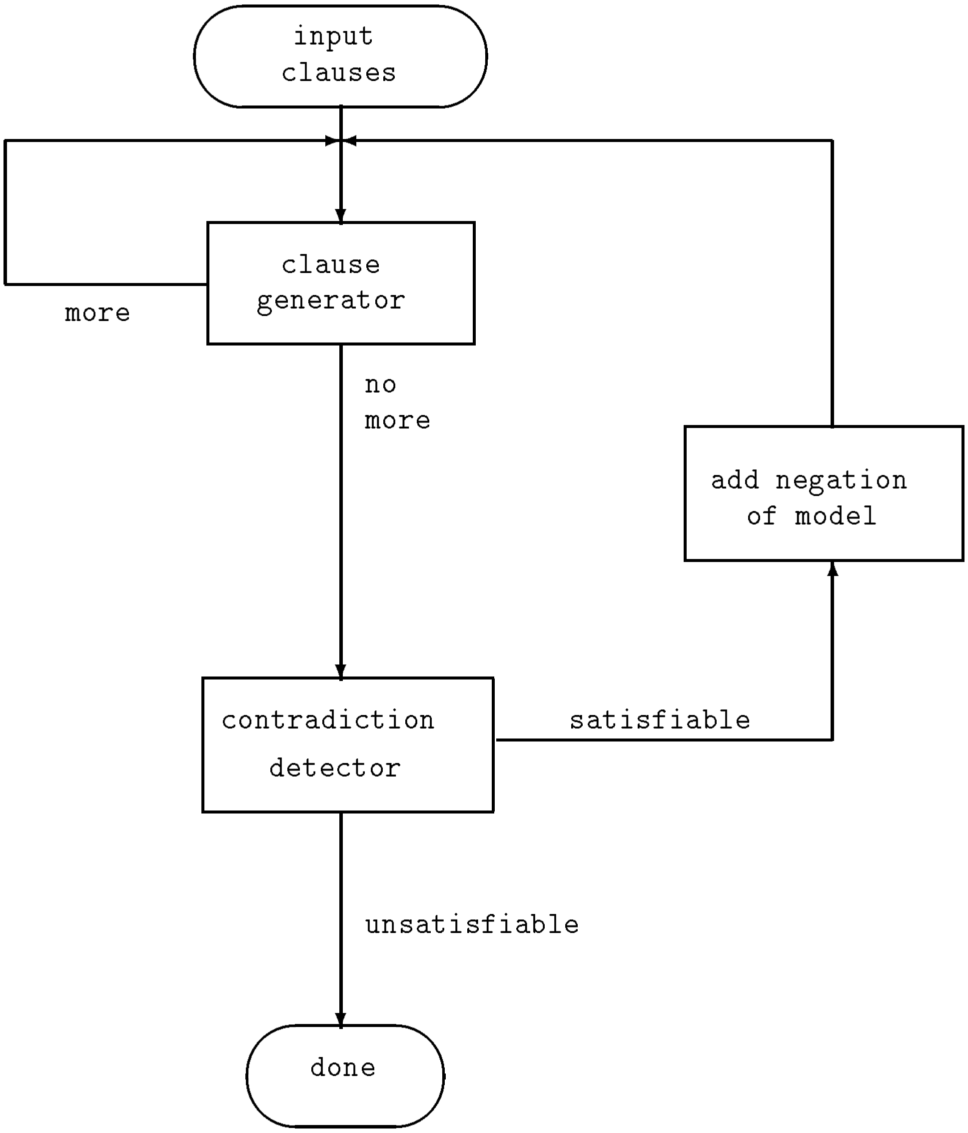

Figure 9.1: CLIN as a problem solver

9.1.2 Transformation rules

We describe some transformation rules for converting a first-order formula with finite domains to function-free first-order formulas. This is essentially the same as Schonfinkel-Bernays form in logic [Lew80], which is known to be decidable. The function-free first-order form guarantees finitely many instances generated by hyper-linking. Thus CLIN is guaranteed to stop with an answer.

Suppose the domain of the problem contains elements a_1, ..., and a_n.

1. Suppose there is an expression:

$$\exists \mathrm{X}\ \mathrm{p}[\mathrm{X}]$$

The expression can be transformed into the following function-free expression:

$$\mathrm{p}[a_1] \vee \mathrm{p}[a_2] \vee \cdots \vee \mathrm{p}[a_n]$$

2. If the formula is of the form

$$\exists \mathrm{X}\ (\mathrm{q}(\mathrm{X}) \wedge \mathrm{p}[\mathrm{X}])$$

where some subset E = $\{e_1, e_2, \ldots, e_m\}$ of D satisfies q, we can translate it into the formula $\mathrm{p}[e_1] \vee \mathrm{p}[e_2] \vee \ldots \vee \mathrm{p}[e_m]$.

3. Consider the following expression:

$$\exists \mathrm{X}\ \forall \mathrm{Y}\ \mathrm{p}[\mathrm{X}, \mathrm{Y}]$$

This expression can be transformed into the following function-free expression:

$$(\forall Y_1\ \mathrm{p}[a_1, Y_1]) \vee (\forall Y_2\ \mathrm{p}[a_2, Y_2]) \vee \cdots \vee (\forall Y_n\ \mathrm{p}[a_n, Y_n])$$

4. The expression for uniqueness often has the following form:

$$(\forall X_1 X_2 Y_1 Y_2\ (\mathrm{p}(X_1, Y_1) \wedge \mathrm{p}(X_2, Y_2) \wedge X_1 \neq X_2 \rightarrow Y_1 \neq Y_2)$$

We may replace it by the following set of formulas:

$$\begin{gathered}\forall \mathrm{X}\ \neg\mathrm{p}(a_1, \mathrm{X}) \vee \neg\mathrm{p}(a_2, \mathrm{X}) \\ \forall \mathrm{X}\ \neg\mathrm{p}(a_1, \mathrm{X}) \vee \neg\mathrm{p}(a_3, \mathrm{X}) \\ \vdots \\ \forall \mathrm{X}\ \neg\mathrm{p}(a_1, \mathrm{X}) \vee \neg\mathrm{p}(a_n, \mathrm{X})\end{gathered}$$

$$\forall \mathrm{X}\ \neg \mathrm{p}(a_2, \mathrm{X}) \vee \neg \mathrm{p}(a_3, \mathrm{X})$$
$$\vdots$$
$$\forall \mathrm{X}\ \neg \mathrm{p}(a_2, \mathrm{X}) \vee \neg \mathrm{p}(a_n, \mathrm{X})$$
$$\vdots$$
$$\forall \mathrm{X}\ \neg \mathrm{p}(a_{n-1}, \mathrm{X}) \vee \neg \mathrm{p}(a_n, \mathrm{X})$$

5. For the expression:

$$\exists X_1 X_2 \cdots X_m\ \mathrm{p}(X_1, X_2, \ldots, X_m)$$

we may end up with n^m ground predicates OR'ed together. This results in exponential increase in size. One way to transform it with polynomial size is to add extra predicate symbols to represent partial expressions [PG86].

Note that the transformation can be done mechanically.

9.1.3 Examples

We illustrate the problem solving capability of CLIN by solving two kinds of problems: constraint satisfaction problems and logic puzzles.

9.1.3.1 Constraint Satisfaction Problems (CSPs)

Constraint Satisfaction Problems are fundamental problems in the area of Artificial Intelligence. They were originated from computer vision tasks, and have attracted a lot of research efforts [Mac77][Fre78][dK86b][DePe88][dK89]. For details, Mackworth [Mac87] gives an excellent introduction to CSPs.

A constraint satisfaction problem (CSP) is specified by a set of variables $x_1, \ldots, x_n$ and a set of constraints on subsets of these variables limiting the values they can take on. $C_{ijk\ldots}$ denotes the constraint among variables x_i, x_j, x_k, ... Usually, every variable x_i is restricted to a finite set D_i of values $\{a_{i1}, \ldots, a_{in}\}$.

The constraint satisfaction task consists of finding all sets of values $a_{1j_1}, \ldots, a_{1j_n}$ for $x_1, \ldots, x_n$ that simultaneously satisfy all the given constraints. Consider the following example from Freuder [Fre78]. We are given three variables x_1, x_2, x_3 with domains $D_1 =$ {a, b}, $D_2 =$ {e, f}, and $D_3 =$ {c, d, g} and binary constraints $C_{12} =$ {be, bf}, $C_{13} =$ {bc, bd, bg}, and $C_{23} =$ {ed, fg}. This CSP has two solutions:

$$x_1 = \text{b},\ x_2 = \text{e},\ x_3 = \text{d}$$
$$x_1 = \text{b},\ x_2 = \text{f},\ x_3 = \text{g}$$

CSPs are typically solved by backtracking search. However, as Mackworth [Mac77] pointed out, backtracking suffers from numerous maladies often making it "grotesquely inefficient." These maladies includes futile backtracks. Constraint preprocessing has been proposed to eliminate such maladies [Mac77][Fre78]. However, the expense of the preprocessing to avoid all futile backtracks typically outweighs the cost of the futile backtracks in the first place [dK89].

CLIN seems to be able to solve CSPs efficiently. One important feature of CLIN is to eliminate futile duplications which include futile backtracks. We don't need preprocessing as was required in [Mac77][Fre78].

Example 9.1 Let's solve Freuder's example mentioned above. For clarity, we restate the problem here. Given three variables x_1, x_2, x_3 with domains D_1 = {a, b}, D_2 = {e, f}, and D_3 = {c, d, g} and binary constraints C_{12} = {be, bf}, C_{13} = {bc, bd, bg}, and C_{23} = {ed, fg}, find the values of x_1, x_2, and x_3 to meet the constraints. Suppose we use predicate v_i for variable x_i and predicate $concat_{ij}$ for concatenation between x_i and x_j. These constraints can be represented by a first-order logic formula which is the conjunction of the following first-order logic expressions:

- $\exists X_1 \exists X_2 \exists X_3\ (v_1(X_1) \wedge v_2(X_2) \wedge v_3(X_3) \wedge concat_{12}(X_1, X_2) \wedge concat_{13}(X_1, X_3) \wedge concat_{23}(X_2, X_3))$
- $concat_{12}(\text{b}, \text{e}) \vee concat_{12}(\text{b}, \text{f})$
- $concat_{13}(\text{b}, \text{c}) \vee concat_{13}(\text{b}, \text{d}) \vee concat_{13}(\text{b}, \text{g})$
- $concat_{23}(\text{e}, \text{d}) \vee concat_{23}(\text{f}, \text{g})$
- $\neg concat_{12}(\text{a}, \text{e})$
- $\neg concat_{12}(\text{a}, \text{f})$
- $\neg concat_{13}(\text{a}, \text{c})$
- $\neg concat_{13}(\text{a}, \text{d})$
- $\neg concat_{13}(\text{a}, \text{g})$
- $\neg concat_{23}(\text{e}, \text{c})$
- $\neg concat_{23}(\text{e}, \text{g})$

- $\neg concat_{23}$(f, c)
- $\neg concat_{23}$(f, d)

Applying the transformation rules followed by Skolemization, we convert this first-order logic formula to the following clauses:

1. $\{v_1(\text{a}),\ v_1(\text{b})\}$
2. $\{v_2(\text{e}),\ v_2(\text{f})\}$
3. $\{v_3(\text{c}),\ v_3(\text{d}),\ v_3(\text{g})\}$
4. $\{\neg v_1(\text{X}),\ \neg v_2(\text{Y}),\ concat_{12}(\text{X, Y})\}$
5. $\{\neg v_1(\text{X}),\ \neg v_3(\text{Y}),\ concat_{13}(\text{X, Y})\}$
6. $\{\neg v_2(\text{X}),\ \neg v_3(\text{Y}),\ concat_{23}(\text{X, Y})\}$
7. $\{concat_{12}(\text{b, e}),\ concat_{12}(\text{b, f})\}$
8. $\{concat_{13}(\text{b, c}),\ concat_{13}(\text{b, d}),\ concat_{13}(\text{b, g})\}$
9. $\{concat_{23}(\text{e, d}),\ concat_{23}(\text{f, g})\}$
10. $\{\neg concat_{12}(\text{a, e})\}$
11. $\{\neg concat_{12}(\text{a, f})\}$
12. $\{\neg concat_{13}(\text{a, c})\}$
13. $\{\neg concat_{13}(\text{a, d})\}$
14. $\{\neg concat_{13}(\text{a, g})\}$
15. $\{\neg concat_{23}(\text{e, c})\}$
16. $\{\neg concat_{23}(\text{e, g})\}$
17. $\{\neg concat_{23}(\text{f, c})\}$
18. $\{\neg concat_{23}(\text{f, d})\}$

We run CLIN on these clauses. In the first run, CLIN outputs the following model in 9.766 seconds: v_1(b), v_2(f), v_3(g). We negate and add this model and run CLIN again. CLIN prints out another model in 10.516 seconds: v_1(b), v_2(e), v_3(d). We negate and add this model and run CLIN again. CLIN detects the clauses unsatisfiable in 3.983 seconds, indicating that there are no more solutions. Therefore, we have found all the solutions for Freuder's CSP example.

9.1.3.2 Logic Puzzles

Logic puzzles usually involve a set of constraints with finite domains. They are usually stated in words instead of formulae like constraint satisfaction problems.

Example 9.2 Here is a description for a logic puzzle "friends" [SS86]:

1. There are three friends: Michael, Richard, and Simon; three nationalities: American, Australian, and Israeli; three sports: basketball, cricket, and tennis.
2. Each friend has a unique nationality and plays a unique sport.
3. These friends came first, second, and third in a programming competition.
4. Michael likes basketball, and did better than the American.
5. Simon, the Israeli, did better than the tennis player.
6. The cricket player came first.

The first rule specifies three different domains. The rest rules are constraints. Given these rules, we want to find out who plays which sport, what is the nationality for each person, etc.. These rules can be expressed in first order logic as follows:

1. $\forall$P $\exists$N nationality(P, N)
   (each person has a nationality)
2. $\forall P_1, N_1, P_2, N_2$ (nationality(P_1, N_1) $\wedge$ nationality(P_2, N_2) $\wedge$ $P_1 \neq P_2 \rightarrow N_1 \neq N_2$)
   (uniqueness of nationality)
3. $\forall$P $\exists$ S play(P, S)
   (each person plays a sport)
4. $\forall P_1, S_1, P_2, S_2$ (play(P_1, S_1) $\wedge$ play(P_2, S_2) $\wedge$ $P_1 \neq P_2 \rightarrow S_1 \neq S_2$)
   (uniqueness of sport)
5. $\forall$P $\exists$O came(P, O)
   (each person came in an order)
6. $\forall P_1, O_1, P_2, O_2$ (came(P_1, O_1) $\wedge$ came(P_2, O_2) $\wedge$ $P_1 \neq P_2 \rightarrow O_1 \neq O_2$)
   (uniqueness of order)
7. play(michael,basketball)
8. $\forall$P (nationality(P, american) $\rightarrow$ did_better(michael, P))

9. nationality(simon,israeli)

10. $\forall$P (play(P, tennis) $\rightarrow$ did_better(simon, P))

11. $\forall P_1,P_2$ (did_better(P_1, P_2) $\vee$ did_better(P_2, P_1))

12. $\forall P_1,P_2$ (did_better(P_1, P_2) $\rightarrow$ $\neg$did_better(P_2, P_1))
    (exclusiveness of did_better)

13. $\forall P_1,P_2,P_3$ (did_better(P_1, P_2) $\wedge$ did_better(P_2, P_3) $\rightarrow$ did_better(P_1, P_3))
    (transitivity of did_better)

14. $\forall$P (play(P, cricket) $\rightarrow$ came(P, 1))

We transformed these first-order formulas into function-free clauses by applying the transformation rules followed by Skolemization. Then we run CLIN on these clauses. CLIN outputs a model M with the following literals contained in M:

- came(simon, 1)
- nationality(michael, australian)
- nationality(richard, american)
- nationality(simon, israeli)
- play(michael, basketball)
- play(richard, tennis)
- play(simon, cricket),
- did_better(michael, richard)
- did_better(simon, richard)

These literals indeed logically follow the above rules. Apparently, either Michael or Richard might come second. Simon might do better than Michael or Michael might do better than Simon, but not both. This can be verified by adding the negated literals above and run CLIN again. The total time for solving this problem is 29.783 seconds.

Example 9.3 The following constraints are for the "jobs" puzzle found in [WOLB84]:

1. There are four people: Roberta, Thelma, Pete, and Steve; eight jobs: actor, boxer, chef, guard, nurse, police officer, teacher, and telephone operator.

2. Each person holds exactly two jobs.
3. The job of nurse is held by a male.
4. The husband of the chef is the telephone operator.
5. Roberta is not a boxer.
6. Pete is not college educated.
7. Roberta, chef, and the police officer went golfing together.
8. The nurse, teacher, and police officer are all college educated.

A model found for the above constraints contains the following literals:

- job(roberta, guard)
- job(roberta, teacher),
- job(thelma, boxer)
- job(thelma, chef),
- job(pete, actor)
- job(pete, telephone_operator),
- job(steve, nurse)
- job(steve, police_officer),
- husband(pete, thelma)

These literals are indeed conclusions of the constraints. The total time for solving this problem is 351.683 seconds.

Example 9.4 The following rules are for a problem called "zebra" [SS86]:

1. There are five people: englishman, spaniard, norwegian, japanese, and ukranian; five houses: 1, 2, 3, 4, and 5. five drinks: orange, coffee, water, tea, and milk; five cigarettes: gold, kools, chesterfields, lucky, and parliaments; five animals: dog, fox, horse, snails, and zebra; five colors: yellow, red, blue, ivory, and green. Each person lives in a unique house, drinks a unique drink, owns a unique animal, smokes a unique cigarette, and each house is a unique color.

2. The englishman lives in the red house.
3. The spaniard owns a dog.
4. The norwegian lives in the first house.
5. Kools are smoked in the yellow house.
6. Chesterfields are smoked next to where the fox is kept.
7. The norwegian lives next to the blue house.
8. The gold smoker owns snails.
9. The lucky smoker drinks orange juice.
10. The ukranian drinks tea.
11. The japanese smokes parliaments.
12. The kools smoker lives next to where the horse horse is kept.
13. Coffee is drunk in the green house.
14. The green house is to the immediate right of the ivory house.
15. Milk is drunk in the middle house.

A model found for the above rules contains contains the following literals:

- color(1, yellow)
- color(2, blue)
- color(3, red)
- color(4, ivory)
- color(5, green)
- lives(norwegian, 1)
- lives(ukranian, 2)
- lives(english, 3)
- lives(spaniard, 4)
- lives(japanese, 5)

- owns(norwegian, fox)
- owns(ukranian, horse)
- owns(english, snails)
- owns(spaniard, dog)
- owns(japanese, zebra)
- smokes(norwegian, kools)
- smokes(ukranian, chesterfields)
- smokes(english, gold)
- smokes(spaniard, lucky)
- smokes(japanese, parliaments)
- drinks(norwegian, water)
- drinks(ukranian, tea)
- drinks(english, milk)
- drinks(spaniard, orange)
- drinks(japanese, coffee)

These literals indeed logically follow the rules. The total time for solving this problem is 1805.950 seconds.

One point we should mention here. These axiomatizations are fully declarative, and do not use negation as failure as Prolog does. Therefore, these formulations are more natural than Prolog ones, even though the times to solve the problems are often slower than Prolog times.

9.2 Database Design

In database area, people are talking about relational databases and deductive databases. Both databases have disadvantages. For relational databases, tuples or relations have to be prepared in different ways. New data can't be deduced from the existing relations [PP90]. The database management system is at most a retrieval instrument. For deductive databases, processing time for queries may be intolerable.

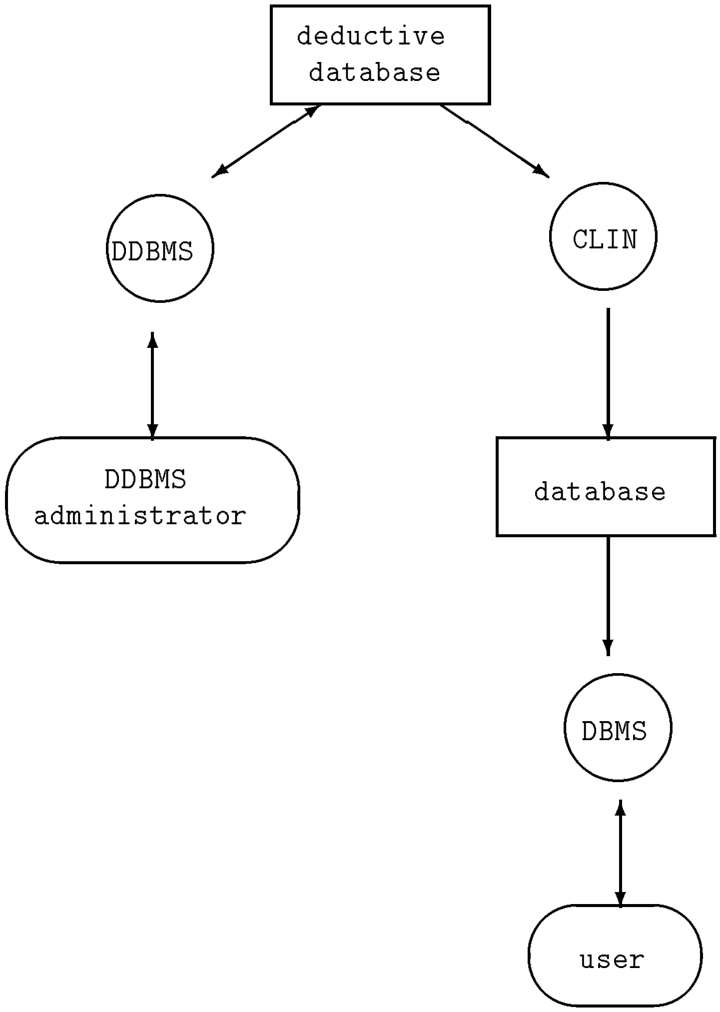

Figure 9.2: CLIN in a database system

A deductive database can be viewed as a constraint satisfaction problem. With problem solving capability, CLIN provides a design of a database that might take advantage of both deductive databases and relational databases. CLIN draws all conclusions (solutions) of a deductive database and stores them in a database. The user retrieves information via a DBMS (database management system) which is essentially an information retrieval system. The database administrator manages the deductive database via a DDBMS (deductive database management system). The main functions of DDBMS include validation, consistency checking, and updating. CLIN may update the database either periodically or by request. In this design, CLIN serves as a back-end processor and the DBMS serves as a front-end processor. The block diagram of this design is shown in Figure 9.2.

All the information stored in the database are sound conclusions of the deductive database; no other mechanisms such as negation as failure are involved in the derivation.

Therefore, negated information can be stored and queried. The processing time of a query is tolerable since the answer is retrieved directly from the database; no reasoning is involved. The data in deductive database is declarative, so they are concise and understandable.

Example 9.5 Consider the following simple deductive database:

1. {father(george, justice)}
2. {male(justice)}
3. {son(X, Y), ¬father(Y, X), ¬male(X)}
4. {daughter(X, Y), ¬father(Y, X), ¬female(X)}
5. {¬male(X), ¬female(X)}

The prover will find the following conclusions and put them in the database.

1. {father(george, justice)}
2. {male(justice)}
3. {¬female(justice)}
4. {son(justice, george)}
5. {¬daughter(justice, george)}

New data structures should be designed to allow alternative sets of conclusions. The DBMS should be able to retrieve such information from the database. The DDBMS has to be capable of detecting inconsistency and locating the inconsistent clauses.

If we allow default rules [Rei80], then common sense reasoning can be done. In this case, we may label each conclusion derived by CLIN in the database with the default rules it depends on. When new facts or rules are added into the deductive database, some conclusions may have to be retracted. The whole system is then essentially a conclusion-based truth maintenance system.

9.3 Debugging Axioms

One of the difficulties encountered in theorem proving concerns the adequacy and correctness of axioms for a given problem. Our system provides some help for this difficulty.

If the axioms are not adequate or incorrect, the given problem may be satisfiable and CLIN will find a model. By looking at the model, one may locate the incorrect axioms or figure out what axioms are missing.

Example 9.6 Consider the following problem:

1. {person(roberta)}
2. {person(steve)}
3. {job(teacher)}
4. {job(nurse)}
5. {¬person(P), hasjob(P, teacher), hasjob(P, nurse)}
6. {¬hasjob(roberta, teacher), ¬hasjob(roberta, nurse)}
7. {¬hasjob(steve, teacher), ¬hasjob(steve, nurse)}
8. {¬job(J), hasjob(roberta, J), hasjob(steve, J)}
9. {¬hasjob(roberta, teacher), ¬hasjob(steve, teacher)}
10. {¬hasjob(roberta, nurse), ¬hasjob(steve, nurse)}
11. {¬person(P), ¬hasjob(P, nurse), male(P)}
12. {¬person(P), male(P), female(P)}
13. {female(roberta)}
14. {male(steve)}
15. {¬hasjob(roberta, teacher)}

CLIN finds a model which includes hasjob(roberta, nurse). From clause 11, we know that if a person is a nurse, then this person is a male. From clause 13, we also know that Roberta is a female. Therefore, we conclude that Roberta was allowed to be a male and a female simultaneously. Obviously, an axiom which denies a person being a male and a female at the same time was missing. The axiom can be

$$\{\neg\text{person}(P), \neg\text{male}(P), \neg\text{female}(P)\}$$

If we add this axiom, the problem is then unsatisfiable.

We used this idea to find out missing or incorrect axioms when we solved puzzles zebra, friends, and jobs.

9.4 Propositional Temporal Logic

Special methods for temporal logic are known [SC85][Pla86][Aba87], but it seems that a general theorem prover might perform better in some cases, and also permit an easier transition to first-order temporal logic, for which no decision procedure exists. We have applied CLIN to temporal logic theorems; the use of replace rules provides a new paradigm for proving these theorems. The proofs obtained by CLIN are more natural and are more similar to those obtained by a human being.

We concentrate on one specific temporal logic that has been proposed in the context of concurrent program verification [MP83]. In this logic, time is discrete, linear, and extends infinitely toward the future. Currently, we only pay attention to propositional temporal logic (PTL).

9.4.1 Introduction to Propositional Temporal Logic

The alphabet of PTL consists of:

- Proposition symbols;
- Connectives: $true$, $false$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\equiv$;
- modal operators: $\Box$, $\Diamond$, $\bigcirc$, and $\mathbf{U}$;
- auxiliary symbols: "(" (left parenthesis) and ")" (right parenthesis).

Informally, $\Box$ means "henceforth", $\Diamond$ means "eventually", $\bigcirc$ means "in the next state", and **U** means "until".

The formation rules for a PTL formula are:

- A proposition symbol is a formula;
- $true$ and $false$ are formulae;
- If A and B are formulae then so are $\neg$A, (A$\wedge$B), (A$\vee$B), (A$\rightarrow$B), (A$\equiv$B), $\Box$A, $\Diamond$A, $\bigcirc$A, and (A**U**B).

As usual, we may omit parentheses if no ambiguities occur.

For example, the following two expressions are PTL formulae:

$$\mathrm{w} \rightarrow \Diamond\ \mathrm{w}$$
$$\bigcirc\ (w_1 \wedge w_2) \equiv (\bigcirc\ w_1 \wedge \bigcirc\ w_2)$$

The semantics of the PTL formulae is defined, following [Pla86], in terms of a structure M = (S,D,L) where S is a sequence (s_1, s_2, s_3, ...) of states, D is a set of proposition symbols, and L is an interpretation which assigns to each proposition symbol w and each state s_i a mapping L(w,i) $\in$ {TRUE, FALSE}. We assume that all proposition symbols are state symbols, i.e. the mapping L(w,i) depends on i.

We write (M,i) $\models$ A to mean that temporal formula A is true in structure M at state s_i. The relation $\models$ is defined inductively as follows:

1. (M,i) $\models$ w iff L(w,i) = *true*.
2. (M,i) $\models$ (A $\vee$ B) iff (M,i) $\models$ A or (M,i) $\models$ B.
3. (M,i) $\models$ (A $\wedge$ B) iff (M,i) $\models$ A and (M,i) $\models$ B.
4. (M,i) $\models$ $\neg$A iff not (M,i) $\models$ A.
5. (M,i) $\models$ (A $\rightarrow$ B) iff whenever (M,i) $\models$ A then (M,i) $\models$ B.
6. (M,i) $\models$ (A $\equiv$ B) iff (M,i) $\models$ (A $\rightarrow$ B) and (M,i) $\models$ (B $\rightarrow$ A).
7. (M,i) $\models$ $\Box$A iff for all j, j $\geq$ i, (M,j) $\models$ A.
8. (M,i) $\models$ $\Diamond$A iff there exists j, j $\geq$ i, such that (M,j) $\models$ A.
9. (M,i) $\models$ $\bigcirc$A iff (M,i+1) $\models$ A.
10. (M,i) $\models$ (A**U**B) iff there exists k , k $\geq$ i, such that (M,k) $\models$ B and for all j, i $\leq$ j $<$ k, (M,j) $\models$ A.

We write M $\models$ A as an abbreviation for (M,1) $\models$ A. If M $\models$ A for all structures M, then we say A is *valid*. If there exists a structure M such that M $\models$ A then we say A is *satisfiable*. If no structures satisfy A, then we say A is *unsatisfiable*.

9.4.2 Replace Rules for Connectives and Modal Operators

A PTL formula doesn't express the time explicitly, while the semantics of PTL formulae expressed the time by states. In order to apply our system, we express the time explicitly. Suppose A is a PTL formula. We use at(A, T) with some arbitrary time T to denote the status of A at state s_i. We use predecessor and successor functions, *pred* and *suc* respectively, to denote time sequence. Thus if T corresponds to s_i, then *suc*(T) corresponds to s_{i+1}, *suc*(*suc*(T)) corresponds to s_{i+2}, ... The correspondent of L(w, i) is L(w,T_i) where T_i corresponds to s_i. The semantics of a PTL formula can then be simplified as follows:

1. at(w, T) iff L(w, T) = *true*.
2. at(A∨B, T) iff at(A, T) or at(B, T).
3. at(A∧B, T) iff at(A, T) and at(B, T).
4. at(¬A, T) iff ¬at(A, T).
5. at(A→B, T) iff whenever at(A, T) then at(B, T).
6. at(A≡B, T) iff at(A→B, T) and at(B→A, T).
7. at(□A, T) iff for all T_1, $T_1 \geq$ T, at(A, T_1).
8. at(◇A, T) iff there exists T_1, $T_1 \geq$ T, such that at(A, T_1).
9. at(◯A, T) iff at(A, *suc*(T)).
10. at(A**U**B, T) iff there exists T_1, $T_1 \geq$ T, such that at(B, T_1) and for all T_2, T $\leq T_2 < T_1$, at(A, T_2).

The above formalism is ideally suited to replace rules. In the following, we show the replace rules for these definitions.

1. For ∨, we have

$$\text{at}(A \vee B,\ T) \rightarrow \text{at}(A,\ T),\ \text{at}(B,\ T)$$

2. For negation of ∨, we have

$$\text{at}(\neg(A \vee B),\ T) \rightarrow \text{at}(\neg A,\ T)$$
$$\text{at}(\neg(A \vee B),\ T) \rightarrow \text{at}(\neg B,\ T)$$

3. For ∧, we have

$$\text{at}(A \wedge B,\ T) \rightarrow \text{at}(A,\ T)$$
$$\text{at}(A \wedge B,\ T) \rightarrow \text{at}(B,\ T)$$

4. For negation of ∧, we have

$$\mathrm{at}(\neg(A \wedge B), T) \rightarrow \mathrm{at}(\neg A, T), \mathrm{at}(\neg B, T)$$

5. For $\neg\neg$, we have

$$\mathrm{at}(\neg\neg A, T) \rightarrow \mathrm{at}(A, T)$$

6. For $\rightarrow$, we have

$$\mathrm{at}(A \rightarrow B, T) \rightarrow \mathrm{at}(\neg A, T), \mathrm{at}(B, T)$$

7. For negation of $\rightarrow$.

$$\mathrm{at}(\neg(A \rightarrow B), T) \rightarrow \mathrm{at}(A, T)$$
$$\mathrm{at}(\neg(A \rightarrow B), T) \rightarrow \mathrm{at}(\neg B, T)$$

8. For $\equiv$, we have

$$\mathrm{at}(A \equiv B, T) \rightarrow \mathrm{at}(A \wedge B, T), \mathrm{at}(\neg A \wedge \neg B, T)$$

9. For negation of $\equiv$, we have

$$\mathrm{at}(\neg(A \equiv B), T) \rightarrow \mathrm{at}(\neg A \wedge B, T), \mathrm{at}(A \wedge \neg B, T)$$

10. For $\neg$, we have

$$\mathrm{at}(\neg \mathrm{prop}(W), T) \rightarrow \neg \mathrm{at}(\mathrm{prop}(W), T)$$

Notice that we use prop(w) to indicate a proposition symbol w. We don't want to pull out negation, we want to push the negation in as far as possible instead.

11. For $\diamond$, we have

$$\mathrm{at}(\diamond A, T) \rightarrow \mathrm{at}(A, \mathrm{first}(A, T))$$
$$\mathrm{at}(\diamond A, T) \rightarrow \mathrm{first}(A, T) \geq T$$

Here first(A, T) means the first time that something is true, in this case A is true. This allows an induction rule for $\diamond$ which is needed in some cases. The induction rule says that A is true at first(A, T) and is false before first(A, T). So we have

$$\mathrm{at}(\diamond A, T), U \geq T, \neg(U \geq \mathrm{first}(A, T)) \rightarrow \mathrm{at}(\neg A, U)$$

12. For negation of $\diamond$, we have

$$\text{at}(\neg\Diamond A,\ T) \rightarrow \text{at}(\Box(\neg A),\ T)$$

13. For $\Box$, we have

$$\text{at}(\Box A,\ T),\ U \geq T \rightarrow \text{at}(A,\ U)$$

14. For negation of $\Box$, we have

$$\text{at}(\neg\Box A,\ T) \rightarrow \text{at}(\Diamond(\neg A),\ T)$$

15. For $\bigcirc$, we have

$$\text{at}(\bigcirc A,\ T) \rightarrow \text{at}(A,\ \mathit{suc}(T))$$
$$\text{at}(\bigcirc A,\ \mathit{pred}(T)) \rightarrow \text{at}(A,\ T)$$

The second rule is sometimes useful.

16. For negation of $\bigcirc$, we have

$$\text{at}(\neg\bigcirc A,\ T) \rightarrow \text{at}(\bigcirc(\neg A),\ T)$$

17. For **U**, we have

$$\text{at}(A\mathbf{U}B,\ T) \rightarrow \text{at}(B,\ \text{first}(B,\ T))$$
$$\text{at}(A\mathbf{U}B,\ T) \rightarrow \text{first}(B,\ T) \geq T$$

Again, we need induction rules for **U** sometimes:

$$\text{at}(A\mathbf{U}B,\ T),\ V \geq T,\ \neg(V \geq \text{first}(B,\ T)) \rightarrow \text{at}(A,\ V)$$
$$\text{at}(A\mathbf{U}B,\ T),\ V \geq T,\ \neg(V \geq \text{first}(B,\ T)) \rightarrow \text{at}(\neg B,\ V)$$

The first rule says B is false before first(B, T). The second rule says A is true before first(B, T).

18. For negation of **U**, the replace rules are more complicated. We have two first times, and these two times results in two induction rules. The replace rules are:

$$\text{at}(\neg(A\mathbf{U}B),\ T) \rightarrow \text{at}(\Box(\neg B),\ T),\ \text{at}(A\bar{\mathbf{U}}B,\ T)$$
$$\text{at}(A\bar{\mathbf{U}}B,\ T) \rightarrow \neg(T \geq \text{first}(B,\ T))$$
$$\text{at}(A\bar{\mathbf{U}}B,\ T) \rightarrow \text{at}(B,\ \text{first}(B,\ T))$$
$$\text{at}(A\bar{\mathbf{U}}B,\ T),\ V \geq T,\ \neg(V \geq \text{first}(B,\ T)) \rightarrow \text{at}(\neg B,\ V)$$
$$\text{at}(A\bar{\mathbf{U}}B,\ T) \rightarrow \text{first}(\neg A,\ T) \geq T$$
$$\text{at}(A\bar{\mathbf{U}}B,\ T) \rightarrow \neg(\text{first}(\neg A,\ T) \geq \text{first}(B,\ T))$$
$$\text{at}(A\bar{\mathbf{U}}B,\ T) \rightarrow \text{at}(\neg A,\ \text{first}(\neg A,\ T))$$
$$\text{at}(A\bar{\mathbf{U}}B,\ T),\ V \geq T,\ \neg(V \geq \text{first}(\neg A,\ T)) \rightarrow \text{at}(A,\ V)$$

We use $\bar{\mathbf{U}}$ as an intermediate expression.

9.4.3 Auxiliary Rules

It's hard to know what auxiliary axioms are needed in order to prove any PTL theorems. Here we listed those that are currently used.

$$
\begin{gathered}
\mathrm{X} > \mathrm{Y} \rightarrow \mathrm{X} \geq \mathrm{Y} \\
<\mathrm{X} \geq \mathrm{T}>,\ <\mathrm{Y} \geq \mathrm{T}> \rightarrow \mathrm{X} \geq \mathrm{Y},\ \mathrm{Y} \geq \mathrm{X} \\
<\mathrm{X} \geq \mathrm{T}>,\ <\mathrm{Y} \geq \mathrm{T}> \rightarrow \mathrm{X} \geq \mathrm{Y},\ \neg(\mathrm{X} \geq \mathrm{Y}) \\
\mathrm{X} \geq \mathrm{Y},\ <\mathrm{at(next(W),\ T)}> \rightarrow \mathit{suc}(\mathrm{X}) \geq \mathrm{Y} \\
\mathrm{X} \geq \mathrm{Y},\ <\mathrm{at(next(W),\ T)}> \rightarrow \mathit{suc}(\mathrm{X}) \geq \mathit{suc}(\mathrm{Y}) \\
\mathrm{X} \geq \mathit{suc}(\mathrm{Y}),\ <\mathrm{at(next(W),\ T)}> \rightarrow \mathit{pred}(\mathrm{X}) \geq \mathrm{Y} \\
\mathrm{X} \geq \mathrm{Y},\ \mathrm{Y} \geq \mathrm{Z} \rightarrow \mathrm{X} \geq \mathrm{Z} \\
\neg(\mathrm{X} \geq \mathrm{Y}),\ \mathrm{Z} \geq \mathrm{Y} \rightarrow \neg(\mathrm{X} \geq \mathrm{Z}) \\
\neg(\mathrm{Y} \geq \mathrm{Z}),\ \mathrm{Y} \geq \mathrm{X} \rightarrow \neg(\mathrm{X} \geq \mathrm{Z}) \\
\mathrm{X} \geq \mathrm{Y},\ \mathrm{at(W,\ Y)},\ <\mathrm{at(next(V),\ T)}> \rightarrow \mathrm{at(W,\ X)},\ \mathrm{X} \geq \mathit{suc}(\mathrm{Y}) \\
\mathrm{X} \geq \mathrm{Y},\ \mathrm{at(W,\ X)},\ <\mathrm{at(next(V),\ T)}> \rightarrow \mathrm{at(W,\ Y)},\ \mathrm{X} \geq \mathit{suc}(\mathrm{Y}) \\
<\mathrm{X} \geq \mathrm{X}>,\ <\mathrm{at(next(W),\ T)}> \rightarrow \neg(\mathit{pred}(\mathrm{X}) \geq \mathrm{X})
\end{gathered}
$$

The first rule is a repeated replace rule, and the rest are once replace rules. The literals bracketed by $<$ and $>$ are context literals.

9.4.4 Example Proofs

We give two examples here for illustration. For clarity, we cross out the "prop" that is used with proposition symbols.

Example 9.7 Consider the following theorem:

$$\Box\bigcirc \mathrm{w} \rightarrow \bigcirc\Box \mathrm{w}$$

Let's see how we human beings prove this theorem by contradiction. Assuming the theorem is false. Then $\bigcirc\Box$w is false and $\Box\bigcirc$w is true at state s_1. From the definition of $\bigcirc$, we know $\Box$w is false in the next state s_2. From the definition of $\Box$ we know that there exists a state s_i, i $\geq$ 2, such that w is false in state s_i. Since $\Box\bigcirc$w is true, we know $\bigcirc$w is true in every state s_j, j $\geq$ 1. This implies that w is true in every state s_k, k $\geq$ 2. This contradicts that w is false in some state s_i, i $\geq$ 2.

Now let's see how CLIN proves it. We call the arbitrary start time "now". The negation of the theorem is expressed by

- C_1: $\{\text{at}(\neg(\Box\bigcirc \text{w} \rightarrow \bigcirc\Box \text{w}), \text{now})\}$

From C_1, we get

- C_2: $\{\text{at}(\Box\bigcirc \text{w}, \text{now})\}$
- C_3: $\{\text{at}(\neg\bigcirc\Box \text{w}, \text{now})\}$

From C_3, we get

- C_4: $\{\text{at}(\bigcirc\neg\Box \text{w}, \text{now})\}$

From C_4, we get

- C_5: $\{\text{at}(\neg\Box \text{w}, suc(\text{now}))\}$

From C_5, we get

- C_6: $\{\text{at}(\Diamond\neg \text{w}, suc(\text{now}))\}$

From C_6, we get

- C_7: $\{\text{first}(\neg \text{w}, suc(\text{now})) \geq suc(\text{now})\}$
- C_8: $\{\text{at}(\neg \text{w}, \text{first}(\neg \text{w}, suc(\text{now})))\}$

From C_8, we get

- C_9: $\{\neg\text{at}(\text{w}, \text{first}(\neg \text{w}, suc(\text{now})))\}$

From C_7, we get

- C_{10}: $\{pred(\text{first}(\neg \text{w}, suc(\text{now}))) \geq \text{now}\}$

From C_2 and C_{10}, we have

- C_{11}: $\{\text{at}(\bigcirc \text{w}, pred(\text{first}(\neg \text{w}, suc(\text{now}))))\}$

From C_{11}, we have

- C_{12}: {at(w, first(¬w, suc(now)))}

C_{12} and C_9 are contradictory. Note that this proof is very similar to the human's proof except the treatment of states. In the human's proof, we didn't specify explicitly at which state w was false. In the proof obtained by CLIN, this particular state was represented explicitly by first(¬w, suc(now)). This example demonstrates that our proving procedure for temporal logic is human-oriented and understandable.

Example 9.8 Here we consider a more complicated theorem:

$$(\Box w_1 \wedge \Diamond w_2) \rightarrow w_1 \mathbf{U} w_2$$

The following are the clauses obtained by CLIN:

- C_1: $\{\text{at}(\neg((\Box w_1 \wedge \Diamond w_2) \rightarrow w_1 \mathbf{U} w_2), \text{now})\}$
- C_2: $\{\text{at}(\Box\neg w_2, \text{now}), \text{at}(w_1 \bar{\mathbf{U}} w_2, \text{now})\}$
- C_3: $\{\neg\text{at}(w_1 \bar{\mathbf{U}} w_2, \text{now}), \text{first}(w_2, \text{now}) \geq \text{now}\}$
- C_4: $\{\text{at}(w_2, \text{first}(w_2, \text{now}))\}$
- C_5: $\{\neg\text{at}(w_1 \bar{\mathbf{U}} w_2, \text{now}), \text{first}(\neg w_1, \text{now}) \geq \text{now}\}$
- C_6: $\{\neg\text{at}(w_1 \bar{\mathbf{U}} w_2, \text{now}), \neg(\text{first}(\neg w_1, \text{now}) \geq \text{first}(w_2, \text{now}))\}$
- C_7: $\{\neg\text{at}(w_1 \bar{\mathbf{U}} w_2, \text{now}), \text{at}(\neg w_1, \text{first}(\neg w_1, \text{now}))\}$
- C_8: $\{\neg\text{at}(\neg w_1, \text{first}(\neg w_1, \text{now})), \neg\text{at}(w_1, \text{first}(\neg w_1, \text{now}))\}$
- C_9: $\{\text{first}(w_2, \text{now}) \geq \text{now}\}$
- C_{10}: $\{\text{at}(w_2, \text{first}(w_2, \text{now}))\}$
- C_{11}: $\{\neg\text{at}(\Box\neg w_2, \text{now}), \text{at}(\neg w_2, \text{first}(w_2, \text{now}))\}$
- C_{12}: $\{\neg\text{at}(\neg w_2, \text{first}(w_2, \text{now})), \neg\text{at}(w_2, \text{first}(w_2, \text{now}))\}$
- C_{13}: $\{\neg(\text{first}(\neg w_1, \text{now}) \geq \text{now}), \text{at}(w_1, \text{first}(\neg w_1, \text{now}))\}$

These clauses are contradictory. This is proved by the propositional calculus prover. However, it is not hard to derive a contradiction by hand. Let's do case analysis on C_2. If at($\Box\neg w_2$, now) is true, then at($\neg w_2$, first(w_2, now)) of C_{11} is true, then $\neg$at(w_2, first(w_2, now)) of C_{12} is true which contradicts C_{10}. On the other hand, if at($w_1\bar{\mathbf{U}}w_2$, now) is true, then first($\neg w_1$, now) $\geq$ now of C_5 is true, then at($\neg w_1$, first($\neg w_1$, now)) of C_7 is true, then $\neg$at(w_1, first($\neg w_1$, now)) of C_8 is true, then at(w_1, first($\neg w_1$, now)) of C_{13} is true, and we got a contradiction. This proof is also very natural. We human beings also need case analysis.

9.4.5 Proved PTL Theorems

We have proved some PTL theorems. The times are listed in Table 9.1. These theorems are obtained from [Pla86][Ost89]. All are proved by turning off the hyper-linking phase and by ground substitution in replacement. All use the same set of axioms. Sometimes to prove A $\equiv$ B we proved A $\rightarrow$ B and B $\rightarrow$ A separately.

9.4.6 Discussion

It is possible to have nontermination of replace rules if care is not taken. For example, suppose the theorem contains the unit clause at($\Box\Diamond$p, now). Suppose we also have the following replace rules:

$$\begin{gathered}
\text{at}(\Box\text{F, T}),\ \text{U} \geq \text{T} \rightarrow \text{at(F, U)} \\
\text{at}(\Diamond\text{F, T}) \rightarrow \text{at(F, first(F, T))} \\
\text{at}(\Diamond\text{F, T}) \rightarrow \text{first(F, T)} \geq \text{T} \\
\text{X} \geq \text{Y},\ \text{Y} \geq \text{Z} \rightarrow \text{X} \geq \text{Z}
\end{gathered}$$

Then we can derive the literals

- at($\Diamond$p, now)
- at(p, first(p, now))
- first(p, now) $\geq$ now
- at($\Diamond$p, first(p, now))
- at(p,first(p, first(p, now))
- first(p, first(p, now)) $\geq$ first(p, now)
- first(p, first(p, now)) $\geq$ now

no	problem	cpu(sec)
1	$\Diamond\Box w \to \Box\Diamond\Box w$	6.383
2	$\neg\Diamond w \equiv \Box\neg w$	6.617
3	$\Diamond w \vee \Diamond\neg w$	0.683
4	$w \to \Diamond w$	0.500
5	$\bigcirc w \to \Diamond w$	1.783
6	$\Box w \equiv \Box\Box w$	3.050
7	$\Diamond w \equiv \Diamond\Diamond w$	3.367
8	$\Diamond\neg w \equiv \neg\Box w$	1.583
9	$\Box(w_1 \to w_2) \to (\Diamond w_1 \to \Diamond w_2)$	3.067
10	$\Box(w_1 \wedge w_2) \equiv (\Box w_1) \wedge (\Box w_2)$	10.250
11	$\Diamond(w_1 \vee w_2) \equiv (\Diamond w_1) \vee (\Diamond w_2)$	9.717
12	$(\Box w_1 \vee \Box w_2) \to \Box(w_1 \vee w_2)$	3.250
13	$\Diamond(w_1 \wedge w_2) \to (\Diamond w_1 \wedge \Diamond w_2)$	3.317
14	$(\Box w_1 \wedge \Diamond w_2) \to \Diamond(w_1 \wedge w_2)$	2.967
15	$\bigcirc(w_1 \wedge w_2) \equiv (\bigcirc w_1 \wedge \bigcirc w_2)$	8.283
16	$\bigcirc(w_1 \vee w_2) \equiv (\bigcirc w_1 \vee \bigcirc w_2)$	8.317
17	$\bigcirc(w_1 \to w_2) \equiv (\bigcirc w_1 \to \bigcirc w_2)$	8.367
18	$\bigcirc(w_1 \equiv w_2) \equiv (\bigcirc w_1 \equiv \bigcirc w_2)$	19.033
19	$\bigcirc\Box w \equiv \Box \bigcirc w$	8.800
20	$\bigcirc\Diamond w \equiv \Diamond \bigcirc w$	9.000
21	$\Box\Diamond\Box w \equiv \Diamond\Box w$	46.983
22	$\Diamond\Box\Diamond w \equiv \Box\Diamond w$	44.467
23	$\Box w \equiv (w \wedge \bigcirc\Box w)$	30.867
24	$\Diamond w \equiv (w \vee \bigcirc\Diamond w)$	27.167
25	$(w \wedge \Diamond\neg w) \to \Diamond(w \wedge \bigcirc\neg w)$	117.167
26	$(\neg w)\mathbf{U}w \equiv \Diamond w$	6.700
27	$\Box w_1 \wedge \Diamond w_2 \to w_1\mathbf{U}w_2$	6.350
28	$(w_1\mathbf{U}w_2)\mathbf{U}w_2 \to (w_1\mathbf{U}w_2)$	14.833
29	$w_1\mathbf{U}(w_1\mathbf{U}w_2) \to (w_1\mathbf{U}w_2)$	25.133
30	$(\Box w_1 \wedge (w_2\mathbf{U}w_3)) \to (w_1 \wedge w_2)\mathbf{U}(w_1 \wedge w_3)$	34.983
31	$(w_1 \wedge w_2)\mathbf{U}w_3 \equiv (w_1\mathbf{U}w_3) \wedge (w_2\mathbf{U}w_3)$	27.033
32	$w_1\mathbf{U}(w_2 \vee w_3) \equiv (w_1\mathbf{U}w_2) \vee (w_1\mathbf{U}w_3)$	236.183
33	$(\Diamond w_1 \vee \Diamond w_2) \to (((\neg w_1)\mathbf{U}w_2) \vee ((\neg w_2)\mathbf{U}w_1))$	119.367
34	$w_1\mathbf{U}(w_2 \wedge w_3) \to (w_1\mathbf{U}w_2) \wedge (w_1\mathbf{U}w_3)$	32.417
35	$(w_1\mathbf{U}w_3) \vee (w_2\mathbf{U}w_3) \to (w_1 \vee w_2)\mathbf{U}w_3$	11.817
36	$(w_1 \to w_2)\mathbf{U}w_3 \to (w_1\mathbf{U}w_3 \to w_2\mathbf{U}w_3)$	10.000
37	$(w_1\mathbf{U}w_2) \wedge ((\neg w_2)\mathbf{U}w_3) \to w_1\mathbf{U}w_3$	33.583
38	$(w_1\mathbf{U}w_2)\mathbf{U}w_3 \to (w_1 \vee w_2)\mathbf{U}w_3$	20.267
39	$w_1\mathbf{U}(w_2\mathbf{U}w_3) \to (w_1 \vee w_2)\mathbf{U}w_3$	33.367

Table 9.1: Times for sample propositional temporal theorems

- at($\Diamond$p, first(p, first(p, now)))
- ...

and so on. Of course, if we use sliding priority, this nontermination won't occur.

We have often found that in order to prove A $\equiv$ B, it is much faster to prove A implies B and B implies A separately. This could be built in to CLIN automatically if desired.

One advantage of this formalism is that by modifying the axioms and strategies, we obtain many different proof procedures for temporal logic. Correctness is not a problem because the underlying search is done by a theorem prover. This contrasts with special purpose procedures where correctness may be more of an issue. A disadvantage of our formalism is that we do not have a decision procedure. This means that even if a formula is satisfiable, we may spend a long time looking for a proof. This could be overcome, we feel, by setting a bound on the maximum term size needed to find a proof. However, if a proof cannot be found in a reasonable time, it is not clear that such a bound would help.

Chapter 10

Conclusion

This chapter summarizes our work and discusses some possible extensions to our system.

10.1 Summary

Our work was motivated by the observation that duplication of instances of clauses might be a significant cause of the inefficiency of a theorem proving method. We have shown how the system CLIN was built to eliminate such duplication. The system is based on the hyper-linking strategy prover plus some refinements. We have demonstrated the effectiveness of our system by comparing it with other major proving systems.

We have presented the hyper-linking strategy prover in detail. The instances generated in the hyper-linking stage don't combine parts of separate clauses, hence the duplication caused by combination is totally avoided. The propositional calculus prover involves two cases at most per literal and avoids the execution of unnecessary cases using a technique similar to dependency-directed backtracking. The propositional calculus prover works so well that the overhead due to the duplication by case analysis is negligible most of the time.

The refinements that have been presented to improve the hyper-linking strategy prover include unit simplification, support sets, small proof checking, predicate replacement, sliding priority, etc. Among these refinements, only the small proof checking may introduce duplication by combination. However, this duplication is controlled by some other techniques.

Some low-level heuristics that a human might use are applied in our system. Allowing restriction on the number of literals in a clause and assignment of high weights to variables prefers terms that contain a small number of variables. The priority function which is defined to be related to the size of the clause prefers terms that are small. The success of the system might suggest that this system provides an approximation of human theorem proving activity. The human-oriented nature of the proofs of the temporal logic theorems obtained by CLIN provides additional evidence.

The user is allowed to convey his or her intuition or knowledge to guide the prover's search via the technique of predicate replacement. We have shown the effectiveness of this technique by proving several hard theorems. Among them the intermediate value theorem and the theorem of the greatest common divisor are described in detail.

We have described the top-level supervisor which serves as a friendly interface between CLIN and the user. This interface accepts the user's problems and directs the system to find proofs without intervention from the user. It frees the user from the necessity of understanding our system or the merits of different strategies in order to use it. It also eliminates the need for tedious bookkeeping by a human in trying a sequence of strategies.

We have shown that our system is competitive with other major theorem proving systems. Our system works especially better on propositional problems and near-propositional problems. This shows the significance of removing duplication of instances of clauses, as we claimed in our thesis in the first chapter.

We have discussed some applications of our system. We have shown how constraint satisfaction problems of AI and logic puzzles are solved by the system. Our system can not only be a theorem prover but also be a problem solver. We have also presented a new paradigm of proving propositional temporal logic theorems. The proofs obtained are very human-oriented and understandable. Potential applications to database design and axioms debugging have also been briefly addressed.

10.2 Extensions

10.2.1 Abstraction

The idea of abstraction [Pla81][Pla90] is to map a set S of clauses onto an "abstract" set T of clauses, find proofs from T, and use these proofs as a guide to the search for a proof from S.

It is possible to use abstraction with our system in the following way. If a clause C corresponds to a clause in an abstract proof, then we can reduce its priority value by some amount, thereby in effect preferring this clause. This will guide the search so as to prefer clauses that correspond to clauses in the abstract proofs. Another idea is to prefer clauses that are closely related to such clauses C. We can measure how closely related two clauses are by their distance in the connection graph, analogous to the support distances already computed by the system. This permits clauses to be used in the proof that do not correspond to clauses in the abstract proof, but which are closely related to them.

10.2.2 Specialized Decision Procedures

Some equality method would be useful on many problems. For example, in temporal logic it is desirable to replace $pred(suc(\mathrm{T}))$ by T since they are equal. Therefore, it would be nice to have facilities, such as term-rewriting built in our system. More sophisticated techniques, like associative-commutative unification [Sti81], are sometimes needed. In general, CLIN is well suited to the use of specialized decision procedures, since it generates a set of ground clauses for testing by the propositional calculus prover. Many specialized decision procedures are especially suited to ground clauses. One way to build in a specialized decision procedure, then, is just to add it to the propositional calculus prover. For example, equality could be built in by adding a congruence closure test [NO80] to the propositional calculus prover. However, this is not complete unless the right instances of the input clauses are generated. Such instances will not necessarily be generated unless the equality axioms are included in the input. Including such axioms largely negates the benefits of a specialized decision procedure.

Another possibility is to use a kind of "theory hyper-linking," analogous to the theory resolution of Stickel [Sti85], to generate the needed instances and guarantee completeness. The idea of theory hyper-linking is similar to the idea of theory links mentioned by Murray and Rosenthal [MR86]. One area where a specialized decision procedure might help is temporal logic, where a decision procedure for discrete linearly ordered sets could be applied.

10.2.3 Semantics

In general, it would be nice to use semantics in a proving system. Since CLIN is not subgoal-oriented, as the modified problem reduction format is [Pla88], it is more difficult to use semantics. There is one way to use semantics, however. If the negation of the theorem is of the form $\mathrm{A}[X_1, \ldots, X_n]$ for free variables $X_1, \ldots, X_n$, and the axioms are H, then we are trying to show that $(\mathrm{H} \wedge \mathrm{A}[X_1, \ldots, X_n])$ is unsatisfiable. It could be that from semantics we may suspect that $(\mathrm{H} \wedge \mathrm{A}[t_1, \ldots, t_n])$ is unsatisfiable, for certain terms t. For example, it could be that $\mathrm{A}[t_1, \ldots, t_n]$ is false in a model of H. If so, we can instantiate the X_i to t_i and search for a proof. For example, in the intermediate value theorem, we are searching for an X such that f(X) is zero. We may know from examples that X may be the least upper bound l. Then we can attempt to show that f(l) is zero, which is easier. The same idea can be used to instantiate clauses in H by using other models.

Another way to use semantics is to modify forward and backward support to support relative to some arbitrary interpretation; forward support implicitly uses the positive interpretation and backward support uses the negative interpretation. However, the problem with this idea is that it is difficult to tell if an arbitrary non-ground clause is true in a non-trivial interpretation.

10.2.4 Selective Hyper-Linking

It is possible to extend the basic hyper-linking strategy to be more careful about which hyper-links to perform. To illustrate, suppose we are using forward support. Initially, the positive clauses have forward distance 0 and the other clauses have forward distances in the interval $[1, \infty]$. The first round of hyper-linking will generate some instances of the non-positive clauses; these instances will have forward distance 1. The forward distances of the non-positive input clauses will then be in the interval $[2, \infty]$, and so on, since the instances of distance 1 will have already been generated. In general, each clause has an interval giving its minimum and maximum forward distance. If the same instance appears twice with intervals $[a_1, b_1]$ and $[a_2, b_2]$, then the interval should be changed to $[\min(a_1, a_2), \max(b_1, b_2)]$. Finally, in a hyper-link, we can require that the distances of the nucleus and electrons be close. If C is a nucleus and L is a positive electron from clause D, then there must exist integers i and i+1 such that i is in the interval for D and i+1 is in the interval for C. A similar idea can be applied to all support sets, forward, backward, and user support.

10.2.5 Others

We have presented our system in the context of the clause form. However, the ideas seem to be more general than this, and should apply to non-clausal proof systems which use explicit quantifiers. The idea is to systematically search for parts of formulas that can be made identical by suitable instantiations of variables.

Of course, it might be useful to extend CLIN to sorted logics, mathematical induction, and higher order logic. Mathematical induction might be difficult because the system does not explicitly generate lemmas.

Appendix A

Test Results

A.1 Test Problems

We have run CLIN on a set of test problems with various options, and list some results in this appendix. Problems 1-93 are from Stickel's paper [Sti86]. The problem "example" is a theorem presented by Pellitier and Rudnicki in AAR Newsletter No. 6, 1986. The problem "exx5" is a verification condition from Hoare's FIND program [Ble77]. The problem "exx7" is a situation calculus theorem developed by David Plaisted [Pla82]. "latinsq" is the latin square problem from [Rob63]. "liar" is the truthtellers and liars problem [LO85]. "salt" is the salt and mustard problem from Lewis Carroll [LO85]. The problem "schubert" is Schubert's steamroller [Wal84]. The stack problem is a proof of a trivial property of push down stores. Problems 104-111 are pigeonhole problems from Pelletier's paper [Pel86]. Problems 112-115 are implicational calculus theorems. The problem "freuder" is a constraint satisfaction problem taken from Freuder's paper [Fre78]. Problems 117-120 are logic puzzles taken from [SS86][WOLB84].

All the times are obtained by running ALS-Prolog (Version 0.60) of Applied Logic System, Inc. [ALS88] on a SUN3/60M workstation with 12 MB memory. We use user support for problems "wos20" and "wos31", and replace rules for problems "ls118" and "ls121". An entry with "***" means we had memory allocation problem; thrashing occurred due to inadequate main memory of the workstation. An entry with "—" indicates we stopped the run on the problem in the entry after the time spent is bigger, sometimes many times bigger, than those of the other entries for the same problem. An entry with "+++" means we didn't try the problem in order to save computer time. In the computation of average times, we only consider the problems of which all entries are solved. One point we should mention here. When small proof checking is done slightly differently, our system can prove wos15 and wos19 very fast and would improve the average times significantly.

A.2 Tables

There are five tables in order.

1. Table A.1. Shows the performance of CLIN without and with sliding priority.
2. Table A.2. Shows the effect of small proof checking. The third column are the times without small proof checking. The fourth column are the times with simple small proof checking. The fifth column are the times with small proof checking and small proof size bound 0. The last column are the times with small proof checking and small proof size bound ∞.
3. Table A.3. Shows the performance of CLIN with simple small proof checking. Four set of options are compared:
   - simple small proof checking and input clauses as nucleus
   - simple small proof checking and all clauses as nucleus
   - simple small proof checking, input clauses as nucleus, and deleting non-forward supported instances
   - simple small proof checking, all clauses as nucleus, and deleting non-forward supported instances

   Columns 3 through 6 are the times for these four set of options respectively.
4. Table A.4. Shows the performance of CLIN with small proof checking and small proof size bound ∞. Four set of options are compared:
   - small proof checking, small proof size bound ∞, and input clauses as nucleus
   - small proof checking, small proof size bound ∞, and all clauses as nucleus
   - small proof checking, small proof size bound ∞, input clauses as nucleus, and deleting non-forward supported instances.
   - small proof checking, small proof size bound ∞, all clauses as nucleus, and deleting non-forward supported instances.

   Columns 3 through 6 are the times for these four set of options respectively.
5. Table A.5. Shows the effectiveness of the top-level supervisor. We used batch mode and put all 120 problems in a batch. The least time bound was set 50, the greatest time bound was set 800, and the time limit multiple was set 4. An entry with "—" in this table indicates that the supervisor fails to get a proof for the problem in the entry within the time limit 800 seconds. Notice that we check time overflow during

hyper-linking and after a round of hyper-linking, unit simplification, propositional unsatisfiability test, and small proof checking. Thus the total time spent on a problem may exceed 800 seconds.

Table A.1: CLIN without and with sliding priority

no	problem	without priority (sec)	with priority (sec)	no	problem	without priority (sec)	with priority (sec)
1	burstall	5.200	5.850	2	shortburst	1.033	1.300
3	prim	9.700	9.150	4	haspartst1	6.333	6.883
5	haspartst2	18.700	49.867	6	ances2	0.133	0.250
7	num1	0.717	0.850	8	group1	0.750	0.967
9	group2	3.883	4.133	10	ew1	0.217	0.300
11	ew2	0.183	0.233	12	ew3	0.150	0.317
13	rob1	0.117	0.250	14	rob2	3.833	4.367
15	dm	0.533	0.683	16	qw	1.383	1.667
17	mqw	4.950	4.150	18	dbabhp	12.200	13.633
19	apabhp	208.166	748.267	20	ex4t1	132.817	4223.300
21	ex4t2	133.450	3725.716	22	ex5	289.917	761.233
23	ex6t1	43.750	562.333	24	ex6t2	43.917	562.967
25	wos1	29.200	30.883	26	wos2	21.417	27.000
27	wos3	1.983	2.600	28	wos4	39.266	36.367
29	wos5	4.500	5.400	30	wos6	49.600	39.083
31	wos7	5.783	6.550	32	wos8	2.750	3.550
33	wos9	5.550	6.367	34	wos10	884.650	46.200
35	wos11	65.816	59.050	36	wos12	2.117	2.833
37	wos13	3.317	4.083	38	wos14	3.967	4.883
39	wos15	***	3509.150	40	wos16	8.883	10.250
41	wos17	98.633	73.017	42	wos18	2.366	2.967
43	wos19	***	6328.533	44	wos20	***	45003.100
45	wos21	***	5519.967	46	wos22	111.016	73.383
47	wos23	3.767	4.833	48	wos24	4.817	6.250
49	wos25	5.200	6.617	50	wos26	30.766	32.100
51	wos27	65.066	45.133	52	wos28	***	1859.683
53	wos29	12.400	13.500	54	wos30	4.767	6.417
55	wos31	***	54992.967	56	wos32	2.517	3.350
57	wos33	8.783	9.550	58	ls5	0.483	0.600
59	ls17	11.800	12.617	60	ls23	0.867	1.017

Table A.1: CLIN without and with sliding priority (continued)

no	problem	without priority (sec)	with priority (sec)	no	problem	without priority (sec)	with priority (sec)
61	ls26	1.117	1.350	62	ls28	3.950	4.667
63	ls29	5.150	5.517	64	ls35	4.283	4.517
65	ls36	66.700	57.000	66	ls37	***	13311.517
67	ls41	0.950	1.250	68	ls55	1.300	1.583
69	ls65	39.667	43.050	70	ls68	1.300	1.633
71	ls75	10.317	11.117	72	ls76t1	1.450	1.883
73	ls87	190.866	69.400	74	ls100	0.833	1.067
75	ls103	43.683	174.350	76	ls105	2.033	2.500
77	ls106	2.067	2.467	78	ls108	114.266	487.233
79	ls111	2.017	2.383	80	ls112	318.883	1248.167
81	ls115	6.716	7.183	82	ls116	24.583	26.367
83	ls118	17.883	18.550	84	ls121	4.800	5.383
85	chang_lee_1	0.700	0.900	86	chang_lee_2	3.883	4.167
87	chang_lee_3	1.250	1.417	88	chang_lee_4	0.733	0.850
89	chang_lee_5	0.950	1.167	90	chang_lee_6	1.117	1.350
91	chang_lee_7	0.733	0.833	92	chang_lee_8	9.750	9.183
93	chang_lee_9	5.183	5.817	94	example	20.717	50.917
95	expq	0.067	0.150	96	exx	0.133	0.217
97	exx5	5.667	5.883	98	exx7	5.033	5.667
99	latinsq	86.450	255.133	100	liar	7.517	8.417
101	salt	28.967	31.633	102	schubert	130.583	135.467
103	stack	0.383	0.433	104	ph2	0.117	0.133
105	ph3	0.183	0.300	106	ph4	0.550	0.917
107	ph5	1.800	2.617	108	ph6	7.350	8.750
109	ph7	39.217	42.050	110	ph8	271.050	277.050
111	ph9	2290.350	2314.867	112	i1	***	27.817
113	s1	5.416	23.067	114	ip1	***	131.783
115	p1	***	1928.016	116	freuder	4.017	4.517
117	friends	67.850	70.117	118	fruit	6.750	7.467
119	jobs	639.883	1540.100	120	zebra	1370.966	2145.567
					average	75.401	185.462

Table A.2: CLIN without and with small proof checking

no	problem	without spc (sec)	simple spc (sec)	spc 0 (sec)	spc ∞ (sec)
1	burstall	9.067	8.267	8.200	5.200
2	shortburst	2.300	2.433	2.417	1.033
3	prim	9.183	8.467	8.500	9.700
4	haspartst1	3.033	4.533	4.567	6.333
5	haspartst2	9.967	14.200	14.050	18.700
6	ances2	0.150	0.133	0.133	0.133
7	num1	1.800	1.900	1.900	0.717
8	group1	2.017	0.817	0.800	0.750
9	group2	5.383	5.383	5.367	3.883
10	ew1	0.217	0.200	0.200	0.217
11	ew2	0.150	0.167	0.183	0.183
12	ew3	0.150	0.167	0.183	0.150
13	rob1	0.150	0.133	0.150	0.117
14	rob2	58.117	8.900	8.883	3.833
15	dm	1.583	0.533	0.517	0.533
16	qw	1.883	1.417	1.383	1.383
17	mqw	4.883	4.983	4.967	4.950
18	dbabhp	14.433	13.800	13.850	12.200
19	apabhp	545.633	174.016	175.383	208.166
20	ex4t1	132.833	132.617	132.683	132.817
21	ex4t2	133.233	133.383	133.250	133.450
22	ex5	222.217	227.100	227.233	289.917
23	ex6t1	255.033	21.983	21.950	43.750
24	ex6t2	231.183	22.117	22.067	43.917
25	wos1	250.733	9.183	9.233	29.200
26	wos2	7.283	5.817	5.850	21.417
27	wos3	1.633	2.000	1.967	1.983
28	wos4	20.183	23.633	23.617	36.367
29	wos5	54.233	8.000	8.017	4.500
30	wos6	17.733	19.150	19.167	39.083
31	wos7	13.550	9.800	9.800	5.783
32	wos8	6.550	4.900	4.933	2.750
33	wos9	18.700	10.717	10.750	5.550
34	wos10	75.917	30.650	30.683	46.200
35	wos11	41.700	31.650	31.717	59.050
36	wos12	1.567	2.050	2.100	2.117
37	wos13	15.733	17.133	17.150	3.317
38	wos14	12.600	5.450	5.483	3.967
39	wos15	9465.017	8887.483	8881.467	3509.150
40	wos16	37.300	34.767	34.883	8.883

Table A.2: CLIN without and with small proof checking (continued)

no	problem	without spc (sec)	simple spc (sec)	spc 0 (sec)	spc ∞ (sec)
41	wos17	265.267	38.833	38.750	73.017
42	wos18	3.833	3.000	2.967	2.967
43	wos19	—	14637.567	14697.017	6328.533
44	wos20	+++	+++	+++	45003.100
45	wos21	2005.683	2013.533	5218.733	5519.967
46	wos22	2006.633	52.650	52.550	73.383
47	wos23	7.017	3.800	3.800	3.767
48	wos24	20.833	17.867	17.783	4.817
49	wos25	49.967	32.650	32.767	5.200
50	wos26	51.983	55.933	55.950	32.100
51	wos27	671.683	25.067	25.000	45.133
52	wos28	2393.767	851.350	851.267	1859.683
53	wos29	77.367	31.500	31.550	13.500
54	wos30	3.550	4.733	4.717	4.767
55	wos31	+++	+++	+++	54992.967
56	wos32	7.900	7.567	7.633	2.517
57	wos33	32.233	27.500	27.433	9.550
58	ls5	0.467	0.450	0.500	0.483
59	ls17	10.000	9.167	9.133	11.800
60	ls23	27.150	4.833	4.800	0.867
61	ls26	3.417	2.633	2.617	1.117
62	ls28	106.517	101.707	101.717	3.950
63	ls29	50.333	53.100	53.083	5.150
64	ls35	13.850	4.933	4.850	4.283
65	ls36	58.650	34.333	34.433	57.000
66	ls37	—	10169.200	12831.117	13311.517
67	ls41	1.883	0.917	0.950	0.950
68	ls55	8.717	5.267	5.250	1.300
69	ls65	72.300	16.150	16.267	39.667
70	ls68	1.217	1.300	1.317	1.300
71	ls75	99.700	16.683	16.733	10.317
72	ls76t1	2.400	1.417	1.467	1.450
73	ls87	43.483	39.417	39.383	69.400
74	ls100	1.617	1.850	1.867	0.833
75	ls103	28.417	33.717	33.700	43.683
76	ls105	4.933	7.100	7.167	2.033
77	ls106	5.083	7.200	7.250	2.067
78	ls108	100.400	106.233	106.150	114.266
79	ls111	4.950	7.233	7.367	2.017
80	ls112	290.750	300.683	300.666	318.883

Table A.2: CLIN without and with small proof checking (continued)

no	problem	without spc (sec)	simple spc (sec)	spc 0 (sec)	spc ∞ (sec)
81	ls115	689.300	698.450	689.267	7.183
82	ls116	5.800	10.800	10.733	24.583
83	ls118	13.200	14.716	15.250	17.883
84	ls121	4.066	4.483	4.500	4.800
85	chang_lee_1	1.617	0.750	0.717	0.700
86	chang_lee_2	5.250	5.283	5.283	3.883
87	chang_lee_3	1.733	1.733	1.667	1.250
88	chang_lee_4	1.700	1.733	1.717	0.733
89	chang_lee_5	1.100	0.917	0.933	0.950
90	chang_lee_6	3.317	2.617	2.533	1.117
91	chang_lee_7	1.800	1.767	1.767	0.733
92	chang_lee_8	9.100	8.500	8.333	9.750
93	chang_lee_9	3.850	4.067	3.967	5.183
94	example	17.300	18.583	18.433	20.717
95	expq	0.100	0.083	0.100	0.067
96	exx	0.150	0.167	0.150	0.133
97	exx5	2.283	3.283	3.183	5.667
98	exx7	3.367	4.833	4.767	5.033
99	latinsq	56.133	60.550	60.133	86.450
100	liar	5.283	6.150	6.050	7.517
101	salt	28.617	28.933	28.633	28.967
102	schubert	187.100	225.300	222.533	135.467
103	stack	0.450	0.383	0.367	0.383
104	ph2	0.117	0.117	0.117	0.117
105	ph3	0.183	0.183	0.183	0.183
106	ph4	0.550	0.550	0.550	0.550
107	ph5	1.800	1.800	1.800	1.800
108	ph6	7.350	7.350	7.350	7.350
109	ph7	39.217	39.217	39.217	39.217
110	ph8	271.050	271.050	271.050	271.050
111	ph9	2290.350	2290.350	2290.350	2290.350
112	i1	18.617	20.067	20.517	27.817
113	s1	17.567	17.400	17.817	23.067
114	ip1	391.200	356.150	357.167	131.783
115	p1	1793.733	1865.883	1859.383	1928.016
116	freuder	3.050	3.567	3.650	4.017
117	friends	22.967	27.550	27.667	67.850
118	fruit	4.883	5.600	5.617	6.750
119	jobs	334.900	353.167	353.983	639.883
120	zebra	869.533	905.917	908.033	1370.966
	average	235.591	181.744	209.212	174.348

Table A.3: CLIN with simple small proof checking

no	problem	input as nucleus (sec)	all as nucleus (sec)	input + nf del. (sec)	all + nf del. (sec)
1	burstall	8.267	8.667	8.833	8.650
2	shortburst	2.433	2.500	2.517	2.567
3	prim	8.467	8.767	8.367	8.300
4	haspartst1	4.533	4.783	12.200	12.067
5	haspartst2	14.200	14.833	117.583	116.717
6	ances2	0.133	0.183	0.150	0.167
7	num1	1.900	1.917	2.050	2.050
8	group1	0.817	0.950	0.833	0.833
9	group2	5.383	5.700	5.533	5.517
10	ew1	0.200	0.200	0.217	0.217
11	ew2	0.167	0.250	0.183	0.167
12	ew3	0.167	0.167	0.167	0.183
13	rob1	0.133	0.133	0.133	0.167
14	rob2	8.900	9.383	7.783	7.800
15	dm	0.533	0.550	0.550	0.533
16	qw	1.417	1.717	3.600	3.567
17	mqw	4.983	5.017	17.467	17.233
18	dbabhp	13.800	14.617	15.183	15.183
19	apabhp	174.016	174.116	173.983	171.000
20	ex4t1	132.617	140.917	88.867	88.167
21	ex4t2	133.383	136.833	88.333	87.850
22	ex5	227.100	236.900	213.900	213.217
23	ex6t1	21.983	23.150	21.067	21.283
24	ex6t2	22.117	22.733	21.200	21.383
25	wos1	9.183	9.500	9.400	9.433
26	wos2	5.817	5.967	6.050	6.017
27	wos3	2.000	2.033	1.983	1.967
28	wos4	23.633	24.117	25.967	26.000
29	wos5	8.000	8.333	7.967	7.933
30	wos6	19.150	19.700	19.317	19.150
31	wos7	9.800	10.333	10.167	10.083
32	wos8	4.900	5.000	5.000	4.950
33	wos9	10.717	11.333	11.033	11.167
34	wos10	30.650	33.300	31.200	31.283
35	wos11	31.650	34.117	31.100	30.917
36	wos12	2.050	2.117	2.150	2.117
37	wos13	17.133	18.067	27.783	27.700
38	wos14	5.450	5.767	5.600	5.533
39	wos15	8887.483	8895.133	8888.683	8846.667
40	wos16	34.767	35.317	28.100	27.767

Table A.3: CLIN with simple small proof checking (continued)

no	problem	input as nucleus (sec)	all as nucleus (sec)	input + nf del. (sec)	all + nf del. (sec)
41	wos17	38.833	39.317	677.250	663.167
42	wos18	3.000	3.050	3.000	3.000
43	wos19	14637.567	14697.750	14679.117	14697.683
44	wos20	+++	+++	+++	+++
45	wos21	2013.533	8986.150	2005.583	8993.617
46	wos22	52.650	52.783	52.867	52.650
47	wos23	3.800	3.800	3.850	3.833
48	wos24	17.867	17.767	18.350	18.300
49	wos25	32.650	32.567	26.817	26.950
50	wos26	55.933	55.967	56.350	55.950
51	wos27	25.067	25.050	25.267	25.050
52	wos28	851.350	851.100	857.917	851.517
53	wos29	31.500	31.400	31.617	31.617
54	wos30	4.733	4.717	4.783	4.833
55	wos31	+++	+++	+++	+++
56	wos32	7.567	7.650	7.150	7.217
57	wos33	27.500	27.500	24.317	24.283
58	ls5	0.450	0.517	0.583	0.600
59	ls17	9.167	9.050	18.100	17.967
60	ls23	4.833	4.850	4.933	4.900
61	ls26	2.633	2.650	2.800	2.783
62	ls28	101.707	101.867	74.850	75.383
63	ls29	53.100	52.983	40.333	40.033
64	ls35	4.933	4.950	5.050	5.050
65	ls36	34.333	34.483	36.517	36.233
66	ls37	10169.200	+++	+++	+++
67	ls41	0.917	0.917	0.967	0.900
68	ls55	5.267	5.283	4.983	4.983
69	ls65	16.150	16.367	14.150	14.117
70	ls68	1.300	1.283	1.333	1.317
71	ls75	16.683	16.717	13.183	13.150
72	ls76t1	1.417	1.450	1.500	1.483
73	ls87	39.417	39.517	39.750	39.467
74	ls100	1.850	1.867	2.083	2.033
75	ls103	33.717	33.683	360.667	358.400
76	ls105	7.100	7.167	8.117	8.100
77	ls106	7.200	7.150	8.183	8.183
78	ls108	106.233	105.666	—	—
79	ls111	7.233	7.317	8.250	8.100
80	ls112	300.683	301.400	—	—

Table A.3: CLIN with simple small proof checking (continued)

no	problem	input as nucleus (sec)	all as nucleus (sec)	input + nf del. (sec)	all + nf del. (sec)
81	ls115	698.450	703.467	356.417	342.483
82	ls116	10.800	10.783	10.983	10.667
83	ls118	14.716	14.516	23.750	23.933
84	ls121	4.483	4.416	4.550	4.666
85	chang_lee_1	0.750	0.733	0.733	0.700
86	chang_lee_2	5.283	5.333	5.567	5.433
87	chang_lee_3	1.733	1.717	1.817	1.783
88	chang_lee_4	1.733	1.733	1.800	1.783
89	chang_lee_5	0.917	0.933	0.950	0.933
90	chang_lee_6	2.617	2.583	2.750	2.617
91	chang_lee_7	1.767	1.750	1.933	1.900
92	chang_lee_8	8.500	8.550	8.333	8.217
93	chang_lee_9	4.067	4.033	6.533	6.433
94	example	18.583	18.700	22.650	22.367
95	expq	0.083	0.100	0.100	0.100
96	exx	0.167	0.183	0.183	0.183
97	exx5	3.283	3.217	3.667	3.600
98	exx7	4.833	4.817	5.567	5.467
99	latinsq	60.550	60.817	78.283	77.383
100	liar	6.150	6.150	6.583	6.433
101	salt	28.933	28.917	21.167	20.883
102	schubert	225.300	225.350	226.650	222.700
103	stack	0.383	0.367	0.367	0.383
104	ph2	0.117	0.117	0.117	0.117
105	ph3	0.183	0.183	0.183	0.183
106	ph4	0.550	0.550	0.550	0.550
107	ph5	1.800	1.800	1.800	1.800
108	ph6	7.350	7.350	7.350	7.350
109	ph7	39.217	39.217	39.217	39.217
110	ph8	271.050	271.050	271.050	271.050
111	ph9	2290.350	2290.350	2290.350	2290.350
112	i1	20.067	20.050	20.150	19.850
113	s1	17.400	17.450	17.533	17.283
114	ip1	356.150	356.517	356.633	353.400
115	p1	1865.883	1868.250	1867.717	1856.400
116	freuder	3.567	3.550	4.067	4.050
117	friends	27.550	27.533	34.250	34.000
118	fruit	5.600	5.600	6.233	6.267
119	jobs	353.167	354.283	391.783	390.717
120	zebra	905.917	908.467	1485.517	1475.767
	average	307.069	368.723	318.509	378.397

Table A.4: CLIN with small proof size ∞

no	problem	input as nucleus (sec)	all as nucleus (sec)	input + nf del. (sec)	all + nf del. (sec)
1	burstall	5.200	5.217	5.433	5.567
2	shortburst	1.033	1.050	1.067	1.083
3	prim	9.700	9.500	9.317	9.450
4	haspartst1	6.333	6.317	10.183	10.467
5	haspartst2	18.700	19.550	123.700	131.633
6	ances2	0.133	0.133	0.133	0.133
7	num1	0.717	0.733	0.783	0.717
8	group1	0.750	0.783	0.817	0.817
9	group2	3.883	3.900	3.883	3.933
10	ew1	0.217	0.200	0.200	0.200
11	ew2	0.183	0.150	0.167	0.167
12	ew3	0.150	0.167	0.183	0.167
13	rob1	0.117	0.150	0.150	0.117
14	rob2	3.833	3.783	3.900	3.867
15	dm	0.533	0.517	0.567	0.550
16	qw	1.383	1.383	1.650	1.667
17	mqw	4.950	4.833	17.100	16.917
18	dbabhp	12.200	12.233	14.700	14.400
19	apabhp	208.166	203.983	210.283	203.933
20	ex4t1	132.817	131.117	88.050	86.867
21	ex4t2	133.450	131.583	87.633	87.383
22	ex5	289.917	286.017	263.883	263.600
23	ex6t1	43.750	42.717	43.150	42.017
24	ex6t2	43.917	43.400	43.517	42.750
25	wos1	29.200	29.250	29.400	29.300
26	wos2	21.417	21.500	21.300	21.450
27	wos3	1.983	1.983	1.983	2.000
28	wos4	36.367	35.933	38.883	38.717
29	wos5	4.500	4.433	4.517	4.667
30	wos6	39.083	39.150	39.183	39.217
31	wos7	5.783	5.783	5.750	5.800
32	wos8	2.750	2.767	2.750	2.783
33	wos9	5.550	5.467	5.617	5.633
34	wos10	46.200	46.067	46.300	46.100
35	wos11	59.050	58.600	57.983	57.483
36	wos12	2.117	2.133	2.117	2.133
37	wos13	3.317	3.283	3.400	3.333
38	wos14	3.967	3.917	3.967	3.900
39	wos15	3509.150	3516.733	3524.683	3517.600
40	wos16	8.883	8.583	8.900	8.950

Table A.4: CLIN with small proof size ∞ (continued)

no	problem	input as nucleus (sec)	all as nucleus (sec)	input + nf del. (sec)	all + nf del. (sec)
41	wos17	73.017	64.233	772.950	699.683
42	wos18	2.967	2.917	2.983	3.000
43	wos19	6328.533	6229.683	6350.367	6227.617
44	wos20	45003.100	+++	+++	+++
45	wos21	5519.967	2054.900	2005.217	5195.467
46	wos22	73.383	72.733	73.417	72.567
47	wos23	3.767	3.833	3.800	3.800
48	wos24	4.817	4.883	4.833	4.917
49	wos25	5.200	5.150	5.233	5.300
50	wos26	32.100	31.883	32.217	31.933
51	wos27	45.133	45.083	45.150	44.933
52	wos28	1859.683	1780.533	1879.000	1763.250
53	wos29	13.500	13.750	13.717	13.750
54	wos30	4.767	4.717	4.733	4.783
55	wos31	54992.967	+++	+++	+++
56	wos32	2.517	2.483	2.567	2.517
57	wos33	9.550	9.667	9.567	9.567
58	ls5	0.483	0.500	0.567	0.567
59	ls17	11.800	11.800	11.000	10.833
60	ls23	0.867	0.917	0.833	0.850
61	ls26	1.117	1.133	1.150	1.133
62	ls28	3.950	3.883	3.950	4.033
63	ls29	5.150	5.117	5.183	5.167
64	ls35	4.283	4.200	4.250	4.350
65	ls36	57.000	56.950	58.650	58.767
66	ls37	13311.517	+++	+++	+++
67	ls41	0.950	0.950	0.933	0.950
68	ls55	1.300	1.317	1.317	1.300
69	ls65	39.667	39.883	37.167	36.467
70	ls68	1.300	1.367	1.317	1.300
71	ls75	10.317	10.350	10.500	10.383
72	ls76t1	1.450	1.583	1.450	1.467
73	ls87	69.400	69.200	69.217	69.000
74	ls100	0.833	0.817	0.833	0.833
75	ls103	43.683	44.900	370.783	372.150
76	ls105	2.033	2.133	2.050	2.033
77	ls106	2.067	2.300	2.000	2.133
78	ls108	114.266	126.716	—	—
79	ls111	2.017	2.100	2.017	2.017
80	ls112	318.883	338.533	—	—

Table A.4: CLIN with small proof size ∞ (continued)

no	problem	input as nucleus (sec)	all as nucleus (sec)	input + nf del. (sec)	all + nf del. (sec)
81	ls115	7.183	7.350	7.217	7.250
82	ls116	24.583	22.700	24.633	22.583
83	ls118	17.883	17.450	28.7	28.016
84	ls121	4.800	4.750	4.900	5.033
85	chang_lee_1	0.700	0.717	0.733	0.717
86	chang_lee_2	3.883	4.000	3.983	3.867
87	chang_lee_3	1.250	1.250	1.300	1.267
88	chang_lee_4	0.733	0.733	0.733	0.700
89	chang_lee_5	0.950	0.933	0.950	0.933
90	chang_lee_6	1.117	1.167	1.133	1.083
91	chang_lee_7	0.733	0.717	0.717	0.717
92	chang_lee_8	9.750	10.433	9.567	9.150
93	chang_lee_9	5.183	5.283	5.883	5.767
94	example	20.717	21.250	24.667	24.800
95	expq	0.067	0.083	0.083	0.100
96	exx	0.133	0.150	0.167	0.150
97	exx5	5.667	5.617	6.133	6.133
98	exx7	5.033	4.967	5.500	5.417
99	latinsq	86.450	86.900	104.450	102.650
100	liar	7.517	7.450	7.883	7.750
101	salt	28.967	29.317	21.200	20.867
102	schubert	135.467	129.683	134.883	127.267
103	stack	0.383	0.417	0.383	0.383
104	ph2	0.117	0.117	0.117	0.117
105	ph3	0.183	0.183	0.183	0.183
106	ph4	0.550	0.550	0.550	0.550
107	ph5	1.800	1.800	1.800	1.800
108	ph6	7.350	7.350	7.350	7.350
109	ph7	39.217	39.217	39.217	39.217
110	ph8	271.050	271.050	271.050	271.050
111	ph9	2290.350	2290.350	2290.350	2290.350
112	i1	27.817	28.017	27.983	27.783
113	s1	23.067	22.933	23.017	22.950
114	ip1	131.783	134.317	131.900	132.350
115	p1	1928.016	1943.583	1929.717	1911.316
116	freuder	4.017	3.983	4.483	4.483
117	friends	67.850	68.250	79.050	75.383
118	fruit	6.750	7.233	7.567	7.583
119	jobs	639.883	626.767	773.500	778.883
120	zebra	1370.966	1329.533	1969.183	1913.283
	average	227.128	194.968	212.790	236.987

Table A.5: CLIN without and with top-level supervisor

no	problem	without supervisor (sec)	with supervisor (sec)	no	problem	without supervisor (sec)	with supervisor (sec)
1	burstall	5.200	5.353	2	shortburst	1.033	1.233
3	prim	9.150	5.833	4	haspartst1	6.333	5.133
5	haspartst2	18.700	46.550	6	ances2	0.133	0.217
7	num1	0.717	0.850	8	group1	0.750	0.983
9	group2	3.883	4.100	10	ew1	0.217	0.300
11	ew2	0.183	0.267	12	ew3	0.150	0.317
13	rob1	0.117	0.250	14	rob2	3.833	4.117
15	dm	0.533	0.650	16	qw	1.383	1.633
17	mqw	4.150	3.867	18	dbabhp	12.200	8.267
19	apabhp	208.166	755.600	20	ex4t1	132.817	—
21	ex4t2	133.450	—	22	ex5	289.917	99.367
23	ex6t1	43.750	17.800	24	ex6t2	43.917	17.583
25	wos1	29.200	31.067	26	wos2	21.417	21.700
27	wos3	1.983	2.600	28	wos4	36.367	20.550
29	wos5	4.500	5.017	30	wos6	39.083	37.767
31	wos7	5.783	6.333	32	wos8	2.750	3.350
33	wos9	5.550	6.067	34	wos10	46.200	35.967
35	wos11	59.050	41.200	36	wos12	2.117	2.650
37	wos13	3.317	3.917	38	wos14	3.967	4.483
39	wos15	3509.150	—	40	wos16	8.883	9.683
41	wos17	73.017	55.450	42	wos18	2.366	3.000
43	wos19	6328.533	—	44	wos20	45003.100	—
45	wos21	5519.967	—	46	wos22	73.383	—
47	wos23	3.767	4.767	48	wos24	4.817	5.733
49	wos25	5.200	6.367	50	wos26	30.766	32.067
51	wos27	45.133	45.067	52	wos28	1859.683	—
53	wos29	12.400	13.483	54	wos30	4.767	5.950
55	wos31	54992.967	—	56	wos32	2.517	3.200
57	wos33	8.783	9.717	58	ls5	0.483	0.567
59	ls17	11.800	7.867	60	ls23	0.867	1.033

Table A.5: CLIN without and with top-level supervisor (continued)

no	problem	without supervisor (sec)	with supervisor (sec)	no	problem	without supervisor (sec)	with supervisor (sec)
61	ls26	1.117	1.383	62	ls28	3.950	4.417
63	ls29	5.150	5.500	64	ls35	4.283	4.433
65	ls36	57.000	159.133	66	ls37	13311.517	—
67	ls41	0.950	1.167	68	ls55	1.300	1.567
69	ls65	39.667	55.033	70	ls68	1.300	1.650
71	ls75	10.317	10.783	72	ls76t1	1.450	1.833
73	ls87	69.400	55.800	74	ls100	0.833	0.983
75	ls103	43.683	20.317	76	ls105	2.033	2.500
77	ls106	2.067	2.483	78	ls108	114.266	—
79	ls111	2.017	2.417	80	ls112	318.883	—
81	ls115	6.716	7.333	82	ls116	24.583	25.917
83	ls118	17.883	—	84	ls121	4.800	—
85	chang_lee_1	0.700	0.883	86	chang_lee_2	3.883	4.200
87	chang_lee_3	1.250	1.433	88	chang_lee_4	0.733	0.883
89	chang_lee_5	0.950	1.167	90	chang_lee_6	1.117	1.350
91	chang_lee_7	0.733	0.883	92	chang_lee_8	9.183	6.000
93	chang_lee_9	5.183	4.550	94	example	20.717	47.700
95	expq	0.067	0.133	96	exx	0.133	0.217
97	exx5	5.667	5.533	98	exx7	5.033	4.533
99	latinsq	86.450	928.333	100	liar	7.517	7.400
101	salt	28.967	31.350	102	schubert	130.583	424.317
103	stack	0.383	0.450	104	ph2	0.117	0.167
105	ph3	0.183	0.333	106	ph4	0.550	0.917
107	ph5	1.800	2.567	108	ph6	7.350	8.700
109	ph7	39.217	41.633	110	ph8	271.050	274.950
111	ph9	2290.350	2301.417	112	i1	27.817	27.817
113	s1	5.416	22.883	114	ip1	131.783	131.783
115	p1	1928.016	—	116	freuder	4.017	2.850
117	friends	67.850	28.000	118	fruit	6.750	5.800
119	jobs	639.883	675.283	120	zebra	1370.966	—
					average	50.119	65.076

Appendix B

Test Problems

This appendix lists the input files, for CLIN, of the 41 problems used in section 7.3. The meanings of the entries in these files are described in Appendix C.

1. prim.

```
assign(support_list,[f]).
axiom_list(sprfn).
l(1,a).
d(X,X).
false :- p(X), d(X, a).
d(g(X), X) :- not(p(X)).
l(1, g(X)) :- not(p(X)).
l(g(X), X) :- not(p(X)).
p(f(X)) :- l(1,X), l(X, a).
d(f(X), X) :- l(1, X), l(X, a).
d(X,Z) :- d(X,Y), d(Y,Z).
end.
end_of_input.
```

2. haspartst1.

```
assign(support_list,[b,f]).
axiom_list(sprfn).
in(john, boy).
in(X, human) :- in(X, boy).
in(sk1(X,Y,Z,Xm,Xn), Y) :- not(hp(X, t(Xm, Xn), Z)), hp(X,Xm,Y).
hp(X, t(Xm, Xn), Z) :- hp(X, Xm, Y), hp(sk1(X, Y, Z, Xm, Xn), Xn, Z).
```

```
hp(X, 5, fingers) :- in(X, hand).
hp(X, 2, arm) :- in(X, human).
hp(X, 1, hand) :- in(X, arm).
false :- hp(john, t(2, 1), hand).
end.
end_of_input.
```

3. haspartst2.

```
assign(support_list,[f,b]).
axiom_list(sprfn).
in(john, boy).
in(X, human) :- in(X, boy).
in(sk1(X,Y,Z,Xm,Xn), Y) :- not(hp(X, t(Xm, Xn), Z)), hp(X,Xm,Y).
hp(X, t(Xm, Xn), Z) :- hp(X, Xm, Y), hp(sk1(X, Y, Z, Xm, Xn), Xn, Z).
hp(X, 5, fingers) :- in(X, hand).
hp(X, 2, arm) :- in(X, human).
hp(X, 1, hand) :- in(X, arm).
false :- hp(john, t(t(2,1),5), fingers).
end.
end_of_input.
```

4. ances2.

```
assign(support_list,[f,b]).
axiom_list(sprfn).
false :- h, c.
false :- a, b.
h :- not(a), j.
h :- not(k), not(j).
h :- not(j), k.
b :- a.
c :- h.
end.
end_of_input.
```

5. num1.

```
assign(support_list,[f]).
axiom_list(sprfn).
p(a).
```

```
m(a, s(c), s(b)).
m(X, X, s(X)).
m(X,Y,Z) :- m(Y,X,Z).
d(X,Z) :- m(X,Y,Z).
d(X,Y) :- p(X), m(Y,Z,W), d(X,W), not(d(X,Z)).
false :- d(a,b).
end.
end_of_input.
```

6. ew2.

```
assign(support_list,[b]).
axiom_list(sprfn).
p :- not(q).
q :- p.
p :- q, r.
r.
false :- q, p.
end.
end_of_input.
```

7. ew3.

```
assign(support_list,[b]).
axiom_list(sprfn).
p :- s, t.
s :- r.
r :- t.
t :- q, r.
r :- q.
r :- not(q).
q :- r.
q :- p, s.
false :- r, q, p.
end.
end_of_input.
```

8. qw.

```
assign(support_list,[f,b]).
axiom_list(sprfn).
```

```
p(X, f(X)) :- not(p(X, a)).
p(f(X), X) :- not(p(X, a)).
false :- p(X, a), p(X, Y), p(Y, X).
end.
end_of_input.
```

9. mqw.

```
assign(support_list,[b]).
axiom_list(sprfn).
g(f(Y), Y) :- not(g(Y, a)).
g(Y, f(Y)) :- not(g(Y, a)).
g(f(Y), Y) :- g(W, Y).
g(Y, f(Y)) :- g(W, Y).
false :- g(Y2, a), g(Y1, Y2).
end.
end_of_input.
```

10. dbabhp.

```
assign(support_list,[b,f]).
axiom_list(sprfn).
a(s,e,n).
h(U,l(V)) :- h(U,V).
h(U,g(U,W)) :- h(V,W).
false :- i(U,l(V)).
r(U) :- a(U,e,V).
a(U,V,l(W)) :- a(U,V,W).
a(U,V,p(W)) :- a(U,V,W).
b(U,p(g(V,l(W)))) :- a(U,V,W).
c(U) :- q(V,t,U), r(V).
q(V,U,g(U,X)) :- b(V,X), a(V,W,X).
a(W,V,g(U,X)) :- a(W,V,X), not i(W,X).
a(V,U,g(U,X)) :- i(V,X), h(W,X).
i(U,p(W)) :- a(U,V,W), h(V,W).
false :- c(U).
end.
end_of_input.
```

11. apabhp.

```
assign(support_list,[r]).
axiom_list(sprfn).
false :- c(Y).
not i(m(X),d(l,Y)).
a(h, Z, d(g(Z), Y)).
r(X) :- a(X, e, n).
c(Y) :- a(X, t, Y), r(X).
a(m(k(Y)), Z, Y) :- a(h, Z, Y).
a(X, Z, Y) :- a(X, Z, d(l, Y)).
i(m(X), d(g(Z), Y)) :- i(m(X), Y).
a(m(X), Z, Y) :- not i(m(X), Y), a(m(X), Z, d(g(Z), Y)).
a(m(X), Z, Y) :- a(h, Z, Y), i(m(X), Y).
i(m(k(Y)), d(p, Y)) :- a(m(X), Z, Y), a(h, Z, Y).
not r(h).
a(m(s), e, n).
a(X, Z, d(p, Y)) :- a(X, Z, Y).
a(X, Z, d(l, Y)) :- a(X, Z, Y).
a(m(X), Z, d(g(Z), Y)) :- a(m(X), Z, Y).
a(h, Z, Y) :- a(m(X), Z, Y), i(m(X), Y).
false :- a(X, e, Y), a(X, t, Y).
end.
end_of_input.
```

12. ex4t1.

```
assign(support_list,[f,b]).
axiom_list(sprfn).
p(X,f(X,Y)) :- p(X,Y),q(Y,f(X,Y)).
q(f(X, Y), f(X, Y)) :- p(f(X, Y), f(X, Y)), p(X, Y).
p(f(X, Y), f(X, Y)) :- not p(X, Y).
p(X, Y) :- q(f(X, Y), f(X, Y)).
q(f(X, Y), f(X, Y)) :- not q(X, Y).
false :- q(f(X, Y), f(X, Y)), q(X, Y).
q(Y, f(X, Y)) :- p(X, f(X, Y)), p(X, Y).
end.
end_of_input.
```

13. ex4t2.

```
assign(support_list,[f,b]).
axiom_list(sprfn).
```

```
q(Y, f(X, Y)) :- p(X, f(X, Y)), p(X, Y).
q(f(X, Y), f(X, Y)) :- p(f(X,Y),f(X,Y)), p(X, Y).
q(f(X, Y), f(X, Y)) :- not(q(X, Y)).
p(X, f(X, Y)) :- q(Y, f(X, Y)), p(X, Y).
p(X1, X2) :- q(f(X1, X2), f(X1, X2)).
p(f(X, Y), f(X, Y)) :- not(p(X, Y)).
false :- q(f(X,Y), f(X, Y)), q(X, Y).
end.
end_of_input.
```

14. ex5.

```
assign(support_list,[f]).
axiom_list(sprfn).
s(a).
p(X,e,X).
false :- s(i(a)).
false :- p(e,e,0).
p(m(X),m(Y),Z) :- p(X, Y, Z).
p(X, Y, Z) :- p(m(X),m(Y),Z).
p(X,m(Y),m(Z)) :- p(X, Y, Z).
p(X, X, 0) :- not p(X, i(X), e).
false :- s(X), s(m(X)).
false :- s(X), p(X, X, 0).
p(X, Y, Z) :- p(Y, X, Z).
s(X) :- not p(X, X, 0), not s(m(X)).
p(X, X, 0) :- p(Y, Z, X), p(Y, Y, 0).
s(X) :- p(Y, Z, X), s(Y), s(Z).
end.
end_of_input.
```

15. ex6t1.

```
assign(support_list,[f,b]).
axiom_list(sprfn).
q(d, X, X).
q(g(X),X,d).
p(X,Y,f(X,Y)).
false :- p(a,d,d).
q(U,Z,W) :- q(X,Y,U),q(Y,Z,V),q(X,V,W).
q(U,XM,W):- p(X,Y,U),p(X,Z,XM),q(Y,Z,V),p(X,V,W).
```

```
p(X,V,W):-p(X,Y,U),p(X,Z,XM),q(Y,Z,V),q(U,XM,W).
q(X,V,W) :- q(X,Y,U),q(Y,Z,V),q(U,Z,W).
end.
end_of_input.
```

16. ex6t2.

```
assign(support_list,[f,b]).
axiom_list(sprfn).
q(d, X, X).
q(g(X),X,d).
p(X,Y,f(X,Y)).
false :- p(a,d,d).
q(X,V,W) :- q(X,Y,U),q(Y,Z,V),q(U,Z,W).
q(U,Z,W) :- q(X,Y,U),q(Y,Z,V),q(X,V,W).
q(U,XM,W):- p(X,Y,U),p(X,Z,XM),q(Y,Z,V),p(X,V,W).
p(X,V,W):-p(X,Y,U),p(X,Z,XM),q(Y,Z,V),q(U,XM,W).
end.
end_of_input.
```

17. wos4.

```
assign(support_list,[b]).
set(size_by_clause).
set(slidepriority).
axiom_list(sprfn).
false :- q(e).
p(e, X, X).
p(g(X), X, e).
p(X, e, X).
p(X, g(X), e).
p(X, Y, f(X, Y)).
p(X, V, W) :- p(X, Y, U), p(Y, Z, V), p(U, Z, W).
p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
q(V) :- q(U), r(U, V).
p(Y, X, Z) :- p(X, Y, Z), q(X).
p(X, Y, Z) :- p(Y, X, Z), q(X).
q(X) :- not p(X, j(X), h(X)), not p(j(X), X, h(X)).
p(X, j(X), h(X)) :- not q(X), not p(j(X), X, h(X)).
p(j(X), X, h(X)) :- not q(X), not p(X, j(X), h(X)).
q(X) :- p(X, j(X), h(X)), p(j(X), X, h(X)).
```

```
r(X, X).
r(Y, X) :- r(X, Y).
r(X, Z) :- r(X, Y), r(Y, Z).
r(U, V) :- p(X, Y, U), p(X, Y, V).
p(X, Y, V) :- p(X, Y, U), r(U, V).
p(X, V, Y) :- p(X, U, Y), r(U, V).
p(V, X, Y) :- p(U, X, Y), r(U, V).
r(f(X, U), f(X, V)) :- r(U, V).
r(f(U, Y), f(V, Y)) :- r(U, V).
r(g(U), g(V)) :- r(U, V).
end.
end_of_input.
clear(slidepriority).
clear(size_by_clause).
```

18. wos15.

```
assign(support_list,[r]).
set(slidepriority).
assign(time_limit,20000).
axiom_list(sprfn).
false :- o(c).
o(a).
o(b).
p(a, b, c).
o(Z) :- p(X, g(Y), Z), o(X), o(Y).
o(Y) :-  r(X, Y), o(X).
p(e, X, X).
p(X, e, X).
p(g(X), X, e).
p(X, g(X), e).
p(X, Y, f(X,Y)).
p(X, V, W) :- p(X, Y, U), p(Y, Z, V), p(U, Z, W).
p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
p(X, Y, V) :- p(X, Y, U), r(U, V).
p(X, V, Y) :- p(X, U, Y), r(U, V).
p(V, X, Y) :- p(U, X, Y), r(U, V).
r(X, X).
r(Y, X) :- r(X, Y).
r(g(U), g(V)) :- r(U, V).
```

```
r(f(X, U), f(X, V)) :- r(U, V).
r(f(U, Y), f(V, Y)) :- r(U, V).
r(X, Z) :- r(X, Y), r(Y, Z).
r(U, V) :- p(X, Y, U), p(X, Y, V).
end.
end_of_input.
clear(slidepriority).
assign(time_limit,2000).
```

19. wos19.

```
assign(support_list,[r]).
set(slidepriority).
assign(size_coef,1).
assign(depth_coef,2).
assign(relevance_coef,2).
assign(literal_coef,0).
assign(time_limit,20000).
axiom_list(sprfn).
o(a).
o(b).
p(b, g(a), c).
p(a, c, d).
false :- o(d).
p(e, X, X).
p(g(X), X, e).
p(X, Y, f(X,Y)).
p(X, V, W) :- p(X, Y, U), p(Y, Z, V), p(U, Z, W).
p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
r(X, X).
r(Y, X) :- r(X, Y).
r(X, Z) :- r(X, Y), r(Y, Z).
r(U, V) :- p(X, Y, U), p(X, Y, V).
p(X, Y, V) :- p(X, Y, U), r(U, V).
p(X, V, Y) :- p(X, U, Y), r(U, V).
p(V, X, Y) :- p(U, X, Y), r(U, V).
r(g(U), g(V)) :- r(U, V).
r(f(X, U), f(X, V)) :- r(U, V).
r(f(U, Y), f(V, Y)) :- r(U, V).
p(X, e, X).
```

```
p(X, g(X), e).
o(Y) :-  r(X, Y), o(X).
o(Z) :-  o(X), o(Y), p(X, g(Y), Z).
o(g(X)) :- o(X).
o(e).
r(i(X, U), i(X, V)) :- r(U, V).
r(i(U, X), i(V, X)) :- r(U, V).
o(X) :- not o(Y), not o(i(X, Y)).
o(X) :- not o(Y), not p(X, i(X, Y), Y).
r(U, V) :- p(X, U, Z), p(X, V, Z).
r(U, V) :- p(U, Y, Z), p(V, Y, Z).
r(g(g(X)), X).
end.
end_of_input.
clear(slidepriority).
assign(size_coef,1).
assign(depth_coef,0).
assign(relevance_coef,0).
assign(literal_coef,0).
assign(time_limit,2000).
```

20. wos20.

```
assign(support_list,[ur]).
set(slidepriority).
assign(size_coef,1).
assign(depth_coef,2).
assign(relevance_coef,2).
assign(literal_coef,0).
assign(time_limit,100000).
axiom_list(sprfn).
o(b).
p(b, g(a), c).
p(a, c, d).
p(X, e, X).
p(X, g(X), e).
false :- o(d).
false :- o(a).
o(e).
p(e, X, X).
```

```
p(g(X), X, e).
o(g(X)) :- o(X).
o(X) :- not o(Y), not o(i(X, Y)).
o(X) :- not o(Y), not p(X, i(X, Y), Y).
p(X, Y, f(X,Y)).
p(X, V, W) :- p(X, Y, U), p(Y, Z, V), p(U, Z, W).
p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
p(X, Y, V) :- p(X, Y, U), r(U, V).
p(X, V, Y) :- p(X, U, Y), r(U, V).
p(V, X, Y) :- p(U, X, Y), r(U, V).
r(g(U), g(V)) :- r(U, V).
r(f(X, U), f(X, V)) :- r(U, V).
r(f(U, Y), f(V, Y)) :- r(U, V).
o(Y) :-  r(X, Y), o(X).
o(Z) :-  o(X), o(Y), p(X, g(Y), Z).
r(i(X, U), i(X, V)) :- r(U, V).
r(i(U, X), i(V, X)) :- r(U, V).
r(U, V) :- p(X, U, Z), p(X, V, Z).
r(U, V) :- p(U, Y, Z), p(V, Y, Z).
r(X, X).
r(Y, X) :- r(X, Y).
r(X, Z) :- r(X, Y), r(Y, Z).
r(U, V) :- p(X, Y, U), p(X, Y, V).
r(g(g(X)), X).
end.
set_of_support.
1.
2.
3.
6.
7.
end.
end_of_input.
clear(slidepriority).
assign(time_limit,2000).
assign(size_coef,1).
assign(depth_coef,0).
assign(relevance_coef,0).
assign(literal_coef,0).
```

21. wos22.

```
assign(support_list,[r]).
set(size_by_clause).
set(slidepriority).
axiom_list(sprfn).
p(k, X, k).
p(X, k, k).
p(a, b, c).
p(g(a), g(b), d).
false :- r(c, d).
q(X,k,X).
q(X,g(X),k).
q(X,Y,j(X,Y)).
q(g(X), X, k).
q(k, X, X).
p(X, Y, f(X, Y)).
q(U,Z,W) :- q(X,Y,U), q(Y,Z,V), q(X,V,W).
q(X,V,W) :- q(X,Y,U), q(Y,Z,V), q(U,Z,W).
q(X, Y, Z) :- q(Y, X, Z).
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).
p(X,V,W) :- p(X,Y,U), p(Y,Z,V), p(U,Z,W).
q(U,Xt,W) :- q(Y,Z,V), p(X,Y,U), p(X,Z,Xt), p(X,V,W).
p(X,V,W) :- q(Y,Z,V), p(X,Y,U), p(X,Z,Xt), q(U,Xt,W).
q(U,Xt,W) :- q(Y,Z,V), p(Y,X,U), p(Z,X,Xt), p(V,X,W).
p(V,X,W) :- q(Y,Z,V), p(Y,X,U), p(Z,X,Xt), q(U,Xt,W).
r(U, V) :- q(X, Y, U), q(X, Y, V).
r(U, V) :- p(X, Y, U), p(X, Y, V).
p(X, Y, V) :- r(U, V), p(X, Y, U).
p(X, V, Y) :- r(U, V), p(X, U, Y).
p(V, Y, X) :- r(U, V), p(U, Y, X).
q(X, Y, V) :- r(U, V), q(X, Y, U).
q(X, V, Y) :- r(U, V), q(X, U, Y).
q(V, Y, X) :- r(U, V), q(U, Y, X).
r(f(X, U), f(X, V)) :- r(U, V).
r(j(X, U), j(X, V)) :- r(U, V).
r(g(U), g(V)) :- g(U, V).
r(X, X).
r(X, Y) :- r(Y, X).
r(X, Z) :- r(X, Y), r(Y, Z).
```

```
end.
end_of_input.
clear(slidepriority).
clear(size_by_clause).
```

22. wos26.

```
assign(support_list,[r]).
axiom_list(sprfn).
p(a).
q(b,b,d).
q(c,c,e).
q(X,Y,f(X,Y)).
q(X,h(X,Y),Y):- n(X,Y).
n(X,Z) :- q(X,Y,Z).
n(X,Y) :- p(X),q(Y,Y,Z),n(X,Z).
r(Z,Y):-q(X,Y,U),q(X,Z,U).
q(U,Z,W) :- q(X,Y,U), q(Y,Z,V), q(X,V,W).
q(X,V,W) :- q(X,Y,U), q(Y,Z,V), q(U,Z,W).
n(X,Z) :- n(X,Y),n(Y,Z).
r(U,V) :- q(X,Y,U),q(X,Y,V).
n(X,Z) :- n(X,Y), r(Z,Y).
n(Z,Y) :- n(X,Y), r(X,Z).
p(Y) :- p(X), r(X,Y).
q(X,Y,V) :- q(X,Y,U),r(Y,V).
q(X,V,Y) :- q(X,U,Y),r(Y,V).
q(V,X,Y) :- q(U,X,Y),r(Y,V).
r(X,X).
r(X,Y) :- r(Y,X).
r(X,Z) :- r(X,Y),r(Y,Z).
q(Y,X,Z) :- q(X,Y,Z).
false :- q(a,e,d).
false :-n(X, b), n(X, c).
end.
end_of_input.
```

23. wos27.

```
assign(support_list,[r]).
set(slidepriority).
axiom_list(sprfn).
```

```
q(X, Y, j(X, Y)).
q(X, V, W) :- q(X, Y, U), q(Y, Z, V), q(U, Z, W).
q(U, Z, W) :- q(X, Y, U), q(Y, Z, V), q(X, V, W).
q(X, Y, Z) :- q(Y, X, Z).
p(X, Y, f(X, Y)).
p(X, V, W) :- p(X, Y, U), p(Y, Z, V), p(U, Z, W).
p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
q(U, Xt,W) :- q(Y, Z, V), p(X, Y, U), p(X, Z,Xt), p(X,V,W).
p(X, V, W) :- q(Y, Z, V), p(X, Y, U), p(X, Z,Xt), q(U,Xt,W).
q(U, Xt,W) :- q(Y, Z, V), p(Y, X, U), p(Z, X,Xt), p(V,X,W).
p(V, X, W) :- q(Y, Z, V), p(Y, X, U), p(Z, X,Xt), q(U,Xt,W).
q(X, k, X).
q(X, g(X), k).
r(U, V) :- q(X, Y, U), q(X, Y, V).
r(U, V) :- p(X, Y, U), p(X, Y, V).
p(X, Y, V) :- r(U, V), p(X, Y, U).
p(X, V, Y) :- r(U, V), p(X, U, Y).
p(V, Y, X) :- r(U, V), p(U, Y, X).
q(X, Y, V) :- r(U, V), q(X, Y, U).
q(X, V, Y) :- r(U, V), q(X, U, Y).
q(V, Y, X) :- r(U, V), q(U, Y, X).
r(f(X, U), f(X, V)) :- r(U, V).
r(j(X, U), j(X, V)) :- r(U, V).
r(g(U), g(V)) :- r(U, V).
p(X, k, k).
p(k, X, k).
p(X, h(X, Y), Y) :- r(X, k).
p(a, b, k).
false :- r(a, k).
false :- r(b, k).
end.
end_of_input.
clear(slidepriority).
```

24. wos28.

```
assign(support_list,[r]).
set(slidepriority).
axiom_list(sprfn).
q(X, Y, j(X, Y)).
```

```
q(X, V, W) :- q(X, Y, U), q(Y, Z, V), q(U, Z, W).
q(U, Z, W) :- q(X, Y, U), q(Y, Z, V), q(X, V, W).
q(X, Y, Z) :- q(Y, X, Z).
p(X, Y, f(X, Y)).
p(X, V, W) :- p(X, Y, U), p(Y, Z, V), p(U, Z, W).
p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
q(U, Xt,W) :- q(Y, Z, V), p(X, Y, U), p(X, Z,Xt), p(X,V,W).
p(X, V, W) :- q(Y, Z, V), p(X, Y, U), p(X, Z,Xt), q(U,Xt,W).
q(U, Xt,W) :- q(Y, Z, V), p(Y, X, U), p(Z, X,Xt), p(V,X,W).
p(V, X, W) :- q(Y, Z, V), p(Y, X, U), p(Z, X,Xt), q(U,Xt,W).
r(U, V) :- q(X, Y, U), q(X, Y, V).
r(U, V) :- p(X, Y, U), p(X, Y, V).
p(X, Y, V) :- r(U, V), p(X, Y, U).
p(X, V, Y) :- r(U, V), p(X, U, Y).
p(V, Y, X) :- r(U, V), p(U, Y, X).
q(X, Y, V) :- r(U, V), q(X, Y, U).
q(X, V, Y) :- r(U, V), q(X, U, Y).
q(V, Y, X) :- r(U, V), q(U, Y, X).
r(f(X, U), f(X, V)) :- r(U, V).
r(j(X, U), j(X, V)) :- r(U, V).
r(X, X).
r(X, Z) :- r(X, Y), r(Y, Z).
r(f(U, X), f(V, X)) :- r(U, V).
r(j(U, X), j(V, X)) :- r(U, V).
q(0, X, X).
q(X, 0, X).
r(U, V) :- q(U, X, Y), q(V, X, Y).
r(U, V) :- q(X, U, Y), q(X, V, Y).
q(X, j(X, Y), Y).
q(j(X, Y), Y, X).
q(X, X, 0).
r(j(X, 0), X).
r(j(X, X), 0).
r(f(X, X), X).
r(f(a, b), c).
r(f(b, a), d).
q(X, Y, j(Y, X)).
p(a, c, c).
p(b, d, d).
```

```
p(c, b, c).
p(d, a, d).
p(Y, f(Y, X), f(Y, X)).
p(a, f(b, X), f(C, X)).
p(a, b, f(c, b)).
p(a, f(b, c), c).
p(b, f(a, X), f(d, X)).
p(b, a, f(d, a)).
p(b, f(a, d), d).
p(b, c, f(d, b)).
p(a, d, f(c, a)).
p(f(Y, X), X, f(Y, X)).
p(f(X, a), b, f(X, c)).
p(a, b, f(a, c)).
p(f(c, a), b, c).
p(d, b, f(b, c)).
p(f(X, b), a, f(X, d)).
p(b, a, f(b, d)).
p(f(d, b), a, d).
p(c, a, f(a, d)).
p(0, X, 0).
p(X, 0, 0).
p(a, j(b, a), j(c, a)).
p(a, j(a, b), j(a, c)).
p(b, j(a, b), j(d, b)).
p(b, j(b, a), j(b, d)).
p(j(a, b), b, j(c, b)).
p(j(b, a), b, j(b, c)).
p(j(b, a), a, j(d, a)).
p(j(a, b), a, j(a, d)).
p(X, X, X).
p(a, b, c).
p(b, a, d).
false :- r(c, d).
end.
end_of_input.
clear(slidepriority).
```

25. wos29.

```
assign(support_list,[r]).
axiom_list(sprfn).
q(X, Y, j(X, Y)).
q(X, V, W) :- q(X, Y, U), q(Y, Z, V), q(U, Z, W).
q(U, Z, W) :- q(X, Y, U), q(Y, Z, V), q(X, V, W).
q(X, Y, Z) :- q(Y, X, Z).
p(X, Y, f(X, Y)).
p(X, V, W) :- p(X, Y, U), p(Y, Z, V), p(U, Z, W).
p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
q(U, Xt,W) :- q(Y, Z, V), p(X, Y, U), p(X, Z,Xt), p(X,V,W).
p(X, V, W) :- q(Y, Z, V), p(X, Y, U), p(X, Z,Xt), q(U,Xt,W).
q(X, k, X).
q(X, g(X), k).
r(U, V) :- q(X, Y, U), q(X, Y, V).
r(U, V) :- p(X, Y, U), p(X, Y, V).
p(X, Y, V) :- r(U, V), p(X, Y, U).
p(X, V, Y) :- r(U, V), p(X, U, Y).
p(V, Y, X) :- r(U, V), p(U, Y, X).
q(X, Y, V) :- r(U, V), q(X, Y, U).
q(X, V, Y) :- r(U, V), q(X, U, Y).
q(V, Y, X) :- r(U, V), q(U, Y, X).
r(f(X, U), f(X, V)) :- r(U, V).
r(j(X, U), j(X, V)) :- r(U, V).
r(g(U), g(V)) :- r(U, V).
q(g(X), X, k).
q(k, X, X).
r(X, X).
r(X, Y) :- r(Y, X).
r(X, Z) :- r(X, Y), r(Y, Z).
p(X, e, X).
p(e, X, X).
r(X, k) :- not p(X, h(X), e).
r(X, k) :- not p(h(X), X, e).
p(X, Y, Z) :- p(Y, X, Z).
r(f(U, X), f(V, X)) :- r(U, V).
r(h(U), h(V)) :- r(U, V).
r(j(U, X), j(V, X)) :- r(U, V).
q(b, c, d).
p(d, a, k).
```

```
p(b, a, l).
p(c, a, n).
false :- q(l, n, k).
end.
end_of_input.
```

26. wos31.

```
set(slidepriority).
assign(time_limit,100000).
assign(size_coef,1).
assign(depth_coef,2).
assign(literal_coef,2).
assign(relevance_coef,2).
assign(support_list,[u]).
assign(user_literalbound,4).
axiom_list(sprfn).
false :-r(m, b).
false :-r(b, k).
false :-r(k, m).
r(Y, m) :- p(Y, m), r(f(Y), m).
r(Y, m) :- p(Y, m), r(f(Y), Y).
r(Y, b) :- r(g(Y), b), not p(Y, b).
r(Y, b) :- r(g(Y), Y), not p(Y, b).
p(Y, k) :- r(Y, m), not r(Y, k).
r(Y, k) :- r(Y, b), not p(Y, k).
r(Y, k) :- p(Y, k), not r(Y, m), not r(Y, b).
r(Y, m) :- p(Y, V), p(V, Y), not p(Y, m), not r(V, m), not r(V, Y).
p(Y, f(Y)) :- p(Y, m), not r(Y, m).
p(f(Y), Y) :- p(Y, m), not r(Y, m).
p(Y, g(Y)) :- not r(Y, b) , not p(Y, b).
p(g(Y), Y) :- not r(Y, b) , not p(Y, b).
r(Y, b) :- p(Y, V), p(V, Y), p(Y, b), not r(V, b), not r(V, Y).
r(X, X).
r(Y, X) :- r(X, Y).
r(X, Z) :- r(X, Y), r(Y, Z).
r(f(X), f(Y)) :- r(X, Y).
r(g(X), g(Y)) :- r(X, Y).
p(Y, Z) :- r(X, Y), p(X, Z).
p(Z, Y) :- r(X, Y), p(Z, X).
```

```
end.
set_of_support.
1.
2.
3.
end.
end_of_input.
clear(slidepriority).
assign(time_limit,2000).
assign(size_coef,1).
assign(depth_coef,0).
assign(literal_coef,0).
assign(relevance_coef,0).
delete(user_literalbound,1).
```

27. wos33.

```
assign(support_list,[b,f]).
axiom_list(sprfn).
p(X,Z) :- p(X,Y),r(Y,Z).
p(Z,Y) :- r(X,Z),p(X,Y).
r(Y,a) :- not r(Y,k),p(Y,a).
r(Y,a) :- not p(Y,a),r(Y,k).
r(Y,n) :- p(Y,n),r(g(Y),n).
r(Y,n) :- p(Y,n),r(g(Y),Y).
r(Y,n) :- p(Y,n),not p(Y,g(Y)).
r(Y,n) :- p(Y,n),not p(g(Y),Y).
r(Y,n) :- not p(Y,n), not r(U,n), not r(U,Y), p(Y,U), p(U,Y).
r(Y,b) :- not p(Y,b), r(j(Y),b).
r(Y,b) :- not p(Y,b), r(j(Y),Y).
r(Y,b) :- not p(Y,b), not p(Y,j(Y)).
r(Y,b) :- not p(Y,b), not p(j(Y),Y).
r(Y,b) :- p(Y,b), not r(V,b), not r(V,Y), p(Y,V), p(V,Y).
r(Y,k) :- not p(Y,k), r(Y,n).
r(Y,k) :- not p(Y,k), r(Y,b).
r(Y,k) :- not r(Y,n), r(Y,b), p(Y,k).
r(X,X).
r(Y,X) :- r(X,Y).
r(X,Z) :- r(X,Y),r(Y,Z).
false :- r(b,a).
```

```
false :- r(n,b).
false :- r(b,k).
false :- r(n,a).
false :- r(k,a).
r(n,k).
end.
end_of_input.
```

28. ls5.

```
assign(support_list,[f]).
axiom_list(sprfn).
p(a) :- not(q(X)).
p(X) :- q(X).
false :- q(X), p(X).
q(X) :- p(X).
end.
end_of_input.
```

29. ls17.

```
assign(support_list,[f]).
axiom_list(sprfn).
false :- l(X, X).
l(X, f(X)).
false :- l(X, Y), l(Y, X).
d(X, X).
d(X, Z) :- d(X, Y), d(Y, Z).
false :- d(X, Y), l(Y, X).
l(Y, X) :- d(X, f(Y)).
p(a).
d(h(X), X) :- not(p(X)).
p(h(X)) :- not(p(X)).
l(h(X), X) :- not(p(X)).
l(f(a), X) :- p(X), l(a, X).
end.
end_of_input.
```

30. ls37.

```
assign(support_list,[r]).
```

```
set(slidepriority).
assign(time_limit,20000).
axiom_list(sprfn).
false :- t(a, e, e).
p(e, X, X).
p(X, e, X).
p(X, g(X), e).
p(g(X), X, e).
p(V, W, X1) :- t(X, Y, V), t(X, Z, W), t(X, U, X1), p(Y, Z, U).
p(V, W, X1) :- t(Y, X, V), t(Z, X, W), t(U, X, X1), p(Y, Z, U).
p(Z, U, W) :- p(X, Y, Z), p(Y, U, V), p(X, V, W).
p(X, V, W) :- p(X, Y, Z), p(Y, U, V), p(Z, U, W).
p(Y, X, Z) :- p(X, Y, Z).
p(X, Y, V) :- p(X, Y, U), eq(U, V).
p(X, Y, f(X, Y)).
t(X, U, X1) :- p(Y, Z, U), p(V, W, X1), t(X, Y, V), t(X, Z, W).
t(U, X, X1) :- p(Y, Z, U), p(V, W, X1), t(Y, X, V), t(Z, X, W).
t(Z, U, W) :- t(X, Y, Z), t(Y, U, V), t(X, V, W).
t(X, V, W) :- t(X, Y, Z), t(Y, U, V), t(Z, U, W).
t(X, Y, h(X, Y)).
eq(U, V) :- p(X, Y, U), p(X, Y, V).
end.
end_of_input.
clear(slidepriority).
```

31. ls65.

```
assign(support_list,[b,f]).
axiom_list(sprfn).
'=='(X, X).
'<'(a, b).
'<'(X, Y) :- '=='('+'(s(Z), X), Y).
'<'(X, Z) :- '<'(X, Y), '<'(Y, Z).
'=='('+'(X, 0), X).
'=='('*'(X, 0), 0).
'=='('*'(X, s(Y)), '+'('*'(X, Y), X)).
'=='('+'(X, s(Y)), s('+'(X, Y))).
'=='(Y, X) :- '=='(X, Y).
'=='(X, Y) :- '=='(s(X), s(Y)).
'=='(s(X), s(Y)) :- '=='(X, Y).
```

```
'=='(Y, Z) :- '=='(X, Y), '=='(X, Z).
'=='(X, Z) :- '=='(X, Y), '=='(Y, Z).
'=='('+'(s(f(X, Y)), X), Y) :- '<'(X, Y).
'=='(X, Y) :- divv(X, Y), not('<'(X, Y)).
divv(b, c).
divv(X, Y) :- '<'(X, Y).
divv(X, Y) :- '=='(X, Y).
false :- '<'(a, c).
false :- '=='(s(X), 0).
end.
end_of_input.
```

32. ls75.

```
assign(support_list,[b,f]).
axiom_list(sprfn).
'=='(X, X).
'=='('+'(X, 0), X).
'=='('*'(X, 0), 0).
'=='('+'(X, Y), '+'(Y, X)).
'=='('+'(X, s(Y)), s('+'(X, Y))).
'=='('*'(X, s(Y)), '+'('*'(X, Y), X)).
'=='(Y, X) :- '=='(X, Y).
'=='(X, Y) :- '=='(s(X), s(Y)).
'=='(s(X), s(Y)) :- '=='(X, Y).
'=='(X, Y) :- '=='('+'(X, Z), '+'(Y, Z)).
'=='('+'(s(f(X, Y)), X), Y) :- '<'(X, Y).
'=='(Y, Z) :- '=='(X, Y), '=='(X, Z).
'=='(X, Z) :- '=='(X, Y), '=='(Y, Z).
'<'(X, Y) :- '=='('+'(s(Z), X), Y).
'<'(a, a).
false :- '=='(s(X), 0).
end.
end_of_input.
```

33. ls87.

```
assign(support_list,[r]).
set(slidepriority).
axiom_list(sprfn).
false :- l(a, b).
```

```
false :- eq(c, 0).
l(m(a,c),m(b,c)).
eq(p(s(f(X, Y)),X),Y) :- l(X, Y).
eq(X, Y) :- eq(s(X), s(Y)).
eq(Y, Z) :- eq(X, Y), eq(X, Z).
eq(s(X), s(Y)) :- eq(X, Y).
eq(p(X, 0), X).
eq(p(X, s(Y)), s(p(X,Y))).
eq(m(X, 0), 0).
eq(m(X, s(Y)), p(m(X, Y), X)).
l(X, Y) :- eq(p(s(Z), X), Y).
eq(m(X, Z), m(Y, Z)) :- eq(X, Y).
l(X, Z) :- l(X, Y), l(Y, Z).
false :- eq(X, Y), l(X, Y).
l(X, Y) :- not eq(X, Y), not l(Y, X).
false :- l(X, X).
not eq(s(X), 0).
l(m(X, Z), m(Y, Z)) :- not eq(Z, 0), l(X, Y).
eq(Z, 0) :- not l(m(X, Z), m(Y, Z)), l(X, Y).
eq(X, X).
eq(Y, X) :- eq(X, Y).
eq(X, Z) :- eq(X, Y), eq(Y, Z).
end.
end_of_input.
clear(slidepriority).
```

34. ls103.

```
assign(support_list,[b]).
axiom_list(sprfn).
un(a, a, b).
m(Z, Y) :- s(X, Y), m(Z, X).
m(f(X, Y), X) :- not(s(X, Y)).
s(X, Y) :- m(f(X, Y), Y).
s(X, Y) :- e(X, Y).
s(Y, X) :- e(X, Y).
e(X, Y) :- s(X, Y), s(Y, X).
m(W, X) :- un(X, Y, Z), m(W, Z), not(m(W, X)).
m(W, Z) :- un(X, Y, Z), m(W, X).
m(W, Z) :- un(X, Y, Z), m(W, Y).
```

```
m(g(X, Y, Z), Z) :- not(m(g(X, Y, Z), X)), not(m(g(X, Y, Z), Y)),
                    not(un(X, Y, Z)).
un(X, Y, Z) :- m(g(X, Y, Z), X), m(g(X, Y, Z), Z).
un(X, Y, Z) :- m(g(X, Y, Z), Y), m(g(X, Y, Z), Z).
false :- e(b, a).
end.
end_of_input.
```

35. ls108.

```
assign(support_list,[b]).
axiom_list(sprfn).
in(a, b, c).
in(b, d, ap).
in(a, ap, bp).
s(X, Y) :- e(X, Y).
s(Y, X) :- e(X, Y).
s(X, Y) :- m(f(X, Y), Y).
e(X, Y) :- s(X, Y), s(Y, X).
m(Z, Y) :- s(X, Y), m(Z, X).
m(f(X, Y), X) :- not(s(X, Y)).
m(W, X) :- m(W, Z), in(X, Y, Z).
m(W, Y) :- m(W, Z), in(X, Y, Z).
m(W, Z) :- m(W, X), m(W, Y), in(X, Y, Z).
m(h(X, Y, Z), X) :- not(in(X, Y, Z)), not(m(h(X, Y, Z), Z)).
m(h(X, Y, Z), Y) :- not(in(X, Y, Z)), not(m(h(X, Y, Z), Z)).
in(X, Y, Z) :- m(h(X, Y, Z), X), m(h(X, Y, Z), Y), m(h(X, Y, Z), Z).
false :- in(c, d, bp).
end.
end_of_input.
```

36. ls112.

```
assign(support_list,[b]).
axiom_list(sprfn).
m(Z, Y) :- s(X, Y), m(Z, X).
m(f(X, Y), X) :- not s(X, Y).
s(X, Y) :- m(f(X, Y), Y).
s(X, Y) :- e(X, Y).
s(Y, X) :- e(X, Y).
e(X, Y) :- s(X, Y), s(Y, X).
```

```
m(W, Y) :- not m(W, X), m(W, Z), un(X, Y, Z).
m(W, Z) :- un(X, Y, Z), m(W, X).
m(W, Z) :- un(X, Y, Z), m(W, Y).
un(X, Y, Z) :- not m(g(X, Y, Z), Z),not m(g(X, Y, Z), X),
       not m(g(X, Y, Z), Y).
un(X, Y, Z) :- m(g(X, Y, Z), X), m(g(X, Y, Z), Z).
un(X, Y, Z) :- m(g(X, Y, Z), Y), m(g(X, Y, Z), Z).
m(W, X) :- in(X, Y, Z), m(W, Z).
m(W, Y) :- in(X, Y, Z), m(W, Z).
m(W, Z) :- in(X, Y, Z), m(W, X), m(W, Y).
m(h(X, Y, Z), Z) :- not m(h(X, Y, Z), X), in(X, Y, Z).
m(h(X, Y, Z), Z) :- not m(h(X, Y, Z), Y), in(X, Y, Z).
in(X, Y, Z) :- m(h(X, Y, Z), Z), m(h(X, Y, Z), X), m(h(X, Y, Z), Y).
un(b, c, ap).
in(a, b, bp).
in(a, c, cp).
in(a, ap, d).
false :- un(bp, cp, d).
end.
end_of_input.
```

37. ls116.

```
assign(support_list,[r]).
axiom_list(sprfn).
m(Z,Y) :- s(X,Y),m(Z,X).
false :- s(d,ap).
s(X,Y) :- m(f(X,Y),Y).
s(X,Y) :- not m(f(X,Y),X).
s(X,Y) :- e(X,Y).
s(Y,X) :- e(X,Y).
e(X,Y) :- s(X,Y), s(Y,X).
m(W,X) :- cm(X,Y,Z), m(W,Z).
false :- cm(X,Y,Z), m(W,Y), m(W,Z).
not cm(X,Y,Z) :-  m(W,Y), m(W,Z).
cm(X,Y,Z) :- not m(k(X,Y,Z),Z), not m(k(X,Y,Z),X).
m(k(X,Y,Z),Z) :- not m(k(X,Y,Z),X), not cm(X,Y,Z).
m(k(X,Y,Z),X) :- not m(k(X,Y,Z),Z), not cm(X,Y,Z).
m(k(X,Y,Z),Z) :- not cm(X,Y,Z), m(k(X,Y,Z), Y).
cm(X,Y,Z) :- m(k(X,Y,Z),X), not m(k(X,Y,Z),Y), m(k(X,Y,Z),Z).
```

```
s(a,b).
cm(c,b,d).
cm(c,a,ap).
m(W,Z) :- not m(W,Y), cm(X,Y,Z), m(W, X).
m(W,Y) :- not m(W,Z), cm(X,Y,Z), m(W, X).
end.
end_of_input.
```

38. chang_lee_2.

```
assign(support_list,[f]).
axiom_list(sprfn).
p(e,X,X).
p(X,e,X).
p(X,X,e).
p(a,b,c).
false :- p(b,a,c).
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).
p(X,V,W) :- p(X,Y,U), p(Y,Z,V), p(U,Z,W).
end.
end_of_input.
```

39. chang_lee_8.

```
assign(support_list,[f]).
axiom_list(sprfn).
l(1,a).
d(X,X).
false :- d(X,a),p(X).
d(X,Z) :- d(Y,Z),d(X,Y).
p(f(X)) :- l(1,X),l(X,a).
d(f(X),X) :- l(1,X),l(X,a).
d(g(X),X) :- not(p(X)).
l(1,g(X)) :- not(p(X)).
l(g(X),X) :- not(p(X)).
end.
end_of_input.
```

40. chang_lee_9.

```
assign(support_list,[f]).
```

```
axiom_list(sprfn).
l(X,f(X)).
false :- l(X,X).
false :- l(X,Y),l(Y,X).
l(Y,X) :- d(X,f(Y)).
l(f(a),X) :- l(a,X),p(X).
d(h(X),X) :- not(p(X)).
p(h(X)) :- not(p(X)).
l(h(X),X) :- not(p(X)).
end.
end_of_input.
```

41. schubert.

```
assign(support_list,[r]).
axiom_list(sprfn).
w(w).
b(b).
s(s).
f(f).
c(c).
g(g).
a(X) :- w(X).
a(X) :- b(X).
a(X) :- s(X).
a(X) :- c(X).
a(X) :- f(X).
p(X) :- g(X).
m(X,Y) :- c(X),b(Y).
m(X,Y) :- b(X),f(Y).
m(X,Y) :- s(X),b(Y).
m(X,Y) :- f(X),w(Y).
false :- f(X),w(Y),e(Y,X).
false :- w(Y),g(X),e(Y,X).
false :- b(X),s(Y),e(X,Y).
e(X,Y) :- b(X),c(Y).
p(h(X)) :- c(X).
p(i(X)) :- s(X).
e(X,h(X)) :- c(X).
e(X,i(X)) :- s(X).
```

```
g(j(X,Y)) :- not(na(X)),not(na(Y)).
false :- not(na(Y)),not(na(X)),e(X,Y),e(Y,j(X,Y)).
e(X,Y) :- split2(X),p(Y).
split2(X) :- m(Z,X),a(X),a(Z),split(Z),not(e(X,Z)).
split(Z) :- e(Z,W),p(W).
false :- na(X),a(X).
end.
end_of_input.
```

Appendix C

User Manual

This user manual describes how to use CLIN. There are a number of user settable flags, but many of them are for experimentation and can be ignored under normal use. CLIN is written in Prolog. It can be run with either C-Prolog [Per84] or ALS-Prolog [ALS88].

CLIN can be used in an automatic way, that is, by triggering the top-level supervisor. It can also be used in a controlled way; the user specifies desired options and decides support strategies before the prover tries on the input problem. The best way of using CLIN may be to call the top-level supervisor first. If it fails, then the user may try again with customized settings and support strategies.

C.1 Using CLIN

In order to use CLIN, the user basically has to do the following steps:

1. Prepare an input file. The format of the file will be explained later.
2. Run C-Prolog or ALS-Prolog. Usually, the user has to allocate more memory for stacks and heap when using C-Prolog, see C-Prolog manual [Per84] for details. Both C-Prolog and ALS-Prolog can be run on SUN3.
3. Consult the prover. A welcome message will come up on the screen.
4. Try the input file. Type in

   prove(File)

   where File is the input file name containing the problem to be solved.

C.2 Running Modes

CLIN can be run in different ways.

1. interactive mode: The user runs Prolog and consults the prover file. Commands can be given to set up the environment. Then using the command

   prove(File)

   to try the problem. The prover will output the results and some statistics on the screen.

2. batch mode: On UNIX or UNIX-like systems, the user can prepare a file IN which includes a consult command (Prolog built-in command), CLIN commands, a list of "prove" CLIN commands, ended by a "halt" command (Prolog built-in command). Then using the UNIX command:

   Prolog <IN >OUT

   The problems will be solved one by one and the outputs will be placed in the output file OUT. For example, the following can be included in the file IN:

   ```
   consult the prover.
   CLIN commands.
   prove(F1).
   prove(F2).
        ...
   prove(FN).
   halt.
   ```

   F1, F2, ..., FN are input files for the problems to be solved.

3. supervisor mode: Will be discussed later.

C.3 Comments

Comments can be placed in any file read by Prolog or the prover. They are preceded by the symbol %. All characters from the first % on a line to the end of the line are ignored.

C.4 Names

Names are used as constant symbols, function symbols, predicate symbols, propositional constants, individual variables, and commands. The syntactic structure of a name follows the conventions in Prolog. In general, the type (predicate symbol, function symbol, constant) of a name is determined by its context.

C.5 Input File

The input file containing the problem to be solved consists of commands and lists of objects (clauses, support set, weights, replace rules). All lists of objects are terminated with "end". In this section, only those commands associated with a list of objects are considered; the other commands will be discussed later.

C.5.1 The Structure of an Input File

A typical input file is structured as follows:

```
commands.
axiom list.
weight list.
replace rule list.
support list.
end_of_input.
commands.
```

An entry can be omitted. The order of the entries is not important except the position of "end_of_input". CLIN starts to solve the input problem when it reads "end_of_input". When done, it executes the commands following "end_of_input" until the end of the file.

All the commands before "end_of_input" are executed so as to create the desirable environment for the specific input problem. The commands after "end_of_input" are used to restore the normal environment.

C.5.2 Axiom List

The axiom list encodes the axioms and the negation of the conclusion of the input problem. It has the following form:

```
axiom_list(Option).
axiom-1.
axiom-2.
 ...
axiom-n.
end.
```

Option specifies the form of each axiom-i. Two forms are accepted by CLIN:

- clause: Each axiom is written as a list structure

  $$[L_1, L_2, \ldots, L_k]$$

  Each L_i is a literal. The list is interpreted as

  $$L_1 \vee L_2 \vee \cdots \vee L_k$$

- sprfn: Each axiom is written in the format

  $$L \text{ :- } M_1, M_2, \ldots, M_m$$

  where L is a positive literal, or "false" if the axiom is negative. L is called the *head* and M_1, M_2, ..., M_m consist of the *body* of the axiom. The above expression is interpreted as

  $$\mathrm{L} \vee \neg M_1 \vee \neg M_2 \vee \cdots \vee \neg M_m$$

A negative literal is represented by "not" followed by an atom. For example, ¬p(X,Y) is represented by not(p(X,Y)). The pair of parentheses for "not" is optional.

C.5.3 Weight List

Two kinds of weights are provided. One deals with the size of a term. The other deals with the depth of a term. The structure of a weight list is as follows:

```
weight_list.
weight-1.
weight-2.
  ...
weight-n.
end.
```

Each weight-i can be one of the following:

- nonvar_depth_weight, Term, Weight.
- var_depth_weight, Term, Weight.
- nonvar_size_weight, Term, Weight.
- var_size_weight, Term, Weight.

The first argument specifies the type. The second argument is the term to be assigned a weight. The third argument is the value of the weight. A weight can be any real number. The size of a term is the number of symbols in the term when it is written as a character string. The depth of a term is one more than the number of recursive application when the term is defined. For example, f(g(h(X), Y), a) has size 6 and depth 4. We don't count "not" for the size and depth of a term in the computation of priority values of clauses. Thus not(f(g(h(X), Y), a)) still has size 6 and depth 4. By specifying weights, the user is able to change the way the size or the depth of a term is computed. For example, if the following weights are specified:

```
nonvar_depth_weight, a, 0.
var_depth_weight, X, 3.
nonvar_size_weight, a, 0.
var_size_weight, X, 3.
```

then f(g(h(X), Y), a) has size 7 and depth 6.

C.5.4 Replace Rule List

There are two kinds of replace rules: repeated replace rules and once replace rules. The structure of a replace rule list is as follows:

```
Rule-Type(Option).
rule-1.
rule-2.
 ...
rule-n.
end.
```

Rule-type is either repeated_replace_rule for repeated replace rules or once_replace_rule for once replace rules. Both kinds of replace rules may appear in an input file. Option indicates the specified rule is in sprfn format or in clause format as described in the section C.5.2. Each rule-i is of the following format:

Axiom,Distinguished-List,Context-List

Axiom is either in sprfn format, or in clause format. Distinguished-List is a list of ordinal number which specifies which literals in Axiom are distinguished literals. Context-List is a list of ordinal number which specifies which literals in Axiom are context literals. For example, suppose [a(X), b(Y), c(Z)] is an axiom with b(Y) and c(Z) being distinguished literals and c(Z) being a context literal. This is represented by the following rule:

[a(X), b(Y), c(Z)], [2,3], [3].

The above rule can also be represented, in the sprfn format, as follows:

(a(X) :- ¬b(Y), ¬c(Z)), [2,3], [3].

If the head of axiom in sprfn format is "false", we don't count it. This is illustrated in the following rule:

(false :- bb(Y), cc(Z)), [1,2], [2].

where bb(Y) and cc(Z) are the first and second literals respectively.

C.5.5 Support List

If the user uses user support strategy, then the support set must be specified. A support set is provided as follows:

```
set_of_support.
Ord-1.
Ord-2.
 ...
Ord-n.
end.
```

Ord-i is the ith axiom in the axiom list described in section C.5.2. For example, suppose axioms 2 and 4 are in the support set. They can be indicated by

```
set_of_support.
2.
4.
end.
```

C.5.6 An Example

The following is the input file for a problem which says that if for all elements X in a group such that X$\times$X = e, then the group is commutative.

```
assign(support_list,[f]).
axiom_list(sprfn).
p(e,X,X).
p(X,e,X).
p(X,X,e).
p(a,b,c).
false :- p(b,a,c).
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).
p(X,V,W) :- p(X,Y,U), p(Y,Z,V), p(U,Z,W).
end.
end_of_input.
```

The first line is a CLIN command.

C.6 CLIN Commands

CLIN provides two kinds of commands. One is to set/clear a flag. The other is to assign/delete a value to a global parameter. Through the execution of commands, CLIN creates the desired running environment for a given input problem.

If the command given is not a valid one, that is, one that is known to CLIN, it is rejected.

C.6.1 Set/Clear Commands

To set a flag, Flag, use

set(Flag).

To reset a flag, Flag, use

clear(Flag).

The allowed flags are discussed below.

C.6.1.1 Output Flags

These commands ask CLIN to print various kinds of output.

- outCagen — default clear. If this flag is set, the stored clauses in the database will be printed out after each round of hyper-linking and predicate replacement.
- outCahm — default clear. If this flag is set, the stored clauses in the database will be printed out after each round of hyper-linking.
- outCainst — default clear. If this flag is set, the stored clauses in the database will be printed out after each round of instance deletion.
- outCasimp — default clear. If this flag is set, the stored clauses in the database will be printed out after each round of unit simplification.
- outfms — default clear. If this flag is set, the input clauses to the propositional calculus prover are printed out in each round.
- outinst — default clear. If this flag is set, each nucleus and its instances are printed out during hyper-linking and predicate replacement.
- outhmlits — default clear. If this flag is set, the literal list for hyper-linking in each round is printed out.

- outpc — default clear. If this flag is set, a model is printed out if the input problem is satisfiable, or a subset of unsatisfiable clauses is printed out if the input problem is unsatisfiable detected by the propositional calculus prover.
- outpcin — default clear. If this flag is set, the input clauses to the last call of the propositional calculus prover are printed out if the input is satisfiable, or if the input is unsatisfiable detected by the propositional calculus prover.

The output information includes literals or clauses, support status, and distances.

C.6.1.2 Control Flags

These commands are used to control the prover.

- count_all_literals — default set. If this flag is set, count all literals when computing the argument associated with literal count in the computation of priority values of clauses.
- delete_all_instances — default clear. If this flag is set, delete any clause that is an instance of another clause.
- delete_nf_instances — default clear. If this flag is set, delete any non-forward supported clause that is an instance of another clause.
- do_hyper_matching — default set. If this flag is set, perform hyper-linking in each round.
- ground_disting_after_match — default clear. If this flag is set, it is required that all distinguished literals be ground after matching in predicate replacement.
- ground_restriction — default clear. If this flag is set, the instance generated in predicate replacement must be ground.
- ground_substitution — default clear. If this flag is set, the electrons used for matching distinguished literals in predicate replacement must be ground.
- hl_literal_reordering — default set. If this flag is set, the ground literals are picked to link first during hyper-linking.
- realfls — default clear. If this flag is set, the input clauses to the propositional calculus prover is the ground set of a fully-linked subset of the retained clauses.
- replace_literal_reordering — default set. If this flag is set, the ground literals are picked to match first in predicate replacement.

- replacement — default clear. If this flag is set, the facility, predicate replacement, is used for generating instances.
- simple_small_proof_check — default clear. If this flag is set, only unit electrons are used in small proof checking.
- size_by_clause — default clear. If this flag is set, use the size of the whole clause for size argument in the computation of priority values of clauses. Otherwise, use the maximum size of constituent literals in a clause for size argument.
- slidepriority — default clear. If this flag is set, the sliding priority is used to control the use of memory.
- small_proof_check — default set. If this flag is set, small proof checking is used to detect unsatisfiability.
- small_proof_check_all — default clear. If this flag is set, use all clauses as nuclei in small proof checking. Otherwise, use only input clauses as nuclei.
- spc_literal_reordering — default set. If this flag is set, the ground literals are picked to link first during small proof checking.
- sum_of_measures — default clear. If this flag is set, the priority value of a clause is the linear sum of its four arguments: size, depth, literal count, and distance. Otherwise, the priority value is the maximum value of the four arguments.
- super_batch — default clear. If this flag is set, batch-mode processing is used in the top-level supervisor.
- tautology_deletion — default set. If this flag is set, a clause that is a tautology is deleted.
- unit_simplification — default set. If this flag is set, unit simplification is performed in each round.
- user_control — default set. If this flag is set, do not use the top-level supervisor.

C.6.2 Assign/Delete Commands

To assign a value, Val, to a parameter, Parm, use

assign(Parm, Val)

To delete an entry of a certain parameter, Parm, from the database, use

$$\text{delete(Parm, Arity)}$$

where Arity is the arity of the parameter Parm. CLIN stores the parameter and its value in the form

$$\text{Parm(Val)}$$

All the parameters presented in this section have arity 1. The following are parameters accepted by CLIN.

- back_literalbound — default is ∞. The value can be any natural number. It is a bound for the number of literals in a backward supported clause.
- depth_coef — default is 0. The value can be any real number. It specifies the coefficient value for the depth argument in the computation of priority values of clauses.
- for_literalbound — default is ∞. The value can be any natural number. It is a bound for the number of literals in a forward supported clause.
- greatest_time_bound — default is 800. The value can be any real number. It is used in the top-level supervisor. When the time limit exceeds greatest_time_bound, the supervisor stops.
- least_time_bound — default is 200. The value can be any real number. It is used in the top-level supervisor. The time limit starts from least_time_bound and increases by the amount specified by time_limit_multiple each time.
- literal_bound — default is ∞. The value can be any natural number. It is a bound for the number of literals in a clause.
- literal_coef — default is 0. The value can be any real number. It specifies the coefficient value for the literal count argument in the computation of priority values of clauses.
- max_no_clauses — default is 1600. The value can be any natural number. It specifies the maximum number of clauses that are allowed in the database when using sliding priority.
- relevance_bound — default is ∞. The value can be any natural number. The distance of an accepted clause should not exceed this bound.
- relevance_coef — default is 0. The value can be any real number. It specifies the coefficient value for the distance argument in the computation of priority values of clauses.

- size_coef — default is 1. The value can be any real number. It specifies the coefficient value for the size argument in the computation of priority values of clauses.
- small_proof_size_bound — default is ∞. The value can be any natural number. It specifies the proof size bound in small proof checking. The bigger the value is, the earlier a small proof may be obtained, but the more time small proof checking takes.
- start_no_clauses — default is 200. The value can be any positive integer. It specifies the starting number of clauses to be stored at the same time when using sliding priority. If the value is not specified, CLIN uses start_no_clauses_coef times the number of input clauses.
- start_no_clauses_coef — default is 3. The value can be any positive integer. This is used for scaling the starting number of clauses to be stored when using sliding priority.
- start_wu_bound — default is the value computed by using start_no_clauses_coef and start_no_clauses. CLIN uses work bound to control the use of memory when sliding priority is used. The sum of the priorities of all stored clauses should not exceed the work bound.
- support_list — default is [b,f]. This is to specify the support strategies that CLIN uses for solving the given problem. Several strategies can be used:
  - user support(u): The user has to supply the support set, usually the clauses corresponding to the negation of the conclusion.
  - backward support(b): The set of all negative clauses.
  - forward support(f): The set of all positive clauses.
  - UR resolution(r).
  - unit clauses(o): The set of all unit clauses.
  - negative clauses(n): The subset of all negative clauses in the support set specified by the user.

  Different strategies can be specified by concatenation, indicating that they are in effect simmultaneously. For example, [uf] means user support and forward support are used at the same time. Strategies separated by commas means that they are used interleavingly. For example, [b,f] means backward support is used in the first round, forward support is used in the second round, backward support is used in the third round, and so on.
- time_limit — default is 2000 seconds. The value can be any real number. CLIN stops if the time spent for a given problem exceeds the actual time limit which is time_limit_coef $\times$ time_limit.

- time_limit_coef — default is 1. The value can be any real number. The value is used for scaling the specified time limit.
- time_limit_multiple — default is 2. The value can be any real number. It is used in the top-level supervisor to increase the time limit each time.
- user_literalbound — default is ∞. The value can be any natural number. It is a bound for the number of literals in a user supported clause.

The ground literals are not counted when using for_literalbound, back_literalbound, literal_bound, or user_literalbound.

C.7 The Top-Level Supervisor

When using the supervisor, The user is allowed to change the values of least_time_bound, greatest_time_bound and time_limit_multiple.

The supervisor can be run in two modes:

- interactive mode: The user type in

  prove(File)

  where File is an input file containing the problem P to be solved. The user may change the default values for least_time_bound, greatest_time_bound and time_limit_multiple before typing in the above command. The output comes up on the screen. In this mode, the supervisor tries all strategies one after another. If P is detected unsatisfiable, then P is solved. Otherwise the supervisor increases the time limit and tries all strategies again.

- batch mode: The user types in

  prove(File)

  File contains a list of problems to be solved

```
prove(P-1).
prove(P-2).
   ...
prove(P-n).
```

The user may change the default values for least_time_bound, greatest_time_bound and time_limit_multiple in the file File. In this mode, the first strategy with the first time bound is tried on all problems, then the second strategy with the first time bound is tried on all problems not already solved, and so on. In this way, the theorems that can be proven quickly are solved early, and the theorems that are difficult do not delay finding the easy proofs.

Note that in UNIX or UNIX-like system, the supervisor can read the standard input and write to the standard output:

Prolog <IN >OUT

IN has the following form:

```
consult the prover.
assign(least_time_bound, 50).
assign(greatest_time_bound, 200).
assign(time_limit_multiple, 4).
prove(File).
halt.
```

File is either an input file for the interactive mode or a list of prove statements for the batch mode that has been described above.

C.8 Output Information

During the process of proof, CLIN prints out some information on the screen.

C.8.1 Error Message

- Number of variables overflows in input clauses — The number of distinct variables allowed to be in a clause is 20. If the number of variables in an input clause exceeds 20, this error comes up and the running stops.
- Number of variables overflows in instances — If the number of variables in an instances generated in hyper-linking or predicate replacement, this error comes up. The prover discards the instance and goes on.

- Syntactic error in input file — If CLIN finds a syntactic error in the input file, it stops running the problem.
- No unit clauses — This happens when the user tries to use UR resolution or use unit clauses as user support set while there are no unit clauses in the input file. CLIN stops running the problem.
- No user support set — If the user uses user support strategy without specifying user support set, CLIN stops running the problem.
- No negative clauses — It happens if the user uses user support strategy and specifies the negative clauses in the specified user support set to be the actual support set, but there is no negative clause in the specified user support set. In this case, CLIN stops running the problem.

C.8.2 Display Message

CLIN prints out some information on the screen during the process of proof. This information tells the user what stages have been gone through and how much time was spent on a certain stage.

C.8.3 Results and Statistics

Finally, CLIN prints out results and statistics for the problem it tried. File names and the number of input clauses are printed out first, followed by the options set for the problem that was tried, and then some statistics are printed out. There are some possible outcomes from the prover:

- unsatisfiable — It means that the input problem is unsatisfiable.
- satisfiable — It means that the input problem is satisfiable.
- time_overflow — It means that the total time spent for the problem exceeds the specified time limit.
- clause_no_overflow — It means that the number of clauses retained in the database exceeds the limit set by the command max_no_clauses.
- misc_overflow — It means that none of the above applies. This may happen when the user uses literal bounds, relevance bound, etc..

If a proof was found, CLIN also indicates how the proof was obtained.

- PC — It means that the proof was obtained by the propositional calculus prover.
- US — It means that the proof was obtained by unit simplification.
- SP — It means that the proof was obtained by small proof checking.
- HM — It means that the proof was obtained by hyper-linking or predicate replacement, i.e. an empty instance was generated.

It prints out the sum of the sizes of all clauses retained in the database. If the proof is obtained by small proof checking, the small proof, in terms of the nucleus and electrons, that was found is also printed out.

Bibliography

[Aba87] M. Abadi. *Temporal-Logic Theorem Proving*. PhD thesis, Stanford University, 1987.

[ALS88] Applied Logic Systems, Inc. *ALS Prolog User's Guide and Reference Manual*. Syracuse, New York, March 1988.

[And81] P. B. Andrews. Theorem proving via general matching. *Journal of the Association for Computing Machinery*, 28:193–214, 1981.

[BH80] W. W. Bledsoe and L. Hines. Variable elimination and chaining in a resolution-based prover for inequalities. In *Proceedings of the 5th International Conference on Automated Deduction*, pages 70–87, 1980.

[Bib82] W. Bibel. *Automated Theorem Proving*. Vieweg, 1982.

[Ble77] W. W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–35, 1977.

[BLM+86] R. Boyer, E. Lusk, W. McCune, R. Overbeek, M. Stickel, and L. Wos. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2:287–327, 1986.

[CL73] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[DaPu60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[Dav63] M. Davis. Eliminating the irrelevant from machanical proofs. In *Proceedings Symp. of Applied Math*, volume 15, pages 15–30, 1963.

[DePe88] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[dK86a] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[dK86b] J. de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28:197–224, 1986.

[dK89] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of 11th International Joint Conference on Artificial Intelligence*, pages 290–296, 1989.

[Fre78] E. C. Freuder. Synthesizing constraint expressions. *Communications of the Association for Computing Machinery*, 21:958–966, 1978.

[Gil60] P. C. Gilmore. A proof method for quantification theory. *IBM Journal of Research and Development*, 4:28–35, 1960.

[Hin88] L. Hines. Hyper-chaining and knowledge-based theorem proving. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 469–486, 1988.

[HW74] L. Henschen and L. Wos. Unit refutations and Horn sets. *Journal of the Association for Computing Machinery*, 21:590–605, 1974.

[JP84] S. Jefferson and D. Plaisted. Implementation of an improved relevance criterion. In *First Conference on Artificial Intelligence Applications*, pages 476–482, 1984.

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

[Kow75] R. Kowalski. A proof procedure using connection graphs. *Journal of the Association for Computing Machinery*, 22:572–595, 1975.

[Lew80] H. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21:317–353, 1980.

[LO85] E. Lusk and R. Overbeek. Non-horn problems. *Journal of Automated Reasoning*, 1:103–114, 1985.

[Lov78] D. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, New York, 1978.

[LP89] S.-J. Lee and D. Plaisted. Theorem proving using hyper-matching strategy. In *Proceedings of the 4th International Symposium on Methodologies for Intelligent Systems*, pages 467–476, 1989.

[LP90a] S.-J. Lee and D. Plaisted. New applications of a fast propositional calculus decision procedure. In *Proceedings of the 8th Biennial Conference of Canadian Society for Computational Studies of Intelligence*, pages 204–211, 1990.

[LP90b] S.-J. Lee and D. Plaisted. Reasoning with predicate replacement. In *Proceedings of the 5th International Symposium on Methodologies for Intelligent Systems*, to appear, 1990.

[Mac77] A. K. Mackworth. Consistency in network of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mac87] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. John Wiley and Son, 1987.

[McA80] D. McAllester. An outlook on truth maintenance. Technical Report AIM-551, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1980.

[McA82] D. McAllester. Reasoning utility package user's manual. Technical Report AIM-667, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1982.

[MCA82] D. Miller, E. Cohen, and P. B. Andrews. A look at TPS. In *Proceedings of the 6th International Conference on Automated Deduction*, pages 50–69, 1982.

[McC89] W. W. McCune. *Otter 1.0 Users' Guide.* Mathematics and Computer Science Division, Aggonne National Laboratory, Argonne, Illinois, January 1989.

[MP83] Z. Manna and A. Pnueli. Verification of concurrent programs: a temporal proof system. Technical report, Dept. of Computer Science, Stanford University, 1983.

[MR86] N. Murray and E. Rosenthal. Theory links in semantic graphs. In *Proceedings of the 8th International Conference on Automated Deduction*, pages 353–364, 1986.

[MW88] D. Maier and D. S. Warren. *Computing with Logic.* The Benjamin/Cummings Publishing Co., Menlo Park, California, 1988.

[Nie89] X. Nie. *Implementation Techniques in Automatic Theorem Proving.* PhD thesis, University of North Carolina at Chapel Hill, 1989.

[NO80] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27:356–364, 1980.

[NP90] X. Nie and D. Plaisted. A complete semantic back chaining proof system. In *Proceedings of the 10th International Conference on Automated Deduction*, 1990.

[NP90] X. Nie and D. Plaisted. Refinements to depth-first iterative deepening search in theorem proving. *Artificial Intelligence*, 41:223–235, 1989/90.

[OMW76] R. J. Overbeek, J. Mecharen, and L. Wos. Complexity and related enhancements for automated theorem-proving programs. *Computers and Mathematics with Applications*, 2:1–16, 1976.

[Ost89] J. S. Ostroff. *Temporal Logic for Real-Time Systems.* Research Studies Press, England, 1989.

[Pel86] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[Per84] F. Pereira. *C-Prolog User's Manual.* SRI International, Menlo Park, California, August 1984.

[PG86] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[PL90] D. Plaisted and S.-J. Lee. Inference by clause linking. Technical Report TR90-022, University of North Carolina – Chapel Hill, Chapel Hill, NC, 1990.

[Pla80] D. Plaisted. An efficient relevance criterion for mechanical theorem proving. In *Proceedings of the First Annual National Conference on Artificial Intelligence*, pages 79–83, 1980.

[Pla81] D. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

[Pla82] D. Plaisted. A simplified problem reduction format. *Artificial Intelligence*, 18:227–261, 1982.

[Pla86] D. Plaisted. A decision procedure for combinations of propositional temporal logic and other specialized theories. *Journal of Automated Reasoning*, 2:171–190, 1986.

[Pla88] D. Plaisted. Non-Horn clause logic programming without contrapositives. *Journal of Automated Reasoning*, 4:287–325, 1988.

[Pla90] D. Plaisted. Mechanical theorem proving. In R. Banerji, editor, *A Sourcebook on Formal Techniques in Artificial Intelligence*. Elsevier, Amsterdam, 1990.

[PM88] D. Plaisted and U. Meyer. A matching strategy for first-order theorem proving. Unpublished, 1988.

[PP88] R. Potter and D. Plaisted. Term rewriting: Some experimental results. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 435–453, 1988.

[PP90] H. Przymusinska and T. Przymusinski. Semantic issues in deductive databases and logic programs. In R. Banerji, editor, *A Sourcebook on Formal Techniques in Artificial Intelligence*. Elsevier, Amsterdam, 1990.

[PPV60] D. Prawitz, H. Prawitz, and N. Voghera. A mechanical proof procedure and its realization in an electronic computer. *Journal of the Association for Computing Machinery*, 7:102–128, 1960.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Rob63] J. Robinson. Theorem proving on the computer. *Journal of the Association for Computing Machinery*, 10:163–174, 1963.

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

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32:733–749, 1985.

[SM77] D.F. Stanat and D.F. McAllister. *Discrete Mathematics in Computer Science*. Prentice Hall, Englewood Cliffs, N.J., 1977.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 1986.

[ST85] M.E. Stickel and W.M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 1073–1075, 1985.

[Sti81] M.E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28:423–434, 1981.

[Sti85] M.E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1:333–355, 1985.

[Sti86] M.E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. In *Proceedings of the 8th International Conference on Automated Deduction*, pages 573–587, 1986.

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

[Wal84] C. Walther. A mechanical solution of schubert's steamroller by many-sorted resolution. In *Proceedings of the 4th National Conference on Artificial Intelligence*, pages 330–334, 1984.

[WB87] T.C. Wang and W.W. Bledsoe. Hierarchical deduction. *Journal of Automated Reasoning*, 3:35–77, 1987.

[WOLB84] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications*. Prentice Hall, Englewood Cliffs, N.J., 1984.

[Wos88] L. Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice Hall, Englewood Cliffs, N.J., 1988.

[WRC65] L. Wos, G. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the Association for Computing Machinery*, 12:536–541, 1965.