# Enhancing a Theorem Prover by Delayed Clause-Construction and Attribute Sequences

Paul Haroun

Department of Computer Science
McGill University, Montreal

Oct 2005

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor in Philosophy

*Your file Votre référence*
*ISBN: 978-0-494-21651-4*
*Our file Notre référence*
*ISBN: 978-0-494-21651-4*

# Abstract

The performance of a resolution-based automated theorem prover (ATP) depends on the speed at which clauses are derived and the efficiency at pruning the search space. The speed at which clauses are derived depends in part on the number of operations performed to construct derived clauses. Depth-first search based ATPs derive clauses in a linear manner. In linear derivations, a large percentage of the derived clauses are intermediate conclusions that are discarded shortly after they are derived. Therefore, the time spent constructing those clauses is wasted. In this thesis we present a stalling strategy, called *delayed clause-construction* (DCC), that reduces this wasted time by delaying the construction of intermediate conclusions until they are needed.

Top-down depth-first search algorithms have the disadvantage of deriving the same clauses over and over again. Bottom-up best-first search approaches solve this problem by redundancy elimination, but their disadvantages are the lack of goal-orientation and the large memory requirements. In this thesis we introduce *semi-linear resolution* (SLR), a top-down bottom-up search procedure that combines advantageous characteristics found in best-first search and depth-first search algorithms. It requires a modest amount of memory and includes redundancy control. SLR relies on DCC for speed. DCC also provides SLR with ability to perform large inference steps through the use of a mega-inference rule.

In order to improve the efficiency of SLR, we developed a restriction strategy, called *attribute sequences* (ATS), that uses sequences of clause characteristics as a guide to limit the participation of clauses in a linear derivation, thereby reducing the *explorable search space*. ATS does not compromise completeness.

The performance enhancements ensuing from the use of DCC and ATS in SLR are shown in this thesis to be quite significant in theory, through mathematical

analysis, and in practice, through the results obtained from CARINE; an implementation of SLR .

# Sommaire

La performance d'un logiciel d'aide à la preuve (*Automated Theorem Prover* – ATP) basé sur la méthode de résolution dépend, d'une part de la vitesse à laquelle les clauses sont dérivées, et d'autre part de l'efficacité de l'élagage de l'espace de recherche. La vitesse dépend, en partie, du nombre des opérations nécessaires pour construire les clauses dérivées. Les ATPs basés sur la recherche en profondeur dérivent des clauses de façon linéaire. La plupart de ces clauses, considérées comme des conclusions intermédiaires, sont éliminées juste après qu'elles aient été dérivées. De plus, juste un terme ou un littéral d'une conclusion intermédiaire est généralement utilisé lors de l'application d'une règle d'inférence. Par conséquent, le temps investi à construire une conclusion intermédiaire entière est gaspillé. Dans cette thèse, nous présentons une méthode nommée construction de clause différée (*Delayed Clause-Construction* – DCC) qui réduit ce temps gaspillé en retardant, jusqu'à l'apparition d'un besoin, la construction de conclusions intermédiaires dans une dérivation linéaire.

Les algorithmes descendants de recherche en profondeur possèdent l'inconvénient de dériver les mêmes clauses à plusieurs reprises. Les approches ascendantes de recherche du meilleur résolvent ce problème en éliminant la redondance. Leur inconvénient est cependant lié aux besoins en terme de mémoire et, généralement, au manque d'orientation vers le but. Nous introduisons la résolution semi-linéaire (*Semi-Linear Resolution* – SLR) qui combine les meilleures caractéristiques de l'algorithme de la recherche en profondeur et de celui de la recherche du meilleur. Il requiert peu d'espace mémoire et est assez flexible pour générer de nouvelles règles d'inférence, de nouvelles stratégies, et de nouvelles règles pour le contrôle de la redondance et pour la simplification. L'usage de DCC dans SLR contribue à l'amélioration de la performance. La

résolution SLR profite également de la stratégie DCC pour pouvoir exécuter de grandes étapes d'inférence à travers l'utilisation d'une règle de méga-inférence.

Afin d'améliorer l'efficacité de SLR, nous présentons une stratégie basée sur les attributs des clauses (*Attribute Sequences* – ATS). ATS sont utilisées comme guide pour réduire l'espace de recherche sans toutefois sacrifier la complétude.

Cette thèse montre que les améliorations de la performance qui proviennent de l'usage de DCC et de ATS dans SLR sont assez significatives. Ceci est prouvé du point de vue théorique à travers les analyses mathématiques, et du point de vue pratique par les résultats obtenus de l'implémentation expérimentale d'un logiciel d'aide à la preuve, nommé CARINE.

# Acknowledgments

First and foremost, I thank God for bestowing upon me the ability to complete this thesis after many years of continuous effort and sacrifice.

I thank my supervisor, Professor Monty Newborn, for his tremendous patience, invaluable feedback, his support to enter the theorem proving competition (CASC), for all the time he spent carefully reading, correcting and reviewing the thesis and for all the long hours of discussions on theorem proving.

I thank my parents for their support which helped me achieve this level of education. I thank my brother Antoine who always believed in me and continuously encouraged me; something that I really needed especially in the last 6 months of writing the thesis. I also thank him for his feedback on the presentation of the thesis.

I thank my friends Sarita Bassil and Danielle Azar for their help with the translation of the abstract into French. The abstract is not that difficult that it requires two experts to translate it, but because I changed it several times, I had to ask whoever was available to help me out.

I also thank Sarita for her feedback and review of parts of this thesis and for her help in extracting and formatting some final results obtained from the experiments I conducted on CARINE. Her efficiency is impressive.

I thank Geoff Sutcliffe for providing me with the initial results obtained from running CARINE over the TPTP library and for his encouragement and support during CASC. I also thank him for his continuous encouragement to improve CARINE and for his revision of the thesis.

I thank Geoff Sutcliffe and Christian Suttner for doing an exemplary job in maintaining the TPTP problem library and for making it available for free to all interested users. I thank the organizers of CASC for their enthusiasm and dedication for organizing a competition that motivates theorem prover authors to

improve their theorem provers. It also facilitates the personal interaction and sharing of ideas between computer scientists and users of theorem provers from all over the world.

I thank all the contributors who presented the difficult subject of automated reasoning in a manner that made it easy for others to grasp and understand. I thank the people who made their theorem provers freely available to experiment with.

I thank the graduate secretary Diti Anastasopoulos and system administrators from the School of Computer Science of McGill University for their prompt response.

# Table of Contents

# List of Tables

# List of Figures

CHAPTER

# 1

# Introduction

An automated theorem prover (ATP) is a program that attempts to determine if a given theory logically implies a given hypothesis. The range of applications of automated theorem provers has increased significantly [Sutcliffe site] since the first attempt at proving the unsatisfiability of a set of clauses several decades ago [Newell et al. 1957], [Gilmore 1960], [Prawitz & Voghera 1960]. Mathematicians and scientists use ATPs as tools for checking proofs and for proving some open problems [Veroff 1997], [Wos 1993]. Engineers use ATPs for software verification [Fensel & Schönegge 1997], hardware verification [Kaufmann et al. 2000], and database transaction verification [Spelt & Even 1998]. The broader application of ATPs is due to the improved performance of ATPs. However, the progress in automated theorem proving is relatively slow with respect to other fields, such as commercial information technology, as indicated in [Sutcliffe et al. 2001]. In addition, there are still potential domains in business and medicine where ATPs can be used if their performance keeps improving. The performance of an ATP relies on two factors: *speed* and *efficiency*.

In this chapter, we list the major factors that affect the speed and efficiency of an ATP in general, and point out the ones that we focus on in this thesis. We then state the main contributions of our research. Finally, we give an overview of the structure of the thesis.

# 1.1 Speed and Efficiency

In this section we list the major factors that affect the speed and efficiency of an ATP and indicate the ones that we focus on in this thesis.

## 1.1.1 Speed

The **speed** of an ATP is its **inference rate**. The inference rate is basically how fast an ATP is able to deduce facts. This is highly dependent on the following factors:

(1) **Implemented search algorithms, strategies and inference rules.** Some algorithms produce clauses faster than others because they perform less processing on the derived clauses. Also some inference rules are simpler than others and can be implemented with less computer instructions. This leads to a generation of more clauses in less time.

(2) **Data structures.** The basic structures that ATPs work with are terms, literals, clauses, and substitution sets. The internal representation (data structures) of those basic elements can affect the time it takes to execute operations on them.

(3) **Programming language.** A program that is written in C, for example, may run faster than the same program written in Java or LISP. This is because more work has been done on optimizing C compilers. In addition, there are more processor architectures which are better suited for procedural languages like C than functional or logic programming languages like ML and Prolog.

(4) **Code optimization.** Even if two programs that do the same thing, use the same algorithms and data structures, and are written in the same computer language under the same platform, they can differ in the speed of their execution based on the tuning of the code. Also not all compilers of the same language produce the same machine code for the same machine [Wilson 2004], [Kientzle 2004], [Duvanenko 2004] (see Appendix G).

(5) **System hardware and software platform.** It is obvious that more advanced system architectures (e.g., more powerful processor, wider and faster system bus, faster memory, etc.) and operating systems [Bach 1986], [Silberschatz et al. 2001], [McKusick & Neville-Neil 2004], [Stallings 2004] are highly likely[1] to result in a noticeably faster execution of an ATP than older systems.

Even though all of the above factors are important in comparing and analyzing the speed of a theorem prover, our focus in this thesis is on (1) and (2). In fact, it is a well-known issue among computer scientists that (1) and (2) have the most impact on the performance of a system [Garey & Johnson 1979], because (3), (4) and (5) can improve a system's performance only by a constant factor.

### 1.1.2 Efficiency

Before we state the factors on which the efficiency of an ATP relies on, we clarify and differentiate between the three concepts: *search space*, *explorable search space* and *explored search space*.

- The **search space** is set of all clauses that can be derived from a given set of *clauses* (defined in Chapter 2) using a given set of *inference rules* (defined in Chapter 2).
- The **explorable search space** is the set of clauses from the search space that an ATP can derive. The explorable search space is a proper subset of the search space if restriction strategies are used in an ATP to avoid the derivation of certain clauses. Otherwise, the explorable search space is the same as the search space. If derivation of certain clauses is prohibited due

[1] We say it is *highly likely* because there have been at least one case in the past where a newer generation of a 32-bit processor (Intel's Pentium Pro) did not perform better than an older generation of the 32-bit processor (Intel's Pentium) over certain 16-bit optimized applications (Microsoft Windows 95 and earlier) [Rupley & Clyman 1995].

to restriction strategies, then the explorable search space is said to be **pruned**.

- If a set of clauses is *unsatisfiable* (defined in Chapter 2), then the **explored search space** is the set of clauses from the explorable search space that the ATP derives before deriving the empty clause.

The **efficiency** of an ATP relies on the following factors.

- The size of the explorable search space.
- The amount of redundancy (defined in Chapter 2) which is roughly the amount of time wasted deriving clauses that do not contribute any new facts that haven't already been discovered through other clauses.
- The number of unsuccessful attempts to derive a clause. This is the amount of time spent trying to resolve clauses that do not resolve together.

In a sense, efficiency is a measure of the amount of work the theorem prover performs in order to find a solution to a given problem. An efficient algorithm uses one or more strategies to reduce the size of the explorable search space, the amount of redundancy, and the number of failed attempts. In this thesis we focus mainly on reducing the size of the explorable search space.

## 1.2 Contributions

Our contributions are aimed at resolution-refutation[1] ATPs whose main search strategy is depth-first search; henceforth, unless explicitly stated, anytime we mention the words "theorem prover" or "ATP", we imply resolution-refutation ATPs using depth-first search.

A resolution-refutation ATP is an ATP based on the resolution calculus (binary resolution and binary factoring) and seeks a derivation of the empty clause by following some search strategy, e.g., depth-first search.

[1] Resolution-refutation ATP is formally defined in Chapter 2.

The contribution in this thesis is threefold:

- The development of *delayed clause-construction* (DCC), a method that improves the inference rate of an ATP.
- The construction of *attribute sequences* (ATS), a restriction strategy that improves the efficiency of an ATP.
- The introduction of *semi-linear resolution* (SLR), a procedure that combines top-down with bottom-up search approaches.

An implementation was developed to demonstrate the effectiveness of those approaches in improving the speed and efficiency of an ATP.

### 1.2.1 Delayed-clause construction

ATPs based on depth-first search algorithms perform an extensive number of linear derivations and produce a large number of clauses, referred to as intermediate conclusions, that are not goal clauses (defined in Chapter 3) but may lead to goal clauses. The amount of time spent in constructing intermediate conclusions can reach 65% of the total running time (see Chapter 6), based on the results obtained from experiments that we conducted. Upon careful observation, we found that, in a linear derivation, only a small part of an intermediate conclusion needs to be constructed when it is involved in an application of an inference rule. By limiting the construction to the needed part of an intermediate conclusion and delaying the construction of the rest until needed, we can reduce the time to generate a new clause, thereby improving the inference rate.

We developed an approach, delayed clause-construction, that delays the construction of intermediate conclusions until needed. The results obtained from the experiments we performed on DCC demonstrate its potential. Furthermore, DCC requires a modest amount of memory, is easy to implement, and works with a wide range of calculi.

### 1.2.2 Attribute sequences

An attribute is a characteristic of a clause such as weight, length, number of variables, particular term, particular literal, etc. An attribute sequence is a sequence of attributes that correspond to a sequence of resolutions. Even though it is not possible to know ahead of time which clauses to select in order to obtain a refutation (i.e., derivation of the empty clause), it is possible to select potential clauses based on their attributes that may lead to a refutation.

One distinguishing characteristic of ATS is that an attribute sequence can be constructed ahead of time and then used as a guide to select the potential clauses that may lead to a refutation. This implies that no time is wasted during the search to construct an ATS. Another distinguishing characteristic of ATS is that it can be used in any resolution-refutation ATP employing a depth-first search strategy without affecting the completeness of the ATP.

Our analysis indicate that the reduction of the size of the explorable search space is exponential in the depth bound when attribute sequences are used as a guide to select potential clauses. We conducted experiments on the use of ATS and the results show that the improvements are significant.

### 1.2.3 Semi-linear resolution

A top-down approach recursively breaks down a goal into subgoals until eventually the subgoals can be proven immediately by a given set of clauses or by derived clauses obtained during the search process. A bottom-up approach derives clauses from the input set until an inconsistency is reached. The advantage of a top-down approach is that it is goal-oriented. Its disadvantage is lack of redundancy control. A bottom-up approach is good in controlling redundancy but lacks goal-orientation.

Semi-linear resolution is a top-down bottom-up search procedure that includes DCC for speed and ATS for efficiency. DCC in SLR can be viewed as a mega-

inference rule that combines several inference rules into one. Every application of a mega-inference rule leads to a goal clause in one big step.

A proof obtained by SLR is generally non-linear but contains linearly derived goals. Hence the name semi-linear resolution. **Figure 1-1** shows an example of an SLR proof.

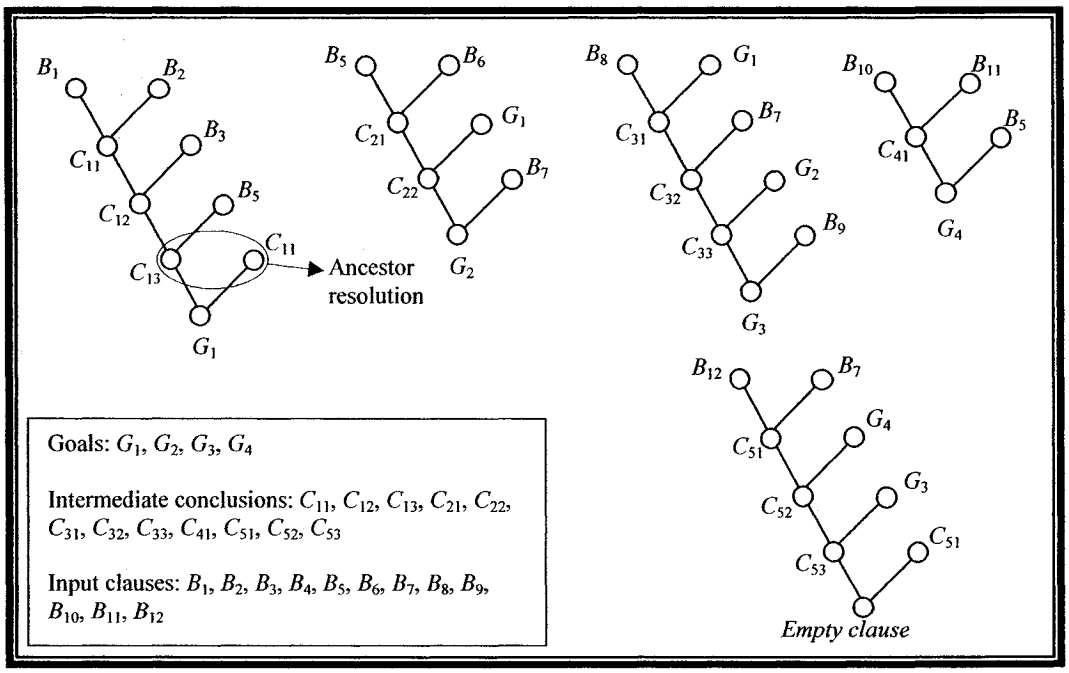



**Figure 1-1: An illustration of a semi-linear resolution proof.**

Sometimes it is useful to gather information during a linear resolution that can help reduce the search space or eliminate redundancy. We show in an example in Chapter 6 that the information gathered from within the application of a mega-inference rule can help an ATP skip certain attribute sequences in following derivations. Just because every application of a mega-inference rule is a big step in a search, it does not mean that useful information obtained from the little steps is ignored. SLR combines both the advantage of moving faster in a search by taking big steps without losing the information that can be obtained from the small steps. This is a distinguishing characteristic of SLR.

SLR can be used in a wide range of applications including systems with limited memory capacities without the need to compromise its refutation completeness. Examples of applications with limited memory capacities are: small embedded systems, microcontrollers, mobile devices, miniature robots, miniature wireless communication devices, etc. The wide range of applications for SLR is due to the combined top-down bottom-up approach where the main loop is an iteratively-deepening depth-first search (IDDFS). IDDFS requires a modest amount of memory which makes it desirable for limited memory devices.

## 1.3 Overview of the Thesis

Chapter 2 covers the basic minimum preliminaries related to the topics discussed within the thesis. It contains all the definitions and notation necessary to understand the terminology and symbols used in the following chapters. Chapter 3 is dedicated to the formal presentation of the delayed clause-construction procedure.

Chapter 4 presents semi-linear resolution. It describes the procedure in detail and compares it with the given-clause algorithm. The advantages and disadvantages of semi-linear resolution are listed and its completeness is discussed.

Chapter 5 analyzes the size of the explorable search space of semi-linear resolution from two perspectives. The first is based on the number of generated clauses and the second is based on the number of attribute sequences. It describes how attribute sequences can be used as a guide to reduce the size of the explorable search space of SLR.

Chapter 6 describes an implementation of semi-linear resolution called CARINE. It presents the experimental results obtained from running CARINE on a sample of 100 theorems selected from the TPTP library (see Appendix A).

Chapter 7 summarizes the contributions, provides concluding remarks and discusses future work.

The appendices include the details on derivations of certain formulas used in the thesis. They also include further information about CARINE, the TPTP library, a list of the sample of 100 theorems selected from the TPTP library, the list of theorems from the TPTP library that were proved by CARINE, and statistical data.

## 1.4 About the Results

Unless specifically stated, all test results are obtained from running our ATP CARINE over the TPTP library with a time limit of 180 seconds per theorem. Experiments were done under a Linux emulation (Cygwin) on a Pentium 4 based machine running Microsoft Windows 2000. The processor's speed is 2.6GHz but we set it to run in Hyper-Threading (HT) mode [Intel site] so that we can run two copies of our system at the same time and thus, reducing the total time needed to obtain the provided results. When running two copies of CARINE in HT mode, the machine roughly acts as two machines running each at about 1.1GHz. The memory installed on the machine is 1 GB DDR1 SDRAM [Rosch 2003] running at 400MHz in Dual Channel which roughly means that in HT mode there would be little or no degradation in speed when running two copies of CARINE since the memory banks can be accessed in parallel.

[1] DDR is a double data rate synchronous dynamic random access memory.

CHAPTER

# 2

# Preliminaries

Theorem provers are applications of mathematical logic. Mathematical logic encompasses many branches of logic and correspondingly there are many kinds of theorem provers. We are only concerned with classical first-order logic [Smullyan 1995] and specifically the subset of this logic that deals with knowledge represented in clause form. The theorem provers that we are interested in are resolution-refutation based automated theorem provers and in this chapter we state the minimum preliminaries that are related to those theorem provers.

## 2.1 Definitions and Conventions

Except for a few minor differences in some definitions (e.g., "derivation"), most of the definitions and notation in this section follow the conventions used in [Riazanov 2003], [Robinson & Voronkov (1) 2001], [Robinson & Voronkov (2) 2001], [Schulz 2000], [Loveland 1978], and [Chang & Lee 1973].

### 2.1.1 Notation

References on the subjects of set theory, logic and automated reasoning, use slightly different notation for the same operators. Also, operator precedence varies between theory and implementation (i.e. computer languages). To avoid any confusion, we use the notation and the order of precedence of operators described in this section.

**Table 2-1** lists the logical and set symbols along with their meaning. Examples are given to clarify the meaning and depending on the meaning of the symbol, the letters $A$ and $B$ used in the examples are either sets or clauses.

**Table 2-1: Set and logical symbols and their meanings**

| Symbol | Meaning |
|---|---|
| $\neg,\wedge,\vee$ | NOT, AND, OR respectively. |
| $\Leftrightarrow$ | Equivalence. E.g. $A \Leftrightarrow B$ |
| $\Rightarrow$ , $\Leftarrow$ | Forward and backward implication. $A \Rightarrow B$ is the same as $B \Leftarrow A$ which is read as $A$ implies $B$ or $B$ logically follows from $A$. |
| $\equiv$ | Identical. |
| $\subset,\supset$ | Right and left proper subset. $B \subset A$ is the same as $A \supset B$ where $B$ is a proper subset of $A$. |
| $\subseteq,\supseteq$ | Right and left subset. $B \subseteq A$ is the same as $A \supseteq B$ where $B$ is a subset of $A$. |
| $\cap,\cup,\setminus$ | Set operations: intersection, union, and difference respectively. |
| $\prec$ | An ordering relation. Read as "less than" even though the domain may not be a number. For example, "abc" $\prec$ "bdf" means that the string "abc" is less than the string "bdf" when the ordering relation represents a lexicographical[1] ordering. |
| $\mathbb{N},\mathbb{Z},\mathbb{R}$ | Respectively: the set of natural numbers, the set of integers, the set of real numbers. |
| $*$ | Wildcard character. Used as a "don't care" or "all values in a domain". For example, $(*,2)$ means a pair where the first element is any number in the domain and the second element is a 2. |

## 2.1.2 Operator precedence

1) Expressions within parentheses () are evaluated first followed by the ones within square brackets [] followed by those within braces {}.
2) Logical operators are performed in the order (highest to lowest): $\neg,\wedge,\vee$.

[1] Lexicographical ordering is an ordering similar to the ordering of the words in a dictionary.

3) Set operators are performed in the order (highest to lowest): $\cap, \cup, \setminus, \in, \subset, \subseteq, =$.

4) Identical: $\equiv$.

5) Implication and equivalence in the order (highest to lowest): $\Rightarrow, \Leftrightarrow$.

### 2.1.3 Definitions

**Definition 2.1: Multiset**

A **multiset** $M$ over a set $S$ is a function $M : S \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. An element $x$ in $M$ is denoted by $x \in M$. If $x \in M$ then $M(x) > 0$, otherwise $M(x) = 0$. In other words, $M(x)$ specifies the number of occurrences of $x$ in $M$. For example, if $M = \{a, a, b, c\}$ then $M(a) = 2$, $M(b) = 1$, $M(c) = 1$. A multiset $M$ over a set $S$ is **finite** if $M(x) > 0$ for a finite number of $x \in S$. The **set of distinct elements** of a multiset $M$ is denoted by $Set(M)$, and defined as $Set(M) = \{x : M(x) > 0\}$.

The set operations are extended to multisets as follows.

(i) Existence:

$$x \in M \Leftrightarrow M(x) > 0.$$

$$x \notin M \Leftrightarrow M(x) = 0.$$

(ii) Cardinality:

$$|M| = \sum_{x \in S} M(x).$$

(iii) Emptiness:

$$M = \{\} \Leftrightarrow |M| = 0.$$

Suppose $M_1$ and $M_2$ are multisets over a set $S$.

(iv) Proper submultiset:

$$M_1 \subset M_2 \Leftrightarrow \forall x \in S : M_2(x) \geq M_1(x) \text{ and } \exists y \in S : M_2(y) > M_1(y).$$

(v) Submultiset:

$$M_1 \subseteq M_2 \Leftrightarrow \forall x \in S : M_2(x) \geq M_1(x).$$

(vi) Union:

$$M_1 \cup M_2 \Leftrightarrow \forall x \in S : (M_1 \cup M_2)(x) = M_1(x) + M_2(x).$$

(vii) Intersection:

$$M_1 \cap M_2 \Leftrightarrow \forall x \in S : (M_1 \cap M_2)(x) = Min(M_1(x), M_2(x)).$$

(viii) Difference:

$$M_1 \setminus M_2 \Leftrightarrow \forall x \in S : (M_1 \setminus M_2)(x) = Max(M_1(x) - M_2(x), 0).$$

In set theory, given the sets $S = \{A_1, \ldots, A_n\}$ and $T = \{B_1, \ldots, B_m\}$, with $n, m \geq 1$, the set $(S \setminus \{A_1, \ldots, A_i\}) \cup (T \setminus \{B_1, \ldots, B_j\})$, where $1 \leq i \leq n$ and $1 \leq j \leq m$, is equal to the set $(S \cup T) \setminus \{A_1, \ldots, A_i, B_1, \ldots, B_j\}$ [Borowski & Borwein 1991]. We call it the **special DU law**[1]. This law also applies to multisets.

**Definition 2.2: List and sequence**

A **list** is a countable (possibly infinite) ordered multiset. A list is denoted by $\ell = \langle \beta_1, \cdots, \beta_n \rangle$, where $\beta_1, \cdots, \beta_n$ are its elements. An empty list is denoted by $\langle\ \rangle$. A **sequence** is a list where each element is computed based on previous elements in the sequence. The **length of a list** is the number of elements in the list and is denoted by $|\ell|$, where $\ell$ is a list. For example, if $\ell = \langle \beta_1, \cdots, \beta_n \rangle$ then $|\ell| = n$.

**Definition 2.3: Term**

A **term** is a variable, or an $n$-ary function of the form $f(t_1, \ldots, t_n)$, where $n \geq 0$ and $t_1, \ldots, t_n$ are terms. The terms $t_1, \ldots, t_n$ are the arguments of the function and $f$

[1] DU stands for difference-union law. We call it *special* because in general, if $A, B, C, D$ are sets, then $(A \setminus B) \cup (C \setminus D) \neq (A \cup C) \setminus (B \cup D)$. E.g. $A = \{1,2\}$, $B = \{1,4\}$, $C = \{4,6\}$, $D = \{5\}$. $(A \setminus B) \cup (C \setminus D) = \{2,4,6\}$, $(A \cup C) \setminus (B \cup D) = \{2,6\}$. However, in the special case where $B \subseteq A$ and $D \subseteq C$, DU law is true.

is the function symbol. A **constant** is a function with arity 0, i.e., without any arguments. The countably infinite set of variables is denoted by $\mathcal{V}$. The (finite or countably infinite) set of function symbols is denoted by $\mathcal{F}$. The set of all terms that can be formed from $\mathcal{V}$ and $\mathcal{F}$ is denoted by $\mathcal{T}(\mathcal{V},\mathcal{F})$. A formal and detailed explanation of the term algebra $\mathcal{T}(\mathcal{V},\mathcal{F})$ is given in [Gallier 1986]. A **ground term** is a term with no variables.

Following the conventions used in many textbooks such as [Newborn 2001], [Sekar et al. 2001], [Bibel 1987], [Gallier 1986], [Loveland 1978], [Chang & Lee 1973], we use (possibly with subscripts) the letters $u,v,w,x,y,z$ to denote variables, $a,b,c$ to denote constants, and $f,g,h$ to denote functions with arity greater than zero.

A term is drawn as a tree. A constant or a variable occupies a single node. A function with one or more arguments is drawn as a tree rooted at the function symbol with its children being the arguments of the function. A tree representation of the term $f(a,g(a,y),x)$ is shown in **Figure 2-1**.

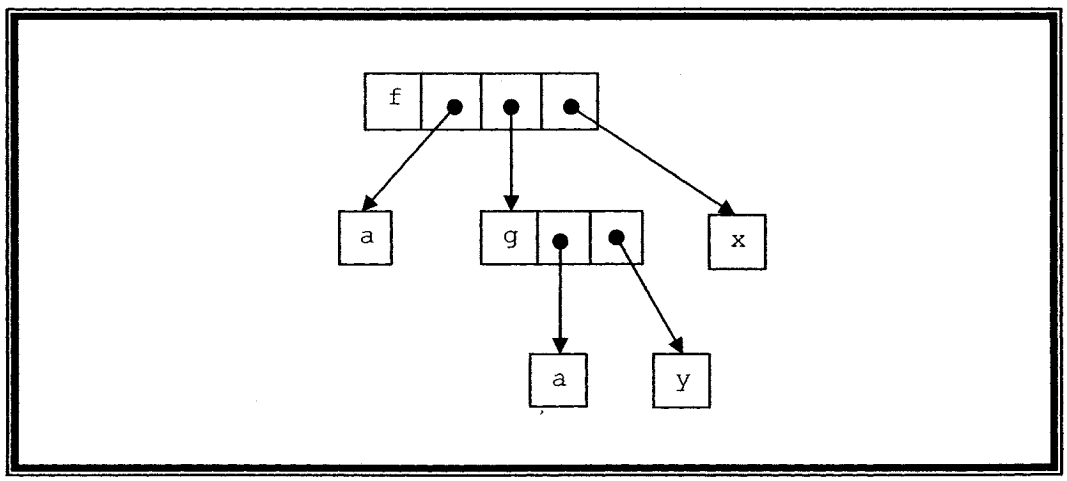



**Figure 2-1: A tree representation of the term $f(a,g(a,y),x)$.**

**Definition 2.4: Subterm**

A term $s$ is a **subterm** of a term $t$, denoted[1] by $s \in^o t$, if $s = t$ or $s$ occurs in $t$. Therefore, a term is a subterm of itself. $s \notin^o t$ denotes $s$ is not a subterm of $t$.

**Definition 2.5: Multiset of arguments**

The **multiset of arguments** of a function term $t$ is denoted by $Args(t)$. For example, $Args(f(a,a,x,g(b))) = \{a,a,x,g(b)\}$ and $Args(a) = \{\}$.

**Definition 2.6: Term symbol**

The **term symbol** of a variable is the variable itself. The **term symbol** of a function is the function symbol.

**Definition 2.7: Weight of a term**

The **weight of a term** $t$, denoted by $Weight(t)$, is the number of term symbols within it. When a term is represented as a tree, the weight of the term is the number of nodes in the tree. The function $Weight(t)$ is computed recursively as

$$Weight(t) = \begin{cases} 1 & \text{if } t \text{ is a variable,} \\ 1 + \sum_{i=1}^{|Args(t)|} Weight(s_i), \text{ where } s_i \in Args(t) & \text{if } t \text{ is a function.} \end{cases}$$

**Example 2.1:**

$Weight(a) = 1$.

$Weight(g(f(x,y),g(a,x)) = 7$.

[1] We add the $o$ (i.e., occurrence) to the symbol $\in$ to differentiate it from the membership relation used on sets and multisets.

**Definition 2.8: Maximum term depth**

The **maximum term depth** of a term $t$, denoted by $MaxDepth(t)$, is defined recursively as

$$MaxDepth(t) = \begin{cases} 1 & \text{if } t \text{ is a constant or a variable,} \\ 1 + \underset{s \in Args(t)}{Max} \{MaxDepth(s)\} & \text{if } t \text{ is a function.} \end{cases}$$

**Example 2.2:**

$MaxDepth(f(x)) = 2$.

$MaxDepth(f(g(x,a,y),b)) = 3$.

$MaxDepth(f(f(a,g(c,y)),f(a,x))) = 4$.

**Definition 2.9: Position**

A **position** is either the empty string $\varepsilon$, or a string of the form $i.\pi'$, where $i \in \mathbb{N}$ and $\pi'$ is a position. The subterm at position $\pi$ of a term $t$ is denoted by $t|_\pi$. If $t$ is a variable or a constant, then there is only one valid position $\pi$ in $t$, and that is $\pi = \varepsilon$. If $t = f(t_1,\ldots,t_n)$, where $n \geq 1$, then

$$t|_\pi = \begin{cases} t & \text{if } \pi = \varepsilon, \\ t_i|_{\pi'} & \text{if } \pi = i.\pi', \text{ where } 1 \leq i \leq n. \end{cases}$$

For example, if $t = f(g(a,b,c,h(x)),y)$ and $\pi = 1.4.1$ then $t|_\pi = x$. A position $\pi$ is **invalid** with respect to a term $t$ if there is no subterm in $t$ at position $\pi$. The result is the empty string $\varepsilon$. For example, if $t = f(a)$ and $\pi = 1.2$, then $\pi$ is invalid with respect to $t$ and $t|_{1.2} = \varepsilon$.

**Definition 2.10: Term replacement**

If $q \in^{o} t$ then the term obtained by replacing all occurrences of $q$ in $t$ with the term $s$ is denoted by $t[q \to s]$. If the subterm $q$ at position $\pi$ in $t$ is replaced by $s$, then the resulting term is denoted by $t[q \to s]_{\pi}$. If the subterm at position $\pi$ is not important, then $t[s]_{\pi}$ denotes the term obtained by replacing whatever subterm in $t$ at position $\pi$ with $s$.

**Definition 2.11: Atom**

An **atom** is an $n$-ary predicate of the form $P(t_1,\ldots,t_n)$, where $n \geq 0$ and $t_1,\ldots,t_n$ are terms. The terms $t_1,\ldots,t_n$ are the arguments of the predicate and $P$ is the predicate symbol. If $n = 0$ then the atom is a propositional constant.

**Definition 2.12: Literal**

A **literal** is an atom or its negation. The negation of an atom is represented as an atom preceded by the negation sign $\neg$. A **positive literal** is an atom and a **negative literal** is a negated atom. An atom with an **equality** predicate is written as $l \simeq r$, where $l$ and $r$ are terms. Its negation, $\neg(l \simeq r)$, is written as $l \not\simeq r$. A **ground literal** is a literal that contains no variables. The multiset of arguments of a literal $L$ is denoted by $Args(L)$.

**Definition 2.13: Weight of a literal**

The **weight of a literal** $L$, denoted by $Weight(L)$, is the sum of the weights of its arguments plus one, i.e.,

$$Weight(L) = 1 + \sum_{i=1}^{|Args(L)|} Weight(s_i), \qquad \text{where } s_i \in Args(L).$$

**Definition 2.14: Maximum literal depth**

The **maximum literal depth** of a literal $L$, denoted by $MaxDepth(L)$, is the maximum depth of any of its arguments:

$$MaxDepth(L) = \begin{cases} 0 & \text{if } |Args(L)| = 0, \\ \underset{s \in Args(L)}{Max} \{MaxDepth(s)\} & \text{if } |Args(L)| > 0. \end{cases}$$

**Example 2.3:**

$MaxDepth(\neg P) = 0$.

$MaxDepth(P(f(x), g(f(a)), y) = 3$.

**Definition 2.15: Clause**

A **clause** is a disjunction of literals, $L_1 \vee \cdots \vee L_n$ (logical representation), or a finite multiset of literals, $\{L_1, \cdots, L_n\}$ (multiset representation). We use both representations depending on the context.

**Definition 2.16: Special clauses**

A **positive clause** is a clause whose literals are all positive literals.

A **negative clause** is a clause whose literals are all negative literals.

The **empty clause** is a clause that has no literals. It is denoted by $\phi$.

A **unit clause** is a clause with one literal.

A **Horn clause** is a clause with at most one positive literal.

An **equation** is a unit clause whose only literal is a positive equality literal.

A **disequation** is a unit clause whose only literal is a negative equality literal.

A **ground clause** is a clause with no variables in any of its literals.

A **propositional clause** is a clause where the atoms of its literals are propositional constants.

An **ordered clause** is a clause whose literals are ordered according to some ordering relation. Therefore, an ordered clause is a list.

**Definition 2.17: Normalized clause**

A **normalized clause** is an ordered clause whose variables follow a certain naming convention. In this thesis we assume the following variable naming convention. In examples where the number of distinct variables in the clauses is 6 or less, we use the naming order $x, y, z, u, v, w$. If the number of distinct variables is bigger than 6, we use the naming order $x_1, x_2, x_3, \ldots$.

**Definition 2.18: Length and weight of a clause**

The **length of a clause** $C$, denoted by $Len(C)$, is the number of literals in it. Since $C$ is a multiset of literals then $Len(C) = |C|$. The **weight of a clause**, denoted by $Weight(C)$, is the sum of the weights of all its literals;

$$Weight(C) = \sum_{i=1}^{|C|} Weight(L_i), \quad \text{where } L_i \in C.$$

A clause is said to be **too long** if its length is greater than a limit that is either set by the user or automatically chosen by the ATP. Similarly, a clause is said to be **too heavy** if its weight is greater than a specified limit imposed by the ATP or set by the user.

**Example 2.4:**

$$Weight(\neg P \vee Q(x, f(y))) = Weight(\neg P) + Weight(Q(x, f(y))) = 1 + 4 = 5.$$

$$\begin{aligned} Weight(&\neg Q(x) \vee \neg Q(x) \vee P(x, y)) \\ &= Weight(\neg Q(x) + Weight(\neg Q(x) + Weight(P(x, y) \\ &= 2 + 2 + 3 \\ &= 7. \end{aligned}$$

$$Size(P \vee Q) = 2.$$

$$Size(P(a) \vee Q(x, y) \vee Q(x, y)) = 3.$$

$$Size(\phi) = 0.$$

**Definition 2.19: Clause attribute**

A **clause attribute**[1] is a characteristic of a clause. It can be a term, literal, a Boolean value, a real value, an integer value, etc. The set of all attributes of a clause $C$ is denoted by $\mathcal{A}(C)$ and the subset of $\mathcal{A}(C)$ where the attributes are real numbers is denoted by $\mathcal{A}_{\mathbb{R}}(C)$.

Some clause attributes are the following.

- The weight of a clause.
- The length of a clause.
- The number of distinct variables in a clause.
- The number of function symbols in a clause.
- The maximum depth of any literal in a clause.
- The number of positive literals in a clause.
- The number of negative literals in a clause.
- A term in a clause.
- A literal in a clause.
- A position in a literal in a clause.

[1] A clause attribute is similar to a clause *feature* defined in [Chang & Lee 1973] and [Schulz 2000] but more general. A feature is a number. An attribute can be a number or a Boolean, term, literal, etc.

- The depth at which a clause was generated in a linear derivation (see Definition 2.31).
- The existence of an equality literal.
- The ratio of the number of distinct variables to the total number of variables in a clause.

**Definition 2.20: Interpretation, model, satisfiability, tautology**

An **interpretation** $I$ of a set of clauses $S$, $|S| \geq 1$, consists of a non-empty domain $D$, and it gives meaning to constants, functions and predicates by relating them to $D$ as follows.

- Each constant is assigned an element from $D$.
- Each $n$-ary $(n > 0)$ function symbol is assigned a mapping from $D^n$ to $D$.
- Each proposition is assigned a value from the set $\{false, true\}$.
- Each $n$-ary $(n > 0)$ predicate symbol is assigned a mapping from $D^n$ to the set $\{false, true\}$.

If there is an interpretation $I$ that makes a set of clauses $S$ true, then $S$ is **satisfiable** or **consistent**, and the interpretation $I$ is a **model** of $S$. A clause that is satisfied by all interpretations is a **tautology**. If no interpretation makes $S$ true, then $S$ is **unsatisfiable** or **inconsistent**.

**Definition 2.21: Substitution sets**

A **substitution** set, denoted by one of the Greek symbols $\sigma, \theta, \mu$, represents a mapping of variables to terms. A finite substitution set has the form $\sigma = \{v_1 \to t_1, \ldots, v_n \to t_n\}$, where $n \geq 0$, $v_1, \ldots, v_n$ are variables, and $t_1, \ldots, t_n$ are substitution terms. If $n = 0$ then $\sigma = \{\}$ is the empty substitution. The set $Dom(\sigma) = \{v_1, \ldots, v_n\}$ is the domain of $\sigma$ and the set $Ran(\sigma) = Set(\{t_1, \ldots, t_n\})$ is

the range of $\sigma$. The **size of a substitution** set $\sigma$ is $|\sigma| = |Dom(\sigma)|$. A **variable renaming substitution set** is a substitution set where all the substitution terms are variables.

**Definition 2.22: Composition of substitution sets**

Given the substitution sets $\sigma = \{v_1 \to t_1, \ldots, v_n \to t_n, u_1 \to t_{n+1}, \ldots, u_i \to t_{n+i}\}$ and $\theta = \{u_1 \to s_1, \ldots, u_m \to s_m\}$, such that the variables $v_1, \ldots, v_n, u_1, \ldots, u_m$ are all distinct, and $0 \le i \le m$ where $i = 0$ means that $\sigma = \{v_1 \to t_1, \ldots, v_n \to t_n\}$, then the composition $\sigma\theta$ as defined in [Loveland 1978] is

$$\sigma\theta = \{v_1 \to t_1\theta, \ldots, v_n \to t_n\theta, u_1 \to t_{n+1}\theta, \ldots, u_i \to t_{n+i}\theta, u_{i+1} \to s_{i+1}, \ldots, u_m \to s_m\}.$$

**Definition 2.23: Idempotent substitution set**

A substitution set $\sigma = \{v_1 \to t_1, \ldots, v_n \to t_n\}$, where $n \ge 1$, is **idempotent** if for all $1 \le i, j \le n$, $v_j \notin^o t_i$. In other words, a substitution set is idempotent when the substitution terms in a substitution set contain no variables belonging to the domain. In [Bibel 87] the author defines an idempotent set follows: if $\sigma\sigma = \sigma$ then $\sigma$ is idempotent.

It is easy to conclude from the definition of the composition of substitution sets that the composition of two or more idempotent substitution sets is idempotent.

If $\sigma, \theta, \mu$ are three idempotent substitution sets then the composition $\sigma\theta\mu$ is associative $\sigma\theta\mu = (\sigma\theta)\mu = \sigma(\theta\mu)$ [Loveland 1978].

**Definition 2.24: Circular substitution**

A substitution set $\sigma = \{v_1 \to t_1, \ldots, v_n \to t_n\}$, where $n \ge 1$, is said to contain a **circular substitution** if there exits a subset $\{v_{i_1} \to t_{i_1}, \ldots, v_{i_k} \to t_{i_k}\} \subseteq \sigma$, where

$1 \le k \le n$, for all $1 \le j \le k$, $1 \le i_j \le n$, such that $v_{i_2} \in^o t_{i_1}, \ldots, v_{i_k} \in^o t_{i_{k-1}}, v_{i_1} \in^o t_{i_k}$ and at least one of the $t_{i_1}, \ldots, t_{i_k}$ is a function with arity greater than zero. For example, $\sigma = \{x \to f(x)\}$ and $\sigma = \{x \to y, y \to f(z), z \to x, w \to a\}$ contain circular substitutions.

**Definition 2.25: Application of substitution set, instance, and variant**

Let $\Lambda$ be one of the following: a term, a literal, a clause, a set of clauses or a substitution set. Applying a substitution $\sigma$ to $\Lambda$ means that the variables in $\Lambda$ are replaced by the corresponding substitution terms from $\sigma$. This is denoted by $\Lambda\sigma$. An **instance** of $\Lambda$ is obtained when some substitution set is applied to it. $\Lambda'$ is called a **variant** of $\Lambda$ if $\Lambda' = \Lambda\sigma$ and $\sigma$ is a variable renaming substitution. $Vars(\Lambda)$ denotes the set of all distinct variables in $\Lambda$. If $\sigma$ is a substitution set, then $Dom_{\Lambda}(\sigma) = Vars(\Lambda) \cap Dom(\sigma)$.

**Definition 2.26: Unifier, most general unifier and unification**

If $t_1$ and $t_2$ are terms then a substitution set $\sigma$ is a **unifier** if $t_1\sigma = t_2\sigma$. Similarly, if $L_1$ and $L_2$ are literals then a substitution set $\sigma$ is a unifier if $L_1\sigma = L_2\sigma$. In other words, a unifier is a substitution set that when applied to two terms or literals makes them identical. The definition of a unifier can be extended to any number of terms or literals.

The **most general unifier** (mgu[1]) is the unifier having the least number of substitutions and still makes two or more terms or literals equal. Formally, if $\sigma$ is the most general unifier of two or more terms or literals, then for every other unifier $\theta$ of these two or more terms or literals, there exist a substitution set $\mu$ such that $\theta = \sigma\mu$. If $t_1, \ldots, t_n$, $n \ge 2$, are terms and $\sigma$ is the mgu of those terms,

[1] We write the plural of mgu as *mgu's* rather than *mgus*. This is suggested by English professors and technical writing experts.

then we write $\sigma = mgu(t_1,\ldots,t_n)$, i.e., $t_1\sigma = \cdots = t_n\sigma$. Similarly, if $L_1,\ldots,L_n$, $n \geq 2$, are literals and $\sigma$ is the mgu of those literals, then we write $\sigma = mgu(L_1,\ldots,L_n)$, i.e., $L_1\sigma = \cdots = L_n\sigma$. **Unification** is the process of finding an mgu of two or more terms or literals.

**Definition 2.27: Complementary literals**

Two literals $L_1$ and $L_2$ are said to be **potentially complementary literals** if there exist an mgu $\sigma$, such that $L_1\sigma = \neg L_2\sigma$. Two literals $L_1$ and $L_2$ are **complementary literals** if $L_1 = \neg L_2$.

**Definition 2.28: Theorem**

Given a set of clauses $S = \{C_1,\ldots,C_n\}$ and a clause $G$, $G$ is a **logical consequence** of $S$ (or $S$ **entails** $G$) if and only if every interpretation that is a model of $S$ is also a model of $G$. If $G$ is a logical consequence of $S$, then $C_1 \wedge \cdots \wedge C_n \Rightarrow G$ is a **theorem**, $S$ is the set of **axioms**, and $G$ is the **conclusion of the theorem**. To prove a theorem is to show that $G$ is a logical consequence of $S$. Since we are concerned only with refutational theorem provers, then to prove a theorem is to show that $C_1 \wedge \cdots \wedge C_n \wedge \neg G \Rightarrow \phi$, i.e., $C_1 \wedge \cdots \wedge C_n \wedge \neg G$ has no model.

**Definition 2.29: Inference rule and inference system**

An **inference rule** is an $n+1$-ary relation on clauses written as

$$\frac{C_1 \cdots C_n}{C} \text{ if } \gamma,$$

where $C_1 \cdots C_n$ are the **premises**, $C$ is the **conclusion**, and $\gamma$ is a set of **conditions**. An inference rule is **sound** if and only if the conclusion is logically implied by the premises. We are only interested in sound inference rules.

Henceforth, unless explicitly stated otherwise, the use of the words "inference rule" imply "sound inference rule". An **inference**, denoted by $\mathcal{I}$, is an instance (an application) of an inference rule. The multiset of the clauses used as premises in an inference $\mathcal{I}$ is denoted by $Prem(\mathcal{I})$ and the conclusion by $C(\mathcal{I})$. An **inference system** is a set of inference rules.

**Definition 2.30: Deduction, derivation, refutation, proof**

A **deduction** of a clause $D$ from a given set of clauses $S$ is a sequence $\langle C_1,\ldots,C_n\rangle$, where $n>0$ and for all, $C_i$ is a logical consequence of $S\cup\{C_1,\ldots,C_{i-1}\}$, and $D=C_n$. A **derivation**[1] of a clause $D$ from a given set $S$ of clauses is a sequence of inferences $\langle \mathcal{I}_1,\ldots,\mathcal{I}_n\rangle$, where $n>0$ and for all $1\le i\le n$, each clause in $Prem(\mathcal{I}_i)$, is either in $S$ or is a logical consequence of $S\cup\{C(\mathcal{I}_1),\ldots,C(\mathcal{I}_{i-1})\}$, and $D=C(\mathcal{I}_n)$. $D$ is referred to as a **derived clause**. The clauses in $S$ are referred to as **input clauses**. A **refutation** is a derivation of the empty clause. In the context of resolution-refutation, a **proof** is a refutation.

A derivation of a clause is graphically represented by a tree. **Figure 2-2** is an example of a derivation of a clause $D$ from a set $S=\{C_1,C_2,C_3,C_4,C_5,C_6\}$. The root node is $D$, all internal nodes are derived clauses, and the leaves (heavily marked) are input clauses.

[1] The words deduction and derivation are used interchangeably in many references and the standard definition is the one we use for "deduction". However, we define "derivation" differently from deduction for the purpose of simplifying our presentation of the topics in this thesis. Our definition of derivation explicitly states the instances of the rules used in a deduction. Therefore, each $\mathcal{I}_i$ appearing in the sequence $\langle \mathcal{I}_1,\ldots,\mathcal{I}_n\rangle$ represents the premises, the conclusion, the mgu, etc. Whereas, the sequence $\langle C_1,\ldots,C_n\rangle$, does not imply the additional information we need in order to present our ideas (such as delayed clause-construction, etc.) in a simple way.

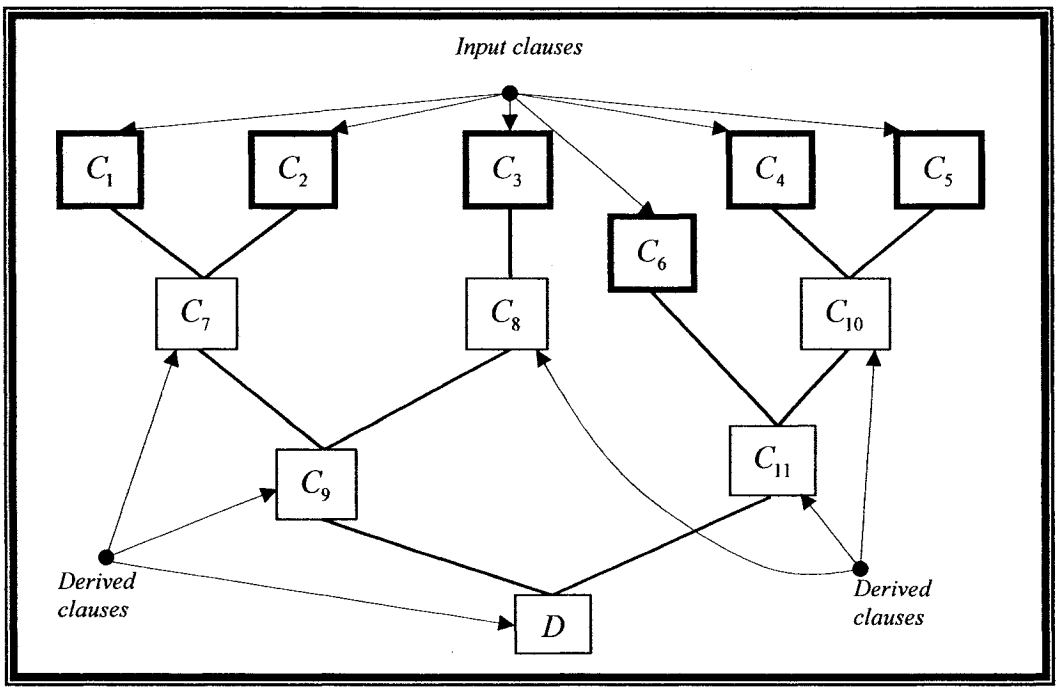



**Figure 2-2: A graphical representation of a derivation.**

**Definition 2.31: Linear derivation**

A **linear derivation** of a clause $D$ from a given set of clauses $S$ is a derivation $\langle \mathcal{I}_1,\ldots,\mathcal{I}_n \rangle$, with the following properties:

- One of the premises of $\mathcal{I}_1$ is called the **initial clause** and is denoted by $C_{init}$.
- For all $2 \le i \le n$, one clause of $Prem(\mathcal{I}_i)$, called the **main premise**, is $C(\mathcal{I}_{i-1})$. The clauses $C(\mathcal{I}_1),\ldots,C(\mathcal{I}_{n-1})$ are **intermediate conclusions** derived, respectively, at depths $1,\ldots,n-1$.
- For all $2 \le i \le n$, the clauses $C_{init}, C(\mathcal{I}_1),\ldots,C(\mathcal{I}_{i-1})$ are called **ancestors** of $C(\mathcal{I}_i)$.
- $D = C(\mathcal{I}_n)$ is the **final conclusion**.

- For all $1 \le i \le n$, all the premises of $\mathcal{I}_i$ except for the initial clause and the main premise are called **side premises**. The multiset of side premises is denoted by $\mathcal{D}(\mathcal{I}_i)$ and is determined as follows:
- $\mathcal{D}(\mathcal{I}_i) = \begin{cases} Prem(\mathcal{I}_1) \setminus C_{init} & i = 1, \\ Prem(\mathcal{I}_i) \setminus C(\mathcal{I}_{i-1}) & 2 \le i \le n. \end{cases}$
- For all $1 \le i \le n$, every clause in $\mathcal{D}(\mathcal{I}_i)$ is either a variant of a clause from $S$ or a variant of an ancestor clause.
- If $C(\mathcal{I}_i) \in \mathcal{D}(\mathcal{I}_j)$, where $1 \le i \le n-2$ and $j > i+1$, then $C(\mathcal{I}_i)$ is called a **far parent** of $C(\mathcal{I}_j)$.

**Definition 2.32: Input derivation**

An (linear) **input derivation** of a clause $D$ from a given set of clauses $S$ is a linear derivation $\langle \mathcal{I}_1, \ldots, \mathcal{I}_n \rangle$ in which every side premise is a variant of an input clause, i.e., for all $1 \le i \le n$, if $C \in \mathcal{D}(\mathcal{I}_i)$ then $C\theta \in S$, where $\theta$ is a variable renaming substitution.

**Definition 2.33: Completeness**

An inference system $I$ is **complete** if and only if given a set of clauses $S$, any clause that is a logical consequence of $S$ can be derived from $S$ using the inference rules in $I$.

An inference system $I$ is **refutation complete** if and only if given any unsatisfiable set of clauses $S$, it can shown using the inference rules in $I$ that $S$ has no model. In other words, an inference system $I$ is refutation complete if and only if the empty clause can be derived from any unsatisfiable set of clauses by applying the inference rules in $I$.

**Definition 2.34: Redundancy**

A clause $C$ is said to be **redundant** in a set of clauses $S$ if and only if there exist a subset $T$ of $S$ where each clause in $T$ is shorter than $C$ and $T$ entails $C$.

**Definition 2.35: Resolution-based ATP**

A **resolution-based ATP** is an ATP that is based on the resolution calculus formed from binary resolution and binary factoring (see section 2.1.4 Inference rules). A resolution-based ATP may include other inference rules.

**Definition 2.36: State of completeness**

We say, **an ATP maintains its state of completeness after it is subjected to a number of modifications**, to mean that if the ATP is refutation complete then it remains refutation complete and if the ATP is not refutation complete then it may or may not become refutation complete but it would still be able to prove all the theorems that it used to prove before the modifications have been made.

**Definition 2.37: Inference rate**

The inference rate of an ATP is the number of inferences performed in a unit of time. The unit of time is usually a second.

### 2.1.4 Inference rules

A list of the inference rules that are relevant to our work are listed below. In order to simplify the presentation of the mega-inference in Chapter 3, we express these rules in a manner that is slightly different from the conventional representation found in [Riazanov 2003], [Robinson & Voronkov (1) 2001], [Robinson & Voronkov (2) 2001], [Schulz 2000], [Loveland 1978], and [Chang & Lee 1973]. The differences in the representation can be summarized as follows.

- We use the multiset representation of clauses rather than the disjunction of literals.
- We make use of the special DU law to express the conclusions of the inference rules below as a multiset difference between clauses and their literals.
- We do not include details (e.g., equality literals, terms at particular positions, ...) about the contents of the clauses when listing the premises. The details are listed in the conditions of the inference rule (beside or below the rule).

In any inference rule stated below, the variables between the premises are not shared. In other words, in an inference $\mathcal{I}$, if $C \in Prem(\mathcal{I})$ and $D \in Prem(\mathcal{I})$ then $Vars(C) \cap Vars(D) = \{\}$.

**BINARY RESOLUTION:**

| | |
|---|---|
| $\dfrac{C_1 \quad C_2}{((C_1 \cup C_2) \setminus \{A, \neg B\})\sigma}$ | **if** $A \in C_1$ and $\neg B \in C_2$ **and** $\sigma = mgu(A, B)$. |

The conclusion is called a **resolvent** and the clauses $C_1$ and $C_2$ are its **parents**. The positive literal $A$ and the negative literal $\neg B$ are said to be **resolved away** or **resolved upon**.

**BINARY FACTORING:**

| | |
|---|---|
| $\dfrac{C}{(C \setminus \{L_2\})\sigma}$ | **if** $\{L_1, L_2\} \subseteq C$ **and** $\sigma = mgu(L_1, L_2)$. |

The conclusion is called a **factor**. $L_2$ is the **factored out** literal. If binary factoring is performed on a derived clause and $\sigma = \{\}$, then the conclusion is called a **merge clause**.

**HYPERRESOLUTION:**

| $$\frac{C_1 \cdots C_n}{((C_1 \cup \cdots \cup C_n) \setminus \{A_1, \ldots, A_{n-1}, \neg B_1, \ldots, \neg B_{n-1}\})\sigma}$$ |
|---|
| **if** $n \geq 2$ **and**<br>$C_1, \ldots, C_{n-1}$ are positive clauses **and**<br>$C_n$ contains $n-1$ negative literals (the rest are positive literals) **and**<br>For all $1 \leq i \leq n-1$, $A_i \in C_i$, $\neg B_i \in C_n$, and $\sigma_i = mgu(A_i, B_i)$ **and**<br>For all $1 \leq i, j \leq n-1$, if $i \neq j$ then $Dom(\sigma_i) \cap Dom(\sigma_j) = \{\}$ **and**<br>$\sigma = \sigma_1 \cup \cdots \cup \sigma_{n-1}$. |

The conclusion is called a **hyperresolvent**. The clauses $C_1, \ldots, C_{n-1}$ are called **satellites**, and $C_n$ is called the **nucleus**.

**Negative hyperresolution** is similar to hyperresolution. Instead of $n-1$ negative literals in the nucleus, there are $n-1$ positive literals, and the satellites are negative clauses.

**PARAMODULATION:**

| | |
|---|---|
| $$\frac{C' \cup \{L\} \qquad D}{((C' \cup \{L[t \to r]_\pi\} \cup D) \setminus \{l \simeq r\})\sigma}$$ | **if** $L\|_\pi = t$ **and**<br>$(l \simeq r) \in D$ **and**<br>$\sigma = mgu(t, l)$. |

The conclusion is called a **paramodulant**. $D$ is called a **paramodulator** or **from clause**. $C' \cup \{L\}$ is called a **paramodulated clause** or **into clause**.

Paramodulation has gone through many refinements since its introduction in [Robinson & Wos 1969-1] and [Robinson & Wos 1969-2]. A summary of the successful contributions made throughout the past several decades on paramodulation and equality reasoning in general can be found in [Nieuwenhuis & Rubio 2001] and [Degtyarev & Voronkov 2001].

The refinements to paramodulation add restrictions in the set of conditions. The versions of paramodulation that include additional conditions to the ones stated above are commonly referred to as **restricted paramodulations**, whereas the above stated rule is referred to as **unrestricted paramodulation**. In this thesis, unless the word "restricted" or "unrestricted" is specifically stated, the use of the word "paramodulation" refers to any kind of paramodulation; whether it is restricted or unrestricted. One commonly used restricted paramodulation in modern theorem provers, such as Vampire [Riazanov 2003] and E [Schulz 2002], is *superposition.*

Since superposition is a restricted paramodulation, then the terminology, *from clause* and *into clause*, apply to superposition as well.

**SUPERPOSITION INTO NON-EQUALITY LITERAL:**

| | |
|---|---|
| $\dfrac{C' \cup \{L\} \qquad D}{((C' \cup \{L[t \to r]_\pi\} \cup D) \setminus \{l \simeq r\})\sigma}$ | **if** $L\|_\pi = t$ **and** $(l \simeq r) \in D$ **and** $\sigma = mgu(t,l)$ **and** $t \notin \mathcal{V}$ **and** $l\sigma \not\prec r\sigma$. |

**SUPERPOSITION INTO A POSITIVE EQUALITY LITERAL:**

| | |
|---|---|
| $\dfrac{C' \cup \{L\} \qquad D}{((C' \cup \{L[q \to r]_\pi\} \cup D) \setminus \{l \simeq r\})\sigma}$ | **if** $L \equiv (s \simeq t)$ **and**<br>$q \in^o s$ **and**<br>$L\|_\pi = q$ **and**<br>$(l \simeq r) \in D$ **and**<br>$\sigma = mgu(q,l)$ **and**<br>$q \notin \mathcal{V}$ **and**<br>$l\sigma \not\prec r\sigma$ **and**<br>$s\sigma \not\prec t\sigma$. |

**SUPERPOSITION INTO A NEGATIVE EQUALITY LITERAL:**

| | |
|---|---|
| $\dfrac{C' \cup \{L\} \qquad D}{((C' \cup \{L[q \to r]_\pi\} \cup D) \setminus \{l \simeq r\})\sigma}$ | **if** $L \equiv (s \not\simeq t)$ **and**<br>$q \in^o s$ **and**<br>$L\|_\pi = q$ **and**<br>$(l \simeq r) \in D$ **and**<br>$\sigma = mgu(q,l)$ **and**<br>$q \notin \mathcal{V}$ **and**<br>$l\sigma \not\prec r\sigma$ **and**<br>$s\sigma \not\prec t\sigma$. |

**EQUALITY RESOLUTION:**

| | |
|---|---|
| $\dfrac{C}{(C \setminus \{l \not\simeq r\})\sigma}$ | **if** $(l \not\simeq r) \in C$ **and**<br>$\sigma = mgu(l,r)$. |

**EQUALITY FACTORING:**

| | |
|---|---|
| $\dfrac{C' \cup \{L_1, L_2\}}{(C' \cup \{L_1\}, \neg L_1[l_1 \to r_2]_1\})\sigma}$ | **if** $L_1 \equiv (l_1 \simeq r_1)$ **and** $L_2 \equiv (l_2 \simeq r_2)$ **and** $\sigma = mgu(l_1, l_2)$ **and** $r_2\sigma \not\prec l_2\sigma$. |

## 2.1.5 Simplification rules

Simplification rules are used on multiset of retained clauses to remove redundant clauses and tautologies. In addition, simplification rules are used to replace some clauses by smaller ones, e.g., demodulation (defined below).

In the following list of simplification rules, we state the rule and then indicate the resulting multiset $S'$ of retained clauses from the original multiset $S$ after the application of the rule.

**DEMODULATION:**

| | |
|---|---|
| $\dfrac{C' \cup \{L\} \qquad D}{(C' \cup \{L[t \to r]_\pi\})\sigma}$ | **if** $D = \{l \simeq r\}$ **and** $L\|_\pi = t$ **and** $\sigma = mgu(t, l)$ **and** $t = l\sigma$ ($t$ is an instance of $l$) **and** $l \succ r$ **and** $l\sigma \succ r\sigma$. |

Retained multiset: $S' = (S \setminus \{C' \cup \{L\}\}) \cup \{(C' \cup \{L[t \to r]_\pi\})\sigma\}$.

The clause $(C' \cup \{L[t \to r]_\pi\})\sigma$ is called a **demodulant**. $D$ is called a **demodulator**. $C' \cup \{L\}$ is called an **into clause** or **demodulated clause**. Demodulation is a restricted paramodulation rule in which the replaced term $t$ in the demodulated clause is an instance of the term $l$ in the demodulator. In addition

to the restrictions over the unrestricted paramodulation, demodulation replaces the demodulated clause by the demodulant.

**DESTRUCTIVE EQUALITY RESOLUTION:**

| | |
|---|---|
| $\dfrac{C}{(C \setminus \{l \not\simeq r\})\sigma}$ | **if** $(l \not\simeq r) \in C$ **and** $l \in \mathcal{V}$ **and** $l \not\preceq^{o} r$ **and** $\sigma = mgu(l,r) = \{l \to r\}$. |

Retained multiset: $S' = (S \setminus \{C\}) \cup \{(C \setminus \{l \not\simeq r\})\sigma\}$.

**SUBSUMPTION:**

If $C \in S$, $D \in S$, and there exists a substitution $\sigma$ such that $C\sigma \subseteq D$, then $C$ **subsumes** $D$.

Retained multiset: $S' = S \setminus \{D\}$.

**SUSBSUMPTION RESOLUTION:**

| | |
|---|---|
| $\dfrac{C \quad D}{(D \setminus \{L_2\})\sigma}$ | **if** $C = C' \cup \{L_1\}$ **and** $D = D' \cup \{L_2\}$ **and** There exists a substitution $\sigma$ **:** $L_1\sigma = \neg L_2$ **and** $C'\sigma \subseteq D'$. |

$L_1$ and $L_2$ are literals.

Retained multiset: $S' = (S \setminus \{D\}) \cup \{(D \setminus \{L_2\})\sigma\}$.

**TAUTOLOGY DELETION:**

If $C \in S$ and $C = C' \cup \{A, \neg A\}$ or $C = C' \cup \{s \simeq s\}$, then $C$ is a **tautology**.

Retained multiset: $S' = S \setminus \{C\}$.

## 2.2 Summary

In this chapter we presented the basics of first-order logic required for an understanding of this thesis. We listed the common inference rules used in modern ATPs in a multiset representation to simplify the presented material in Chapter 3.

CHAPTER

# 3

# Delayed Clause-Construction

In an ATP, a derived clause can be stored either explicitly as a data structure that contains references to its literals and their terms or implicitly in a data structure that contains references to the clauses from which it was derived. A clause stored explicitly in memory is referred to as a *constructed clause*, otherwise it is referred to as a *non-constructed clause*. *Clause construction* is the process performed by an ATP to transform a non-constructed clause into a constructed clause.

*Discarding a clause* $C$ is a two step process performed by an ATP. The first step is the construction of all the non-constructed clauses referring to $C$. The second step is the deletion of $C$ from memory.

In a linear derivation a large number of intermediate conclusions are generated and discarded shortly thereafter, because they are a means to an end (which is a goal clause). The time spent in constructing and discarding intermediate conclusions can be substantial. The use of a stalling strategy called *delayed clause-construction* (DCC) can reduce this time to a minimum by delaying the construction of intermediate conclusions until they are needed. However, there are cases where intermediate conclusions must be constructed.

In this chapter, we begin by a brief discussion on the benefits of DCC and how it differs from other similar research done recently. We then introduce certain terms that are necessary to present a formal definition of delayed clause-construction. We state and discuss the cases in which intermediate conclusions must be constructed. We then derive a general formula for expressing an

intermediate conclusion in terms of constructed clauses, a single substitution set, and a single term replacement list. Finally, as a consequence of the derived general formula, we construct a mega-inference rule that combines several rules into one.

# 3.1 Benefits of DCC

## 3.1.1 Performance improvement

The construction time of an intermediate conclusion is linear in its weight. When a few hundred or thousand short clauses are constructed and discarded, the overall performance of an ATP may not be affected much (see Chapter 6). However, when hundreds of thousands of long clauses are constructed, then the overall performance is affected a lot. The time spent in constructing and discarding intermediate conclusions can be substantial. The results of the experiments we conducted (see Chapter 6) reveal that the percentage of time spent in constructing clauses can reach 65% of the total running time. By comparison with unification, which is considered as one of the most time consuming operations in an ATP, the time spent in constructing clauses is 4.86 times, on average, more than the time spent in unification (see Appendix F). This implies that clause construction can be a more time consuming process than unification. Therefore, the use of DCC to reduce the time spent in constructing clauses can improve the performance of an ATP (see Chapter 6).

## 3.1.2 The reduction of memory requirements

When DCC is employed in an ATP, intermediate conclusions are represented in a compact form that uses less memory than the amount needed to store the intermediate conclusions in their constructed form. A reduced representation of derived clauses, which is different from ours (see Chapter 6), was also accomplished in WALDMEISTER [Gaillourdet et al. 2003], [Hillenbrand & Löchner

2002]. However, in WALDMEISTER, the clauses are constructed first, in order to determine information useful for a heuristic assessment, and then they are "thrown[1]" away except for minimal information that allows the ATP to reconstruct the clause if necessary. In DCC clauses are generally not constructed except in few cases which are discussed in this chapter.

### 3.1.3 Efficiency improvement

Based on the theorems and corollaries presented in this chapter, we constructed an inference rule, called a mega-inference rule, that combines several inference rules into one. The main purpose of a mega-inference rule is to take large steps in a search, thereby improving the efficiency of an ATP. Examples showing the benefits of taking large steps in a search are given in [Wos et al. 1992]. The oldest example of combining multiple inference rules into one single inference rule in order to achieve larger steps in a search is hyperresolution. More recent examples are s-paramodulation [Benanav 1990], the linked inference principle [Veroff & Wos 1992], and the extended link strategy [Jeff Ho 1999]. However, all of the aforementioned references combine multiple applications of a single inference rule, such as binary resolution or paramodulation in one rule. The mega-inference rule combines different inference rules into one.

## 3.2 Definitions

In this section, we present the formal definitions for *term replacement list*, *p-idempotent substitution set*, *constructed* and *non-constructed clause*, *goal clause*, and *delayed clause* that are prerequisites for the understanding of delayed clause-construction.

[1] This is the term used in [Hillenbrand & Löchner 2002].

**Definition 3.1: Term replacement list**

A term replacement list is a relation between positions in literals and terms. A finite term replacement list has the form

$$\tau = \left\langle \vec{L}_1\big|_{\pi_1} \to t_1, \ldots, \vec{L}_n\big|_{\pi_n} \to t_n \right\rangle,$$

where $n \geq 0$, $\pi_1, \ldots, \pi_n$ are, respectively, valid positions in the literals $\vec{L}_1, \ldots, \vec{L}_n$ of some clauses, and $t_1, \ldots, t_n$ are, respectively, the terms replacing the terms at positions $\pi_1, \ldots, \pi_n$ in $\vec{L}_1, \ldots, \vec{L}_n$. The terms being replaced are not important as long as the positions are valid. The **empty term replacement list** is denoted by $\tau = \langle\ \rangle$; in this case $n = 0$. The literals $\vec{L}_1, \ldots, \vec{L}_n$ are references (pointers) to specific literals in some clauses stored in memory; the little arrow $\to$ on top is added to emphasize that. This specific referencing is necessary for the following reasons.

- If $\tau$ is applied to an ordered clause that contains identical literals, then only the literal referenced by a literal in $\tau$ is changed.
- If $\tau$ is applied to a multiset of clauses and any two clauses in this multiset contain identical literals, then only the literals specifically referenced by $\tau$ are changed.

Some properties of $\tau = \left\langle \vec{L}_1\big|_{\pi_1} \to t_1, \ldots, \vec{L}_n\big|_{\pi_n} \to t_n \right\rangle$ are the following:

- Some or all of $\vec{L}_1\big|_{\pi_1}, \ldots, \vec{L}_n\big|_{\pi_n}$ may be the same. Therefore, $\{\vec{L}_1\big|_{\pi_1}, \ldots, \vec{L}_n\big|_{\pi_n}\}$ is a multiset.
- $Dom(\tau) = Set(\{\vec{L}_1\big|_{\pi_1}, \ldots, \vec{L}_n\big|_{\pi_n}\})$ denotes the domain of $\tau$.
- $\mathcal{L}(\tau) = Dom(Dom(\tau)) = Set(\{\vec{L}_1, \ldots, \vec{L}_n\})$.
- $Ran(\tau) = Set(\{t_1, \ldots, t_n\})$ is the range of $\tau$.

**Definition 3.2: The application of a term replacement list to a clause**

The application of $\tau$ to a clause $C$ is denoted by $C\tau$. If $C = \phi$ then $C\tau = \phi$.

Suppose $C = \{A_1, \ldots, A_m\}$, $m > 0$ and $\tau = \left\langle \vec{L}_1\big|_{\pi_1} \rightarrow t_1, \ldots, \vec{L}_n\big|_{\pi_n} \rightarrow t_n \right\rangle$.

$C\tau$ is formed as follows.

1. Make a copy of $C$ and call it $C' = \{A_1', \ldots, A_m'\}$.
2. For $i := 1$ to $n$

   If $\vec{L}_i = A_j$ for some $1 \leq j \leq m$ then

   Replace $A_j'$ with $A_j'[t_i]_{\pi_i}$
3. Return $C'$

$C'$ is $C\tau$.

Line 2 checks if $\vec{L}_i$ is pointing to a literal in $C$, then the corresponding literal in $C'$ is changed.

**Example 3.1**

$C = \{A_1, A_2, A_3\} = \{P(f(a,b)), \neg Q(a,x), P(f(a,b))\}$.

$\tau = \left\langle A_1|_1 \rightarrow g(b), A_2|_1 \rightarrow b, A_1|_{1.1} \rightarrow g(c), A_3|_{1.2} \rightarrow d, A_4|_{1.2} \rightarrow d, A_1|_{1.1.1} \rightarrow d \right\rangle$.

Make a copy of $C$ and call it $C'$.

$C' = \{A_1', A_2', A_3'\} = \{P(f(a,b)), \neg Q(a,x), P(f(a,b))\}$.

**Table 3-1** lists the iterations performed to obtain $C\tau$.

**Table 3-1: Example of an application of a term replacement list to a clause**

| Iteration | Old literal | Position | Replacement term | New literal |
|---|---|---|---|---|
| 1 | $A_1' = P(f(a,b))$ | 1 | $g(b)$ | $A_1' := P(g(b))$ |
| 2 | $A_2' = \neg Q(a,x)$ | 1 | $b$ | $A_2' := \neg Q(b,x)$ |
| 3 | $A_1' = P(g(b))$ | 1.1 | $g(c)$ | $A_1' := P(g(g(c)))$ |
| 4 | $A_3' = P(f(a,b))$ | 1.2 | $d$ | $A_3' := P(f(a,d))$ |
| 5 | $A_4 \notin C$ so nothing happens | | | |
| 6 | $A_1' = P(g(g(c)))$ | 1.1.1 | $d$ | $A_1' := P(g(g(d)))$ |

$$C\tau = C' = \{A_1', A_2', A_3'\} = \{P(g(d(d))), \neg Q(d,x), P(f(a,d))\}$$

In **Table 3-1**, every entry in the *old literal* column is a literal from $C'$ before a term replacement is made. The corresponding entry in the *new literal* column is the literal after the term replacement is made. The new literal becomes the old literal the next time a term replacement is done to this literal. For instance, on the first iteration, $A_1'$ is equal to the original literal $A_1$ before the term $f(a,b)$ at position 1 is replaced with $g(b)$. The new literal $P(g(b))$ becomes the old literal at iteration 3.

**Partitioning property.** Every term replacement list $\tau$ can be partitioned into sequences $\tau_1, \ldots, \tau_n$, where $n = |\mathcal{L}(\tau)|$, and

$$\text{for all } 1 \le i, j \le n, \text{ if } i \ne j \text{ then } \mathcal{L}(\tau_i) \cap \mathcal{L}(\tau_j) = \{\},$$

such that if $\alpha$ and $\beta$ are elements of $\tau$, and $\alpha$ occurs before $\beta$, then if $\alpha$ and $\beta$ are in the same partition $\tau_j$, $1 \le j \le n$, then $\alpha$ occurs before $\beta$ in $\tau_j$.

Furthermore, the sequences $\tau_1,\ldots,\tau_n$ can be applied in any order to a clause $C$ and the result would still be $C\tau$. For instance, in Example 3.1, the partitions are:

$\tau_1 = \langle A_1|_1 \to g(b), A_1|_{1.1} \to g(c), A_1|_{1.1.1} \to d \rangle$,

$\tau_2 = \langle A_2|_1 \to b \rangle$,

$\tau_3 = \langle A_3|_{1.2} \to d \rangle$,

$\tau_4 = \langle A_4|_{1.2} \to d \rangle$.

These partitions (sequences) can be applied in any order to $C$ and the result would still be equal to $C\tau$. For example, $(((C\tau_1)\tau_2)\tau_3)\tau_4 = (((C\tau_4)\tau_2)\tau_1)\tau_3 = C\tau$.

The ability to partition a term replacement list into sequences is a very useful property for an efficient implementation of the application of a term replacement list to a clause. For example, as will be shown later in delayed clause-construction, most of the time only one or two literals need to be changed from one application of an inference rule to another. Because of the partitioning property of a term replacement list, only the partitions related to those literals need to be applied to those literals. Furthermore, optimizations on the application of a sequence can be done based on the positions. For example, if a term at position 1.1 in some literal appears several times in one of the sequences, then only the last one is applied. There is no need to apply the others. This is demonstrated in Example 3.2.

**Example 3.2**

$C = \{L_1, L_2\} = \{P(a,b), Q(x)\}$.

$\tau = \langle \vec{L}_1|_1 \to g(a), \vec{L}_1|_{1.1} \to g(a), \vec{L}_1|_{1.1} \to b, \vec{L}_1|_1 \to c \rangle$.

$C\tau = \{P(c,b), Q(x)\}$.

There is no need to go through all of $\tau$ in order to obtain the correct value for $C\tau$. Since $\tau$ contains only references to literal $L_1$, then only one partition is

formed. This partition is equal to $\tau$ itself. The terms at positions 1 and 1.1 are changed twice. There is no need to apply them both. Only the application of the last of each would suffice. Furthermore, the term at position 1 is a parent node in the tree (see Chapter 2 for term representation as a tree) of the term at position 1.1, and the term replacement at position 1 occurs after the term replacement at position 1.1. Therefore, only the term replacement at position 1 needs to be performed. This implies that only one term replacement is needed instead of four, i.e. only the term replacement $\vec{L}_1\big|_1 \to c$ needs to be performed. This obviously saves a lot of time.

**Definition 3.3: Composition of term replacement lists**

The composition of two term replacement lists,

$$\tau_1 = \left\langle \vec{L}_{11}\big|_{\pi_{11}} \to t_{11}, \ldots, \vec{L}_{1n}\big|_{\pi_{1n}} \to t_{1n} \right\rangle \text{ and } \tau_2 = \left\langle \vec{L}_{21}\big|_{\pi_{21}} \to t_{21}, \ldots, \vec{L}_{2n}\big|_{\pi_{2n}} \to t_{2n} \right\rangle,$$

where $n, m \geq 0$, is defined as

$$\tau_1\tau_2 = \left\langle \vec{L}_{11}\big|_{\pi_{11}} \to t_{11}, \ldots, \vec{L}_{1n}\big|_{\pi_{1n}} \to t_{1n}, \vec{L}_{21}\big|_{\pi_{21}} \to t_{21}, \ldots, \vec{L}_{2n}\big|_{\pi_{2n}} \to t_{2n} \right\rangle.$$

**Definition 3.4: Application of a substitution set to a term replacement list**

The application of a substitution set $\sigma$ to $\tau$ is defined as

$$\tau\sigma = \left\langle \vec{L}_1\sigma\big|_{\pi_1} \to t_1\sigma, \ldots, \vec{L}_n\sigma\big|_{\pi_n} \to t_n\sigma \right\rangle.$$

If $C = \{A_1, \ldots, A_m\}$, $m > 0$, then $C(\tau\sigma)$ is formed as follows.

1. Make a copy of $C$ and call it $C' = \{A_1', \ldots, A_m'\}$.
2. For $i := 1$ to $n$

   If $\vec{L}_i\sigma = A_j$ for some $1 \le j \le m$ then

   Replace $A_j'$ with $A_j'[t_i\sigma]_{\pi_i}$
3. Return $C'$

$C'$ is $C\tau$.

The following are some properties on the relation between the application of substitution sets and term replacement lists.

- If $C = \{A_1, \ldots, A_n\}$ is a clause and $\sigma$ is a substitution set, then $(C\sigma)\tau = C\sigma\tau = \{A_1\sigma, \ldots, A_n\sigma\}\tau$.
- If $C$ is a clause and $\sigma$ is a substitution set then $C\tau\sigma = (C\tau)\sigma$.
- If $C$ is a clause, $\sigma$ is a substitution set, and $\tau_1, \tau_2$ are term replacement lists, then $C\tau_1\sigma\tau_2 \Leftrightarrow (C\sigma(\tau_1\sigma))\tau_2$.
- If $C$ is a clause, $\sigma$ is a substitution set, and $L \in C$, then $L\tau \in C\tau$ and $L\tau\sigma \in C\tau\sigma$.
- If $C$ is a clause and $L \in C$, then $L\tau \in C$ if and only if either $\vec{L} \notin \mathcal{L}(\tau)$ or $\vec{L} \in \mathcal{L}(\tau)$ and $\tau$ does not change $L$. If $L\tau \in C$ then we write[1] $L\tau = L$.

  For example, suppose $C = \{L\} = \{P(f(a,b))\}$ and

  $\tau = \left\langle \vec{L}\big|_1 \to a, \vec{L}\big|_1 \to f(b,c), \vec{L}\big|_{1.1} \to a, \vec{L}\big|_{1.2} \to b \right\rangle$ then

  $\tau$ does not change $L$ and therefore, $L\tau = L$.

[1] This property is important because a term replacement list is associated with clauses. Therefore, a clear definition of the notation $L\tau$ (which is an application of a term replacement list to a literal instead of clause) must be given.

**Definition 3.5: P-idempotent substitution set**

A **p-idempotent** (i.e., potentially idempotent) substitution set is a substitution set that contains no circular substitutions. When applied recursively a finite number of times, a p-idempotent substitution set produces an idempotent substitution. A p-idempotent substitution is distinguished from an idempotent substitution by "..." on top of the substitution symbol, such as $\ddot{\sigma}$ and $\ddot{\theta}$. We write $\ddot{\sigma} \rightarrow_n \sigma$ to mean

$$\sigma = \ddot{\sigma}\underbrace{(\ddot{\sigma}(\ldots\ddot{\sigma}(\ddot{\sigma}(\ddot{\sigma}\ddot{\sigma}))\ldots))}_{n}, \text{ for some } n \geq 0.$$

$n$ is the minimum number of applications of $\ddot{\sigma}$ over itself needed to become idempotent. The idempotent set $\sigma$ obtained from $\ddot{\sigma}$ is unique, i.e., for all $m > n$, $\ddot{\sigma} \rightarrow_m \sigma$; for all $m < n$, the resulting substitution obtained from the application of $\ddot{\sigma}$ over itself $m$ times remains p-idempotent. When $n = 0$, $\ddot{\sigma} \rightarrow_0 \sigma \Rightarrow \sigma = \ddot{\sigma}$. Therefore, every idempotent substitution set is p-idempotent. If $n$ is not important, we write $\ddot{\sigma} \rightarrow \sigma$.

The following are some properties of p-idempotent substitution sets:

- If $\ddot{\sigma} \rightarrow \sigma$ then $Dom(\ddot{\sigma}) = Dom(\sigma)$.
- The composition of p-idempotent substitution sets is associative, i.e., if $\ddot{\sigma}, \ddot{\theta}, \ddot{\mu}$ are three p-idempotent substitution sets, then $\ddot{\sigma}\ddot{\theta}\ddot{\mu} = \ddot{\sigma}(\ddot{\theta}\ddot{\mu}) = (\ddot{\sigma}\ddot{\theta})\ddot{\mu}$.
- The associativity property implies that $\ddot{\sigma}\ddot{\sigma}\ddot{\sigma}\ddot{\sigma} = \ddot{\sigma}(\ddot{\sigma}\ddot{\sigma}\ddot{\sigma}) = (\ddot{\sigma}\ddot{\sigma}\ddot{\sigma})\ddot{\sigma}$. Therefore, if $\ddot{\sigma} \rightarrow \sigma$ then

$$\sigma = \ddot{\sigma}\underbrace{(\ddot{\sigma}(\ldots\ddot{\sigma}(\ddot{\sigma}(\ddot{\sigma}\ddot{\sigma}))\ldots))}_{n} = \ddot{\sigma}\underbrace{\ddot{\sigma}\cdots\ddot{\sigma}}_{n}, \text{ for some } n \geq 0.$$

**Example 3.3**

$\ddot{\sigma} = \{x \rightarrow f(y), y \rightarrow g(z,w), z \rightarrow f(w), w \rightarrow a\}$ is p-idempotent because it can be transformed into an idempotent substitution as follows (using the definition of composition of substitution sets stated in Chapter 2).

$\ddot{\sigma}_1 = \ddot{\sigma}\ddot{\sigma} = \{x \to f(g(z,w)), y \to g(f(w),a), z \to f(a), w \to a\}$

$\ddot{\sigma}_2 = \ddot{\sigma}_1\ddot{\sigma} = \ddot{\sigma}\ddot{\sigma}_1 = \ddot{\sigma}\ddot{\sigma}\ddot{\sigma} = \{x \to f(g(f(w),a)), y \to g(f(a),a), z \to f(a), w \to a\}$

$\ddot{\sigma}_3 = \ddot{\sigma}_2\ddot{\sigma} = \ddot{\sigma}\ddot{\sigma}_2 = \ddot{\sigma}\ddot{\sigma}\ddot{\sigma}\ddot{\sigma} = \{x \to f(g(f(a),a)), y \to g(f(a),a), z \to f(a), w \to a\}$

$\ddot{\sigma}_4 = \ddot{\sigma}_3\ddot{\sigma} = \ddot{\sigma}\ddot{\sigma}_3 = \ddot{\sigma}\ddot{\sigma}\ddot{\sigma}\ddot{\sigma}\ddot{\sigma} = \{x \to f(g(f(a),a)), y \to g(f(a),a), z \to f(a), w \to a\}$

Since $\ddot{\sigma}_4 = \ddot{\sigma}_3$ then the transformation process is over and $\ddot{\sigma} \to \sigma$, where $\sigma = \ddot{\sigma}\ddot{\sigma}\ddot{\sigma}\ddot{\sigma} = \{x \to f(g(f(a),a)), y \to g(f(a),a), z \to f(a), w \to a\}$.

**Example 3.4**

**Table 3-2** shows some examples of substitution sets in the first column and whether or not they are p-idempotent in the second column.

**Table 3-2: Examples of p-idempotent and not p-idempotent substitution sets**

| Substitution set | P-idempotent | Idempotent |
|---|---|---|
| (1) $\{x \to y, z \to f(y)\}$ | Yes | Yes |
| (2) $\{x \to y, y \to f(z), z \to w\}$ | Yes | No |
| (3) $\{x \to y, y \to z, z \to f(x)\}$ | No | No |

In **Table 3-2** the substitution set in (2) is p-idempotent because it can be transformed into the idempotent substitution set $\{x \to f(w), y \to f(w), z \to w\}$. However, the substitution set in (3) is not p-idempotent because of the circular substitution. It is not possible to transform this substitution set into an idempotent substitution by applying the substitution on itself a finite number of times.

**Partitioning property.** A p-idempotent substitution $\ddot{\sigma}$ can be partitioned into $k \geq 0$ idempotent substitution subsets, $\mu_1, \ldots, \mu_k$ such that $\ddot{\sigma} = \mu_1 \cup \cdots \cup \mu_k$. Furthermore, there exists a partitioning with a minimum number of partitions and there exists a partition with the following property:

the subsets $\mu_1,\ldots,\mu_k$ are ordered in a way such that for every $1 \le i \le k$,

$$\text{for all } v \in Dom(\mu_i) \Rightarrow v \not\in^{o} \bigcup_{j=1}^{k} Ran(\mu_j).$$

**Example 3.5**

$\ddot{\sigma} = \{x \to f(y), y \to g(z,w), z \to f(w), w \to a\}$ can partitioned into idempotent substitution sets. The possible partitions are shown in **Table 3-3**.

**Table 3-3: Partitions of $\ddot{\sigma} = \{x \to f(y), y \to g(z,w), z \to f(w), w \to a\}$**

| | **Partitions of $\ddot{\sigma} = \{x \to f(y), y \to g(z,w), z \to f(w), w \to a\}$** |
|---|---|
| (1) | $\mu_1 = \{x \to f(y)\}, \mu_2 = \{y \to g(z,w)\}, \mu_3 = \{z \to f(w)\}, \mu_4\{w \to a\}$ |
| (2) | $\mu_1 = \{x \to f(y), w \to a\}, \mu_2 = \{y \to g(z,w)\}, \mu_3 = \{z \to f(w)\}$ |

In **Table 3-3**, (1) is the partitioning of $\ddot{\sigma}$ into idempotent subsets such that none of the variables in the domain of $\mu_i$ appears in the range of the following sets. The partitioning in (2) shows a partitioning with the minimum number of partitions.

The following definitions and theorems concerning p-idempotent substitution sets are essential for deriving a general expression for a sound non-constructed conclusion.

**Definition 3.6: Consistency of p-idempotent substitution sets**

Two p-idempotent substitution sets $\ddot{\sigma}$ and $\ddot{\theta}$ are **consistent** if and only if

1. $Dom(\ddot{\sigma}) \cap Dom(\ddot{\theta}) = \{\}$, and
2. $\ddot{\sigma} \cup \ddot{\theta}$ is p-idempotent.

This definition can be extended to any number of p-idempotent substitution sets. Notice that since idempotent substitution sets are p-idempotent, then the above definition applies to idempotent substitution sets as well.

**Definition 3.7: Confluent p-idempotent substitution sets**

Two p-idempotent substitution sets $\ddot{\sigma}_1$ and $\ddot{\sigma}_2$ are **confluent** if and only if $\ddot{\sigma}_1 \to \theta$ and $\ddot{\sigma}_2 \to \theta$. If $\ddot{\sigma}_1$ and $\ddot{\sigma}_2$ are confluent, we write $\ddot{\sigma}_1 \Downarrow \ddot{\sigma}_2$. Confluent p-idempotent substitution sets have the following properties.

**Property 1.** If $\ddot{\sigma}_1 \Downarrow \ddot{\sigma}_2$ then $Dom(\ddot{\sigma}_1) = Dom(\ddot{\sigma}_2)$.

**Property 2.** If $\ddot{\sigma}_1 \Downarrow \ddot{\sigma}_2$, then for any p-idempotent set $\ddot{\sigma}_3$, such that $\ddot{\sigma}_1$ and $\ddot{\sigma}_3$ are consistent, and $\ddot{\sigma}_2$ and $\ddot{\sigma}_3$ are consistent, $\ddot{\sigma}_1 \cup \ddot{\sigma}_3 \Downarrow \ddot{\sigma}_2 \cup \ddot{\sigma}_3$.

**Example 3.6**

**Table 3-4** shows some examples of p-idempotent substitution sets that are confluent and not confluent.

**Table 3-4: Examples of confluent and not confluent p-idempotent substitution sets.**

| | $\ddot{\sigma}_1$ | $\ddot{\sigma}_2$ | **Confluent** |
|---|---|---|---|
| (1) | $\{x \to f(y), y \to g(z), z \to a\}$ | $\{x \to f(a), y \to g(z), z \to a\}$ | Yes |
| (2) | $\{x \to f(y), y \to g(z)\}$ | $\{x \to f(y), y \to g(w)\}$ | No |
| (3) | $\{x \to f(w), y \to g(w)\}$ | $\{x \to f(w), y \to g(z), z \to w\}$ | No |

$\ddot{\sigma}_1$ and $\ddot{\sigma}_2$ of row (2) in **Table 3-4** are not confluent because $\ddot{\sigma}_1 \to \theta_1$ and $\ddot{\sigma}_2 \to \theta_2$, where $\theta_1 = \{x \to f(g(z)), y \to g(z)\}$, $\theta_2 = \{x \to f(g(w)), y \to g(w)\}$, but $\theta_1 \neq \theta_2$. In row (3) $\ddot{\sigma}_1$ and $\ddot{\sigma}_2$ are not confluent because their domains are not equal.

**Theorem 3.1**

*Given a p-idempotent substitution set* $\ddot{\sigma}$, *if* $\ddot{\sigma} \rightarrow \sigma$ *then* $\ddot{\sigma} \Downarrow \sigma$.

**Proof:**

Since every idempotent substitution set is p-idempotent, then $\sigma$ is p-idempotent and $\sigma \rightarrow \sigma$. If $\ddot{\sigma} \rightarrow \sigma$ then we have $\ddot{\sigma} \rightarrow \sigma$ and $\sigma \rightarrow \sigma$. This implies that $\ddot{\sigma} \Downarrow \sigma$ by Definition 3.7. Therefore if $\ddot{\sigma} \rightarrow \sigma$ then $\ddot{\sigma} \Downarrow \sigma$.□

**Theorem 3.2**

*Given two idempotent substitution sets* $\sigma_1$ *and* $\sigma_2$ *that are consistent, if none of the variables in* $Dom(\sigma_1)$ *occurs in any of the terms in* $Ran(\sigma_2)$, *then* $\sigma_1 \cup \sigma_2 \rightarrow \sigma_1\sigma_2$.

**Proof:**

The proof is listed in Appendix B due to its technical complexity.

**Theorem 3.3**

*Given* $k \geq 2$ *idempotent substitution sets* $\sigma_1,\ldots,\sigma_k$ *that are pair-wise consistent, if for each* $1 \leq i \leq k-1$ *none of the variables in* $Dom(\sigma_i)$ *occurs in any of the terms in any* $Ran(\sigma_j)$, *where* $i+1 \leq j \leq k$, *then* $\sigma_1 \cup \cdots \cup \sigma_k \rightarrow \sigma_1 \cdots \sigma_k$.

**Proof:**

The proof is listed in Appendix B due to its technical complexity.

**Theorem 3.4**

*Given* $k \geq 2$ *p-idempotent substitution sets* $\ddot{\sigma}_1,\ldots,\ddot{\sigma}_k$ *that are pair-wise consistent, if for each* $1 \leq i \leq k-1$ *none of the variables in* $Dom(\ddot{\sigma}_i)$ *occurs in any of the terms in any* $Ran(\ddot{\sigma}_j)$, *where* $i+1 \leq j \leq k$, *and* $\ddot{\sigma}_1 \rightarrow \sigma_1,\ldots,\ddot{\sigma}_k \rightarrow \sigma_k$, *then* $\ddot{\sigma}_1 \cup \cdots \cup \ddot{\sigma}_k \rightarrow \sigma_1 \cdots \sigma_k$.

**Proof:**

Since $\ddot{\sigma}_1 \rightarrow \sigma_1,\ldots,\ddot{\sigma}_k \rightarrow \sigma_k$ then by Theorem 3.1 $\ddot{\sigma}_1 \Downarrow \sigma_1,\ldots,\ddot{\sigma}_k \Downarrow \sigma_k$. Since $\ddot{\sigma}_1 \Downarrow \sigma_1,\ldots,\ddot{\sigma}_k \Downarrow \sigma_k$ and $\ddot{\sigma}_1,\ldots,\ddot{\sigma}_k$ are pair-wise consistent, then by the second property from Definition 3.7 $\ddot{\sigma}_1 \cup \cdots \cup \ddot{\sigma}_k \Downarrow \sigma_1 \cup \cdots \cup \sigma_k$.

By Definition 3.7, $\ddot{\sigma}_1 \cup \cdots \cup \ddot{\sigma}_k \Downarrow \sigma_1 \cup \cdots \cup \sigma_k$ means $\ddot{\sigma}_1 \cup \cdots \cup \ddot{\sigma}_k \rightarrow \theta$ and $\sigma_1 \cup \cdots \cup \sigma_k \rightarrow \theta$. By Theorem 3.3, $\sigma_1 \cup \cdots \cup \sigma_k \rightarrow \sigma_1 \cdots \sigma_k$.

By Definition 3.5 $\theta$ is unique; therefore $\theta = \sigma_1 \cdots \sigma_k$ . Since $\ddot{\sigma}_1 \cup \cdots \cup \ddot{\sigma}_k \rightarrow \theta$, then $\ddot{\sigma}_1 \cup \cdots \cup \ddot{\sigma}_k \rightarrow \sigma_1 \cdots \sigma_k$.□

**Definition 3.8: Constructed and non-constructed clauses**

Abstractly, a clause is **constructed** when its literals are explicitly listed, and it is no longer expressed in terms of other clauses, substitution sets, and term replacement lists; otherwise, the clause is referred to as a **non-constructed clause**. For example, $\{P(x),Q(a)\}$ is a constructed clause, whereas $\{P(x)\}\sigma\tau$ is a non-constructed clause.

"To evaluate an expression representing a non-constructed clause" or simply "to evaluate a non-constructed clause" means to replace all references to literals from constructed clauses by copies of the literals themselves and all substitution sets and term replacement lists are then applied to those literals. A non-constructed

clause becomes a constructed clause when the expression representing the non-constructed clause is evaluated. When a constructed clause is constructed, it is also normalized. Therefore, a constructed clause is a normalized clause.

To avoid confusion between constructed and non-constructed clauses, we use "..." on top of a clause label to denote a non-constructed clause. For example, given the two constructed clauses $P_1 = \{\neg Q(x), R(y,x)\}$ and $P_2 = \{Q(f(a))\}$, their resolvent $C = \{R(x, f(a))\}$ is a constructed clause. On the other hand, $\ddot{C} = ((P_1 \setminus \{\neg Q(x)\}) \cup (P_2 \setminus \{Q(f(a))\}))\sigma$, where $\sigma = \{x \to f(a)\}$, is a non-constructed resolvent. If $\ddot{C}$ is evaluated and the resulting clause is exactly the same as $C$, then $\ddot{C}$ is said to be **an acceptable representation of *C*** or simply, $\ddot{C}$ is acceptable.

In general, when a non-constructed conclusion, $\ddot{C}(\mathcal{I})$, of an inference $\mathcal{I}$ is evaluated and the resulting clause is exactly $C(\mathcal{I})$, then $\ddot{C}(\mathcal{I})$ is **acceptable**. If $\ddot{C}(\mathcal{I})$ is acceptable, then we write $\ddot{C}(\mathcal{I}) \to C(\mathcal{I})$.

**Example 3.7**

$S = \{P_1, P_2, P_3\}$ is a set of constructed clauses such that

$P_1 = \{L_1\} = \{R(f(g(x), y))\}$, $P_2 = \{L_2\} = \{g(a) \simeq b\}$, and $P_3 = \{L_3\} = \{f(x,y) \simeq a\}$.

Suppose that $\Delta = \langle \mathcal{I}_1, \mathcal{I}_2 \rangle$ is a linear derivation such that $\mathcal{I}_1$ is a paramodulation from $P_2$ into $P_1$, and $\mathcal{I}_2$ is a paramodulation from $P_3$ into $C(\mathcal{I}_1)$. The first inference yields the paramodulant (after normalizing the variables) $C(\mathcal{I}_1) = \{L_4\} = \{R(f(b,x))\}$, the mgu $\sigma_1 = \{x \to a\}$, and the term replacement list $\tau_1 = \langle \vec{L}_1|_{1.1} \to b \rangle$. Before using $P_3$ as a paramodulator in inference $\mathcal{I}_2$, we rename its variables according to the naming convention stated in Chapter 2. Let

$P_3' = P_3\theta = \{L_3'\} = \{f(z,u) \simeq a\}$, where $\theta = \{x \to z, y \to u\}$ is a variable renaming substitution set. The inference $\mathcal{I}_2$ yields the paramodulant $C(\mathcal{I}_2) = \{L_5\} = \{R(a)\}$, the mgu $\sigma_2 = \{z \to b, u \to x\}$ and the term replacement list $\tau_2 = \left\langle \vec{L}_4\big|_1 \to a \right\rangle$. **Table 3-5** shows some acceptable and unacceptable non-constructed conclusions for Example 3.7.

**Table 3-5: Examples of acceptable and unacceptable non-constructed clauses**

| Non-constructed conclusion | Evaluation (variables normalized) | Acceptable |
|---|---|---|
| (1) $\ddot{C}(\mathcal{I}_1) = ((P_1 \cup P_2) \setminus \{L_2\})\tau_1\sigma_1$ | $\{R(f(b,x))\}$ | Yes |
| (2) $\ddot{C}(\mathcal{I}_2) = ((C(\mathcal{I}_1) \cup P_3') \setminus \{L_3\})\tau_2\sigma_2$ | $\{R(a), f(x,y) \simeq a\}$ | No |
| (3) $\ddot{C}(\mathcal{I}_2) = ((C(\mathcal{I}_1) \cup P_3') \setminus \{L_3'\})\tau_2\sigma_2$ | $\{R(a)\}$ | Yes |
| (4) $\ddot{C}(\mathcal{I}_2) = (((((P_1 \cup P_2) \setminus \{L_2\})\tau_1\sigma_1) \cup P_3') \setminus \{L_3'\})\tau_2\sigma_2$ | $\{R(f(b,x))\}$ | No |
| (5) $\ddot{C}(\mathcal{I}_2) = (((((P_1 \cup P_2) \setminus \{L_2\})\tau_1\sigma_1) \cup P_3') \setminus \{L_3'\})\tau_2'\sigma_2$ where $\tau_2' = \left\langle \vec{L}_1\big|_1 \to a \right\rangle$ | $\{R(a)\}$ | Yes |
| (6) $\ddot{C}(\mathcal{I}_2) = (((((P_1 \cup P_2) \setminus \{L_2\}) \cup P_3') \setminus \{L_3'\})\tau_1\sigma_1)\tau_2'\sigma_2$ where $\tau_2' = \left\langle \vec{L}_1\big|_1 \to a \right\rangle$ | $\{R(a)\}$ | Yes |

The non-constructed conclusions (2) and (4) in **Table 3-5** are unacceptable for the following reasons.

In (2) the expression indicates that the literal $L_3$ should be removed from $C(\mathcal{I}_1) \cup P_3'$. $L_3$ is not an element of $P_3'$. It is an element of $P_3$ and the variables in $P_3$ were renamed. Therefore, $(C(\mathcal{I}_1) \cup P_3') \setminus \{L_3\} = C(\mathcal{I}_1) \cup P_3'$. The evaluation of $(C(\mathcal{I}_1) \cup P_3')\tau_2\sigma_2$, as indicated in the **Table 3-5**, yields $\{R(a), f(x,y) \simeq a\}$ which

is not the correct paramodulant $\{R(a)\}$. Therefore, the non-constructed conclusion in (2) is unacceptable. A similar non-constructed conclusion that is acceptable is given in (3), where $L_3$ is replaced with $L_3'$.

In (4) the literal referred to in $\tau_2$ is $L_4 = R(f(b,x))$. $L_4$ does not exist in the expression $((((P_1 \cup P_2) \setminus \{L_2\})\tau_1\sigma_1) \cup P_3') \setminus \{L_3'\}$. Therefore, $\tau_2$ has no effect on the expression $((((P_1 \cup P_2) \setminus \{L_2\})\tau_1\sigma_1) \cup P_3') \setminus \{L_3'\}$. The evaluation of the non-constructed conclusion in (4), as indicated in **Table 3-5**, yields $\{R(f(b,x))\}$ which is not the correct paramodulant $\{R(a)\}$. Therefore, the non-constructed conclusion in (4) is unacceptable. A similar but acceptable non-constructed conclusion is given in (5), where $\tau_2$ is replaced with $\tau_2'$. Another acceptable non-constructed conclusion is given in (6). In comparison with the expression in (5), the substitution set $\sigma_1$ and the term replacement list $\tau_1$ in (6) were moved to the front of the expression. The ability to move substitution sets and term replacement lists to the end of an expression of a non-constructed conclusion while keeping it acceptable is a fundamental issue in DCC as will be shown later on.

**Definition 3.9: Goal clause**

A **goal clause** is a constructed clause (except if it is the empty clause) derived at depth $k \geq 1$ by a linear derivation from a set of constructed clauses, and it conforms to criteria either initially set by the user or determined automatically by an ATP. A goal clause is retained in memory and used in derivations under the same restrictions or a subset of those restrictions imposed on input clauses. Some examples of goal clauses are the following.

- The empty clause which when obtained marks the end of a proof by refutation.

- A unit clause that is not subsumed by already retained constructed clauses. Unit clauses are useful in UR-resolution [Wos et al. 1992] and in the unit preference strategy [Wos et al. 1964].
- A lemma as defined in [Loveland 1978] and [Astrachan 1992].
- An equation or disequation. Those are useful when equality based inference and simplification rules are applied.

**Definition 3.10: End of a derivation**

We say the end of a derivation is attained when one of the following occurs.

- The empty clause is obtained.
- A goal clause is obtained.
- The depth bound is reached in an iteratively deepening depth first search (see Chapter 4).

**Definition 3.11: General form of the conclusion of an inference rule**

We denote the multiset union of the literals of a multiset of clauses $\mathcal{D}$ by $\mathcal{L}(\mathcal{D})$. Formally, if $\mathcal{D} = \{C_1, \ldots, C_n\}$ is a multiset of clauses then the multiset

$$\mathcal{L}(\mathcal{D}) = \bigcup_{i=1}^{n} C_i .$$

Using a term replacement list, we can write the conclusion of any of the inference rules listed in Chapter 2 in the form

$$((\boldsymbol{C} \setminus \boldsymbol{D}) \cup \boldsymbol{E})\sigma(\tau\sigma),$$

where $\boldsymbol{C} = \mathcal{L}(Prem(\mathcal{I}))$, $\boldsymbol{D} \subseteq \mathcal{L}(Prem(\mathcal{I}))$, $\boldsymbol{E} \subseteq \{\neg L\}$, $L \in \mathcal{L}(Prem(\mathcal{I}))$, $\mathcal{I}$ is an inference, $\sigma$ is an mgu of some terms or literals from the premises, and $\tau$ is a term replacement list of some terms in some literals from the premises. **Table 3-6** lists the conclusions of the inference rules from Chapter 2 but expressed using $\tau$

instead of $L[t \to r]_\pi$ or $L[q \to r]_\pi$. The premises and the conditions of the inference rules remain exactly the same. If $\tau = \langle\ \rangle$ then $\sigma(\tau\sigma) = \sigma$. In all the inference rules, except equality factoring, $\boldsymbol{E}$ is not shown because $\boldsymbol{E} = \{\}$.

**Table 3-6: Inference rules conclusions in the form $((\boldsymbol{C} \setminus \boldsymbol{D}) \cup \boldsymbol{E})\sigma(\tau\sigma)$**

| Inference Rule | Conclusion | $\tau$ |
|---|---|---|
| Binary resolution | $((\underbrace{(C_1 \cup C_2)}_{C} \setminus \underbrace{\{A, \neg B\}}_{D})\sigma(\tau\sigma)$ | $\langle\ \rangle$ |
| Binary factoring | $(\underbrace{(C}_{C} \setminus \underbrace{\{L_2\}}_{D})\sigma(\tau\sigma)$ | $\langle\ \rangle$ |
| Hyper-resolution | $((\underbrace{(C_1 \cup \cdots \cup C_n)}_{C} \setminus \underbrace{\{A_1, \ldots, A_{n-1}, \neg B_1, \ldots, \neg B_{n-1}\}}_{D})\sigma(\tau\sigma)$ | $\langle\ \rangle$ |
| Paramodulation | $((\underbrace{(C' \cup \{L\} \cup D)}_{C} \setminus \underbrace{\{l \simeq r\}}_{D})\sigma(\tau\sigma)$ | $\langle \vec{L}\|_\pi \to r \rangle$ |
| Superposition into non-equality | $((\underbrace{(C' \cup \{L\} \cup D)}_{C} \setminus \underbrace{\{l \simeq r\}}_{D})\sigma(\tau\sigma)$ | $\langle \vec{L}\|_\pi \to r \rangle$ |
| Superposition into a positive equality | $((\underbrace{(C' \cup \{L\} \cup D)}_{C} \setminus \underbrace{\{l \simeq r\}}_{D})\sigma(\tau\sigma)$ | $\langle \vec{L}\|_\pi \to r \rangle$ |
| Superposition into a negative equality | $((\underbrace{(C' \cup \{L\} \cup D)}_{C} \setminus \underbrace{\{l \simeq r\}}_{D})\sigma(\tau\sigma)$ | $\langle \vec{L}\|_\pi \to r \rangle$ |
| Equality resolution | $(\underbrace{(C}_{C} \setminus \underbrace{\{l \not\simeq r\}}_{D})\sigma(\tau\sigma)$ | $\langle\ \rangle$ |
| Equality factoring | $(((\underbrace{(C' \cup \{L_1, L_2\})}_{C}) \setminus \underbrace{\{L_2\}}_{D}) \cup \underbrace{\{\neg L_1'\}}_{E})\sigma(\tau\sigma)$ | $\langle \vec{L}_1'\|_1 \to r_2 \rangle$ |
| Demodulation | $((\underbrace{(C' \cup \{L\} \cup D)}_{C} \setminus \underbrace{D}_{D})\sigma(\tau\sigma)$ | $\langle \vec{L}\|_\pi \to r \rangle$ |

**Theorem 3.5**

*In a linear-input derivation, if no intermediate conclusion is a from clause, then there is no need to construct any intermediate conclusion, and every non-constructed intermediate conclusion* $\dddot{C}$ *can be expressed in terms of variants of constructed clauses, a single substitution set* $\sigma$, *and a single term replacement list* $\tau$. $\sigma$ *is the composition of all the mgu's resulting from the inferences performed from the beginning of the derivation up to and including the inference that produced* $\ddot{C}$. $\tau$ *is the composition of all the term replacement lists from the beginning of the derivation up to and including the inference that produced* $\dddot{C}$.

**Proof:**

The proof is listed in Appendix B due to its technical complexity.

**Corollary 3.1**

*In a linear derivation, if every far parent and every intermediate conclusion that is a from clause is constructed, then any intermediate conclusion that is not a far parent or a from clause can be expressed in terms of the input clauses, the constructed intermediate clauses, a single substitution set, and a term replacement list.*

**Proof:**

Suppose $S$ is the set of input clauses used in a derivation $\Delta$. Let $T$ be the set of intermediate conclusions that are far parents and *from clauses*. Therefore, $T$ is a set of constructed clauses. Let $S' = S \cup T$. $S'$ is a set of constructed clauses. With $S'$ as a set of constructed clauses, $\Delta$ can be viewed as a linear input derivation where each side premise is a variant of a clause from $S'$. In this case, we have the same conditions as the ones indicated in Theorem 3.5. Therefore, by Theorem 3.5, we can conclude that any intermediate conclusion that is not a far parent or a

from clause can be expressed in terms of the input clauses, the constructed intermediate clauses, a single substitution set, and a term replacement list. □

**Corollary 3.2**

*In a linear derivation, if every far parent and every intermediate conclusion that is a from clause is constructed, then any intermediate conclusion* $\ddot{\ddot{C}}$ *that is not a far parent or a from clause can be expressed in terms of the input clauses, the constructed intermediate clauses, a single p-idempotent substitution set* $\ddot{\sigma}$*, and a term replacement list* $\tau$*.* $\ddot{\sigma}$ *is the union of all the mgu's resulting from the inferences performed from the beginning of the derivation up to and including the inference that produced* $\ddot{\ddot{C}}$*.* $\tau$ *is the composition of all the term replacement lists from the beginning of the derivation up to and including the inference that produced* $\ddot{\ddot{C}}$*.*

**Proof:**

Suppose $\Delta = \langle I_1, \ldots, I_k \rangle$. From Corollary 3.1, we can conclude that any intermediate conclusion $\ddot{\ddot{C}}(I_i)$, $1 \le i \le k$, that is not a far parent or a *from clause* can be written in terms of constructed clauses (input and intermediate conclusions), a single idempotent substitution set $\sigma_{1..i}$, and a term replacement list. From Theorem 3.1 and Theorem 3.3 we conclude that the union of consistent idempotent substitution sets is confluent with the composition of those substitution sets. Therefore, we can replace $\sigma_1 \cdots \sigma_i$ with $\ddot{\sigma}_{1..i} = \sigma_1 \cup \cdots \cup \sigma_i$, for $1 \le i \le k$, and hence, any intermediate conclusion $\ddot{\ddot{C}}(I_i)$ that is not a *from clause* or a far parent can be expressed in terms of constructed clauses, a single p-idempotent substitution set $\ddot{\sigma}_{1..i} = \sigma_1 \cup \cdots \cup \sigma_i$, and a term replacement list. □

**Definition 3.12: Delayed clause**

In a linear derivation, any intermediate conclusion that is not constructed is a **delayed clause**. When the end of a derivation is reached, any delayed clause that has not yet been constructed is discarded.

**Definition 3.13: Delayed clause-construction**

Except for hyperresolution, all the inference rules listed in Chapter 2 perform either a unification on two terms in one or two literals, or perform a unification between two literals. In a linear derivation, there is no need to construct a whole clause (i.e., intermediate conclusion) just to perform a unification on a part of it. However, sometimes there are cases where an intermediate conclusion needs to be constructed and other times when it is not necessary to construct the clause but it is better to construct it. From the theorems and corollaries stated above, we can deduce that when a clause is used as a far parent or as a *from clause*, then its construction is necessary. If an intermediate conclusion is not a far parent or a *from clause* but conforms to certain criteria that would make it potentially useful beyond its current purpose (i.e. as an intermediate conclusion), then it is worth constructing the whole clause and storing it for future use. Another reason to construct an intermediate conclusion is if the time it takes to construct it and determine its attributes is less than the time it takes to determine its attributes (e.g., length, weight, ...) without constructing it (see Appendix C). The above three justifications for constructing delayed clauses fall into one of the three following categories used as guidance for an ATP to decide whether to construct a delayed clause or not.

- Strategies implemented.
- Heuristics.
- The time to construct a clause and then determine its attributes with the time to determine its attributes without constructing it.

These three issues are discussed further in Chapter 4 when semi-linear resolution is introduced. The strategy of delaying the construction of intermediate clauses without the need to maintain more than one substitution set and one term replacement list, is called **delayed clause-construction (DCC).**

It is possible to delay the construction of intermediate conclusions if several substitution sets and several term replacements lists are maintained, but this has two disadvantages. First, more storage space is required for each substitution set and term replacement list. Second, more complicated data structures are needed to create a relationship between the substitution sets and term replacement lists so that when the time comes to construct a delayed clause, the construction process can be done efficiently. If more complicated data structures are used, then more operations are needed for their maintenance, and thus more time is wasted. On the other hand, if a single substitution set and a single term replacement list are used, then less storage space is needed and the data structures are quite simple and easy to maintain, as will be shown in Chapter 6.

Theorem 3.5 and corollaries 3.1 and 3.2 prove that the construction of an intermediate conclusion can be expressed in terms of constructed clauses, a single p-idempotent substitution set, and a single term replacement list. Furthermore, any delayed clause can be constructed at a later time from the information of the maintained substitution set and term replacement list. We now summarize the conditions required to be able to delay clauses and use a single substitution set and term replacement list to construct any of the delayed clauses at a later time.

Let $\Delta = \langle I_1, \ldots, I_k \rangle$, $k \geq 1$, be a linear derivation of a goal clause $G$ with $C_{init}$ as its initial clause. The construction of an intermediate conclusion $C(I_i)$, $1 \leq i < k$, can be delayed if $\ddot{C}(I_i) \rightarrow C(I_i)$. From Corollary 3.2 we deduce that $\ddot{C}(I_i) \rightarrow C(I_i)$ is possible if the following three conditions are satisfied.

**Condition 1:** $C_{init}$ is a constructed clause and every clause in the multiset union $\bigcup_{i=1}^{k} \mathcal{D}(I_i)$ is a constructed clause.

**Condition 2:** $Vars(C_{init}) \cap Vars(\bigcup_{i=1}^{k} \mathcal{D}(I_i)) = \{\}$ and for all $C \in \bigcup_{i=1}^{k} \mathcal{D}(I_i)$ and for all $D \in \bigcup_{i=1}^{k} \mathcal{D}(I_i)$, if $C \neq D$ then $Vars(C) \cap Vars(D) = \{\}$.

In other words, no variable is shared between the initial clause and any of the side premises, and no variable is shared between any of the side premises.

**Condition 3:** The mgu's resulting from the inferences $I_1,\ldots,I_i$, must be p-idempotent and consistent.

Condition 1 ensures that only intermediate conclusions in a linear derivation can be non-constructed clauses. If one of the intermediate conclusions in a linear derivation is used as a side premise, that is if an intermediate conclusion is a far parent, then it must be constructed first.

Condition 2 says that the variable names of any two side premises must be disjoint. This ensures that every mgu formed from the unification of terms or literals at any depth is consistent with all the mgu's formed at earlier depths. This eliminates variable substitution ambiguity when the union of all the mgu's is formed. Condition 2 is very important because in DCC intermediate conclusions are generally[1] not constructed and normalized. Therefore, variables of all clauses participating in the linear derivation are susceptible to modification. Any ambiguity concerning the substitution of variables by the substitution terms can lead to an unsound derivation due to the dependency of the variables upon each other. Chapter 6 demonstrates how the renaming of variables in a side premise can be done efficiently (in almost constant time) without the need to traverse the clause.

[1] Only far parents and *from clauses* are constructed.

Condition 3 is necessary to maintain a single substitution set which can be accessed at any time during a linear derivation in order to construct a delayed clause without leading to an unacceptable constructed clause. Condition 3 is also necessary for an efficient backtracking in depth-first search. Without Condition 3, backtracking becomes very inefficient, and practically impossible without the storage of additional information about the derivation (see section 3.3).

The three conditions ensure that the construction of a delayed clause at any time during a linear derivation leads to a clause which is exactly the same clause had the delayed clause been constructed at the time of its generation. Consequently, any clause obtained by a linear derivation employing DCC is obtained by a sound derivation. This is demonstrated in Example 3.8.

**Example 3.8**

$S = \{B_1, B_2, B_3\}$ is a set of input clauses, where

$B_1 = \{L_{11}, L_{21}\} = \{P(f(x_0)), Q(x_0)\}$,

$B_2 = \{L_{12}, L_{22}\} = \{\neg P(x_0), Q(x_0)\}$,

$B_3 = \{L_{13}, L_{23}\} = \{\neg Q(a), \neg Q(f(b)\}$.

In **Figure 3-1** the derivation of the clause $C_3 = \{Q(f(a)), Q(f(b))\}$ from $S$ is shown. DCC is not used in this derivation, so the intermediate conclusions $C_1$ and $C_2$ are constructed and their variables normalized. $B_2' = B_2\theta$ where $\theta = \{x_0 \rightarrow x_1\}$ is variable renaming substitution. The circled literals are the resolved upon literals. $C_1$ is a far parent (indicated by the dashed line) but it is shown to the right of $C_2$ to emphasize the fact that it is a constructed clause.

When DCC is not used, intermediate conclusions are constructed and the substitution sets are discarded because they are not needed anymore. However, when DCC is used, intermediate conclusions are generally not constructed. The substitution sets are combined. The combination is performed as a union instead

of a composition for a very important reason. The reason is to be able to extract the information from the substitution sets in order to construct a delayed clause. **Figure 3-2** performs the same derivation as in **Figure 3-1** but with DCC. The reason why the mgu's must be combined as a union and not as a composition of substitution sets is demonstrated in **Figure 3-3** (see section 3.3).

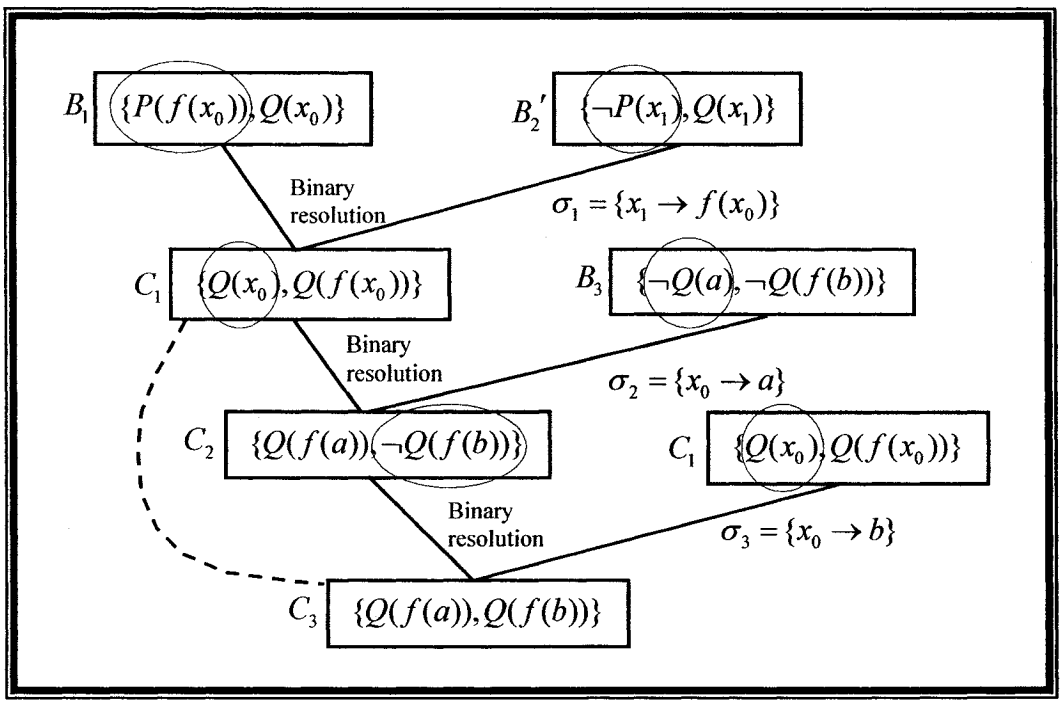



**Figure 3-1: An example of a linear derivation without DCC.**

In **Figure 3-2** the initial clause of the linear derivation and side premises are labeled $B_1^1, B_2^2, B_3^3, C_1^4$ instead of $B_1, B_2, B_3, C_1$ because they are variants of $B_1, B_2, B_3, C_1$. $B_1^1, B_2^2, B_3^3, C_1^4$ are determined as follows: $B_1^1 = B_1\theta_1$, $B_2^2 = B_2\theta_2$, $B_3^3 = B_3\theta_3$, $C_1^4 = C_1\theta_4$, where $\theta_1 = \{\}$, $\theta_2 = \{x_0 \rightarrow x_1\}$, $\theta_3 = \{\}$, and $\theta_4 = \{x_0 \rightarrow x_2\}$ are variable renaming substitution sets. The intermediate conclusions are kept as non-constructed clauses. $\ddot{C}(I_1)$ in **Figure 3-2** corresponds to $C_1$ in **Figure 3-1**. Since it is a far parent it is constructed and labeled $C_1^4$.

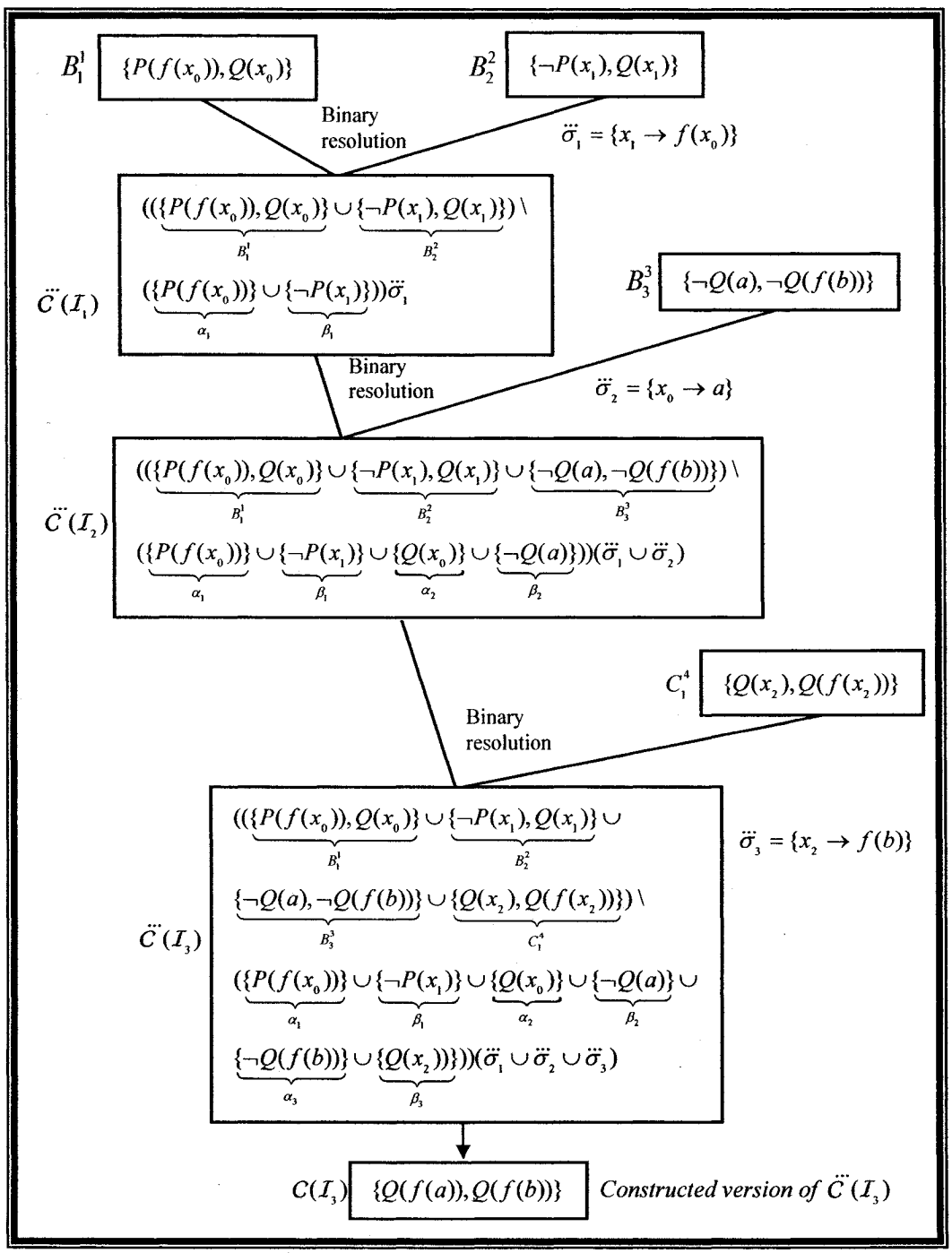



**Figure 3-2: An example of a linear derivation using DCC.**

# 3.3 P-idempotent substitution sets in DCC

In this section we explain why it is necessary in DCC to keep the maintained single substitution set as a union rather than a composition of the mgu's resulting from the application of the inference rules.

Suppose that during a linear deduction process the intermediate conclusions are kept as non-constructed clauses, and the accumulation of mgu's are stored as composition of substitutions rather than a union of substitutions. If an ATP decides at depth $k$ to construct an intermediate conclusion that was generated at depth $i < k$ (i.e., a delayed clause) it would be impossible without the need to perform the linear derivation again. This is demonstrated in **Figure 3-3**.

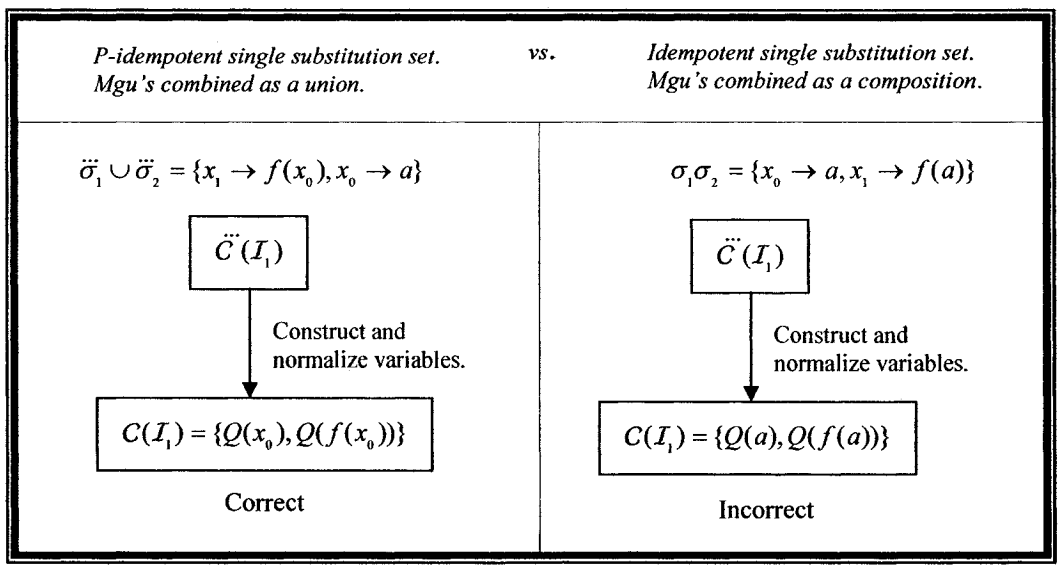



**Figure 3-3: An example demonstrating the problem with the composition of substitutions as opposed to the union of substitutions when delayed clauses are constructed.**

In **Figure 3-3** the idempotent version of the substitution set from Example 3.8 did not work because it was not possible to retrieve the subset $\sigma_1$ from $\sigma_1\sigma_2$. When the composition $\sigma_1\sigma_2$ was formed, the separation of the substitution sets was lost.

In a sense, the substitution terms were blended. On the other hand, the p-idempotent version, which is the union of the mgu's, of the maintained substitution set does not have this problem. $\sigma_1$ can be easily extracted from $\ddot{\sigma}_1 \cup \ddot{\sigma}_2$. All that is needed is to label the elements of $\ddot{\sigma}_1 \cup \ddot{\sigma}_2$ so that they can be identified as subsets of $\sigma_1$ or $\sigma_2$. There are several ways to implement the labeling system. For example, a tag with the depth at which the mgu is formed can be attached to each substitution term as shown in **Figure 3-4** (top). Another possibility is to maintain an ordered set and mark the beginning and end of each subset as shown in **Figure 3-4** (bottom). Those are simple direct and easy to implement methods.

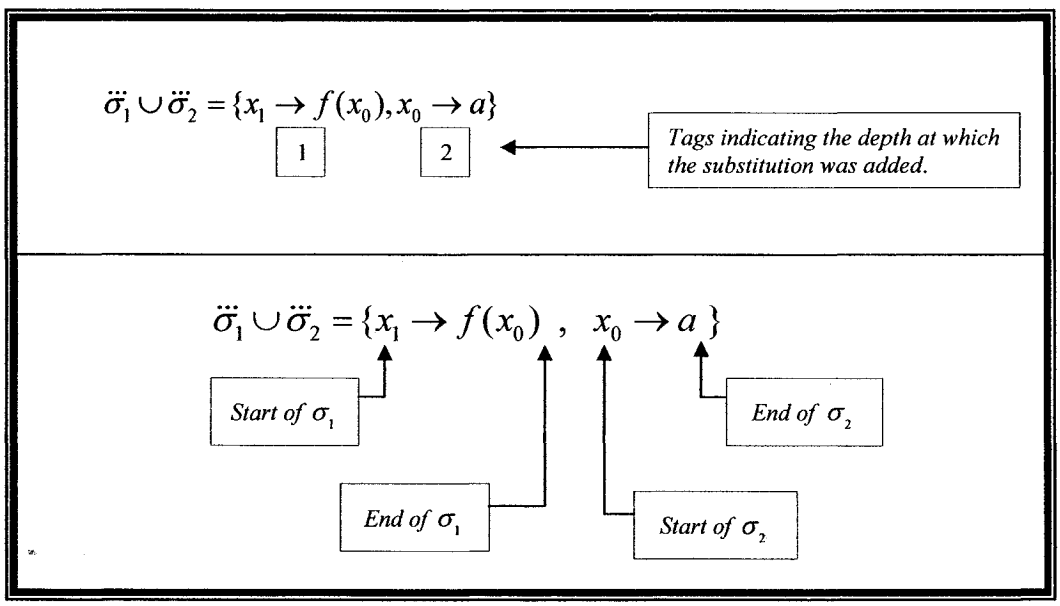



**Figure 3-4: Possible implementations of the p-idempotent set $\ddot{\sigma}_1 \cup \ddot{\sigma}_2$ that make it easy to extract $\ddot{\sigma}_1$ and $\ddot{\sigma}_2$.**

Another advantage of maintaining a p-idempotent set of the union of the mgu's over an idempotent set of the composition of the mgu's is the ability to backtrack efficiently in a depth-first search algorithm. When backtracking from depth $k$ to

depth $k-1$, for $k>1$, then all that is needed is the deletion of the substitution terms that were added at depth $k$. An efficient implementation, such as the *start-end* labelling (see **Figure 3-4**), can perform the deletion operation in constant time. Backtracking is not limited only to the previous depth but to any lower depth. Suppose we want to backtrack to depth $j>1$ from $k>j$, and suppose the maintained p-idempotent substitution set is $\ddot{\sigma}_{1..k}$, then all we have to do is remove $\ddot{\sigma}_{j+1..k}$ from $\ddot{\sigma}_{1..k}$ and we will be back to $\ddot{\sigma}_{1..j}$.

# 3.4 Mega-Inference Rule (MIR)

The mega-inference rule is a direct consequence of DCC. From Theorem 3.5 and Corollary 3.2 we conclude that in a derivation $\Delta$ of length $k$, every intermediate conclusion $\ddot{C}(\mathcal{I}_i)$, $1 \le i \le k$, can be expressed as (see Appendix B)

$$\ddot{C}(\mathcal{I}_i) = ((B^1_{r_1} \cup \cdots \cup B^m_{r_m}) \setminus (\alpha_{1..i} \cup \beta_{1..i}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i}),$$

where

- $m$ is the total number of variants of clauses from a set of constructed clauses $S$ used in $\Delta$,
- for all $1 \le j \le m$, $r_j \in \{1,\ldots,n\}$, $n = |S|$, and $B^j_{r_j} = B_{r_j}\theta_j$, where $B_{r_j} \in S$, and $\theta_j$ is variable renaming substitution, such that

$$Ran(\theta_j) \cap (\bigcup_{q=1}^{j-1} Vars(B^q_{r_q}) \cup Vars(C_{init})) = \{\},$$

- $\alpha_{1..i} = \alpha_1 \cup \cdots \cup \alpha_i$, where $\alpha_1 \subseteq C_{init}$ and for all $1 \le j \le i$,
- $\alpha_j \subseteq \mathcal{L}(\mathcal{D}(\mathcal{I}_1) \cup \cdots \cup \mathcal{D}(\mathcal{I}_{j-1}))$,
- $\beta_{1..i} = \beta_1 \cup \cdots \cup \beta_i$, where for all $1 \le j \le i$,
- $\beta_j \subseteq \mathcal{L}(\mathcal{D}(\mathcal{I}_j))$,
- $\sigma_{1..i} = \sigma_1 \cdots \sigma_i$ and $\tau_{1..i} = \tau_1 \cdots \tau_i$.

Let $\gamma_{1..k}$ be the conjunction of all the conditions of all the inference rules applied in the derivation $\Delta$, then we can write the mega-inference rule as

$$\frac{B^1_{r_1} \cdots B^m_{r_m}}{((B^1_{r_1} \cup \cdots \cup B^m_{r_m}) \setminus (\alpha_{1..k} \cup \beta_{1..k}))\sigma_{1..k}(\tau_{1..k}\sigma_{1..k})}, \text{ if } \gamma_{1..k}.$$

# 3.5 Summary

In this chapter we presented a formal treatment of delayed clause-construction. We defined the terms, *term replacement list*, *p-idempotent substitution*, *constructed* and *non-constructed* clause, and *delayed clause*. We showed that in a linear derivation every intermediate conclusion can be expressed in terms of constructed clauses, a single p-idempotent substitution set, and single term replacement list and thus there is no need to construct it. We showed that, in order to be able to construct a delayed clause at a later time and to perform efficient backtracking, the substitution set must be the union of p-idempotent mgu's resulting from the inference rules applied in a linear derivation. We discussed the three cases in which intermediate conclusions must be constructed. The cases occur when:

- an intermediate conclusion is a far parent or a *from* clause (e.g., paramodulator),
- an intermediate conclusion satisfies certain criteria based on heuristics that indicate the clause is worth constructing, or
- the time needed to construct an intermediate conclusion and determine its attributes is less than the time needed to determine its attributes without constructing it.

We listed and discussed in detail the conditions required to obtain a sound linear derivation using DCC. We derived a general formula for expressing an intermediate conclusion. Finally, we gave a definition of a mega-inference rule.

CHAPTER

# 4

# Semi-Linear Resolution

In modern ATPs, the most commonly used bottom-up approach is based on a best-first search algorithm called the *given-clause* algorithm, and the most commonly used top-down approach is based on an *iteratively deepening depth-first search* algorithm.

In this chapter, we begin with an overview of the research done on combined top-down bottom-up search procedures and briefly point out the similarities and differences between our work and the work that has been done. We then describe briefly the given-clause algorithm and the iteratively deepening depth-first search algorithm. We present the main advantages and disadvantages of each algorithm. We introduce and discuss in detail semi-linear resolution which is an iteratively-deepening depth-first search that shares some of the advantages enjoyed by the given-clause algorithm such as redundancy elimination and simplification rules. Semi-linear resolution implements DCC as a mega-inference rule and uses attribute sequences (discussed in Chapter 5) to reduce the explorable search space. Finally, we list the conditions that must exist in order for semi-linear resolution to be refutation complete.

## 4.1 Overview of Top-Down Bottom-Up Approaches

A top-down approach recursively breaks down a goal into subgoals until eventually the subgoals can be proven immediately by a given set of clauses or by

derived clauses obtained during the search process. A bottom-up approach derives clauses from the input set until an inconsistency is reached. The advantage of a top-down approach is that it goal-oriented. Its disadvantage is its insufficient redundancy control [Fuchs & Fuchs 1999]. A bottom-up approach is good in controlling redundancy but lacks goal-orientation.

In [Astrachan & Loveland 1991] a top-down theorem prover, METEOR, which included a bottom-up search through the use of lemmas, revealed the potential of combining the two approaches. It provided a certain amount of redundancy elimination which improved the efficiency of the search. Consequently, METEOR was able to prove more theorems with the addition of the bottom-up approach than without it [Astrachan 1992].

Schumann [Schumann 1994] combined top-down with bottom-up approaches by developing a preprocessor, named DELTA, that performed a bottom-up search and generated unit-clauses that are added to the original clauses. He then used SETHEO [Letz et al. 1992] to perform a top-down search. The results showed that the combined approach was able to prove more theorems than SETHEO was able to prove on its own.

[Fuchs & Fuchs 1999] have shown that by combining the two approaches they were able to solve almost twice as many hard problems than either approach could solve alone. They used SPASS [Weidenbach et al. 1999] for the bottom-up search and SETHEO for the top-down search.

VAMPIRE uses a splitting rule to integrate a top-down approach into its bottom-up approach. With the inclusion of this strategy, VAMPIRE solved 98 problems from the TPTP library [Sutcliffe 1994] that it couldn't solve without the splitting rule [Riazanov 2003].

SLR shares similar characteristics with the approach used in METEOR. SLR generates goals and adds them to the input set of clauses and then uses them to control redundancy. However, instead of relying only on Model Elimination (ME) [Fleisig et al. 1974], [Loveland 1969], as in METEOR, SLR can be used with

different calculi. SLR also shares (to a certain extent) the best-first search strategy of the given-clause algorithm. SLR selects the "best" initial clauses first and then conducts a depth-first search. SLR differs from the methods used by Schumann and Fuchs in that SLR generates goals, which are similar to lemmas, dynamically like METEOR, and adds them to the original set of clauses, rather than going through a preprocessing phase first. In addition, SLR depends on a mega-inference rule to perform large steps and uses attribute sequences to reduce the explorable search space.

## 4.2 The Given-Clause Algorithm (GCA)

The given-clause algorithm is a best-first search algorithm that selects the "best" clause, called the **given clause**, based on heuristics, and then infers all clauses from the given clause using a special set of clauses, called the **active** set. The general version of the algorithm is shown in **Figure 4-1**. We assume that the parameters are passed by value to the procedure GIVENCLAUSEALGORITHM. *S* is the input set of clauses and *I* is the inference rules that the given-clause algorithm will use to infer new clauses. The set *I* is either selected by a user or automatically selected by the ATP.

The variables *active*, *passive*, and *inferred* each is a set of clauses. The variable *given_clause* is one clause. Initially *passive* is *S* (line 3) and *active* is empty (line 4). A test is performed at line 5 to check if the empty clause is in *S*. If no empty clause is found the loop starts.

```
GIVENCLAUSEALGORITHM(S:clauses, I:inference_rules)
1.  given_clause:clause
2.  inferred, active, passive:clauses
3.  passive := S
4.  active := {}
5.  if φ in S then return "unsatisfiable"
6.  while passive ≠ {} do
7.      given_clause := SELECTCLAUSE(passive)
8.      REMOVECLAUSE(given_clause, passive)
9.      ADDCLAUSE(given_clause, active)
10.     inferred := INFER(given_clause, active, I)
11.     if φ in inferred then return "unsatisfiable"
12.     MOVECLAUSES(inferred, passive)
13. end while
14. return "passive is empty"
```

**Figure 4-1: A given clause algorithm.**

A clause is selected from *passive* and assigned to *given_clause* (line 7). The selected clause is then removed from *passive* (line 8) and added to *active* (line 9). In the procedure INFER (line 10), all inference rules in *I* are applied to *given_clause* and the rest of the clauses in *active*, such that every time a rule is applied, *given_clause* is one of the premises. All the conclusions resulting from the application of the inference rules in INFER are gathered in *inferred*. A test is made to find out if the empty clause was obtained (line11). If the empty clause is not in *inferred*, then the clauses in *inferred* are moved to *passive*. The loop continues until *passive* is empty or the empty clause is obtained.

There are two commonly used variations of the GCA, the OTTER loop [McCune 2003] and the DISCOUNT loop [Avenhaus et al. 1995], [Denzinger et al. 1997]. They both add simplification rules to the GCA presented in **Figure 4-1**. However, the difference between the two lies in the time at which the simplification rules are applied and on the sets of clauses to which they are applied to. A comparison between the two is given in [Riazanov 2003].

Some state-of-the-art ATPs that use the given-clause algorithm are E [Schulz 2002], GANDALF [Tammet 1997], OTTER [McCune 2003], SPASS [Weindenbach et al. 1999], VAMPIRE [Riazanov 2003], and WALDMEISTER [Hillenbrand et al. 1997].

### 4.2.1 Cases when GCA is not refutation complete

The given-clause algorithm as presented in **Figure 4-1** is refutation complete if the following conditions exist (each condition is explained in detail in the following sections).

- The selection procedure SELECTCLAUSE is fair.
- The inference rules in $I$ form an inference system that is refutation complete.

#### 4.2.1.1 Fairness of the selection procedure

A GCA is said to be **fair** if the selection procedure SELECTCLAUSE (line 7) ensures that every clause in *passive* will eventually be selected.

The set *passive*, in practice, usually contains enough clauses to keep the loop running for a very long time (possibly an infinitely long time) if the empty clause is not obtained. The selection procedure SELECTCLAUSE selects a clause based on heuristics. If the heuristics do not provide a "fair" selection, then a clause in *passive* may never be selected. If this unselected clause is necessary to obtain a refutation, then the GCA will never derive the empty clause. This implies that an ATP employing GCA that is using an "unfair" selection process is not refutation complete.

#### 4.2.1.2 Refutation completeness of the inference system

The refutation completeness of an inference system depends on the inference rules (Definition 2.33). If $I$ is a set of inference rules that are not refutation complete, then GCA is not refutation complete.

## 4.3 Iteratively-Deepening Depth-First Search

An iteratively-deepening depth-first search [Korf 1985], [Stickel & Tyson 1985], combines the advantages of breadth-first search (BFS) and depth-first search (DFS). In BFS, all possible clauses that can be generated by an implemented set of inference rules at each depth are generated before moving to the next depth. This guarantees a shortest proof possible. However, due to the large number of clauses that can be generated, the use of BFS requires a lot of storage space. The memory requirements for BFS grow exponentially with the depth. In DFS, storage space is a not a problem. The memory requirements for DFS grow linearly with the depth because DFS explores every branch up to the point where no more clauses can be generated before moving to the next branch. This implies that DFS does not guarantee a shortest proof. Another more important problem with DFS is that if a branch extends indefinitely, then DFS can continue exploring this branch forever. Therefore, an ATP using DFS can get stuck on a single branch that extends indefinitely without reaching a refutation even though the input clauses are unsatisfiable. This implies that an ATP using DFS is not refutation complete. With IDDFS a bound is set for each iteration to force the DFS along a branch to backtrack once the bound is reached. This solves the problem of exploring infinitely long branches indefinitely. The bound increases by one with every iteration, so IDDFS guarantees the shortest proof because it generates all clauses that can be derived at each depth. The downside with IDDFS is the repetitive derivation of clauses from lower levels, because every time the bound is increased, clauses derived at depths lower than the bound are derived again. However, in practice, the number of clauses increases exponentially with every iteration so the number of clauses that are repeatedly derived at lower depths is relatively small by comparison to the total number of clauses derived within an iteration. An upper bound on the number of repetitions is calculated in [Korf 1985] to be $(b/(b-1))^2$, where $b$ is the branching factor.

A general IDDFS algorithm is shown on **Figure 4-2**. The IDDFS algorithm shown in **Figure 4-2** is recursive. The main procedure IDDFS calls DFS and DFS calls itself until either the depth exceeds the bound (line 4) or the empty clause is obtained (line 6). *S* is the set of input clauses and *I* is the set of inference rules selected by a user or automatically selected by an ATP.

```
IDDFS(S:clauses, I:inference_rules, max_bound:integer)
1.  input_clause:clause
2.  bound:integer
3.  rule:inference_rule

4.  for bound := 1 to max_bound
5.      for each input_clause in S
6.          for each rule in I
7.              DFS(input_clause, S, I, 1, bound, rule, {})
8.          end for
9.      end for
10. end for
```

```
DFS(C:clause, S:clauses, I:inference_rules, depth:integer, bound:integer,
    rule: inference_rule, ancestors: clauses)
1.  inferred: clauses
2.  new_clause: clause
3.  new_rule:inference_rule

4.  if depth > bound then return
5.  inferred := INFER(C, S ∪ ancestors \ C , rule)
6.  if φ in inferred then TERMINATE("unsatisfiable")
7.  for each new_clause in inferred
8.      for each new_rule in I
9.          DFS(new_clause, S, I, depth+1, bound, new_rule, ancestors ∪ C )
10.     end for
11. end for
```

**Figure 4-2: An IDDFS algorithm.**

The algorithm picks one clause from $S$ in the order the clauses are stored and performs a depth-first search starting with this clause (line 7 in IDDFS procedure). A depth-first search implies that all derivations are linear. The condition at line 4 in the procedure DFS ensures that the depth-first search does not exceed the iteration bound. The procedure INFER applies the inference rule *rule* with $C$ as the main premise. All possible conclusions that can be formed from $C$ and the clauses in the set $S \cup ancestors \setminus C$ using the inference rule *rule* are stored in the set *inferred*. Each clause in *inferred* is an intermediate conclusion generated at depth *depth*. Each clause in *inferred* is then used as a main premise for the next application of an inference rule (lines 7-11 in DFS). A depth-first search is performed from each clause in *inferred* using all inference rules in $I$.

The search continues until either the empty clause is obtained or the *max_bound* is reached. If the empty clause is obtained, then the search terminates and the string "unsatisfiable" is displayed (line 6 in DFS).

IDDFS as presented in **Figure 4-2** is refutation complete if *max_bound* can be set to infinity. If *max_bound* is chosen too small, then the search may terminate before it finds a refutation. The inference rules binary resolution and binary factoring form the resolution calculus which is refutation complete [Loveland 1978], [Robinson 1965].

Some of the current ATPs that use an iteratively-deepening depth-first search are METEOR [Astrachan & Loveland 1991], PROTEIN [Baumgartner & Furbach 1994], PTTP [Stickel 1984], [Stickel 1992], SETHEO [Letz et al. 1992], and THEO [Newborn 2001].

## 4.4 Comparison between GCA and IDDFS

The main advantages of GCA over IDDFS are:

- Assuming that well tuned heuristics are used to select the "best" clause, then compared to depth-first search, best-first search can lead to a refutation in a shorter period of time. This has been confirmed from the results of CASC [CASC site].
- It is easy to add simplification rules because derived clauses exist in memory and can be can be simplified using rewrite rules, subsumption and tautology elimination; thereby reducing redundancy.
- There are no repetitive computations from lower depths like in IDDFS.
- Because derived clauses exist in memory, it is easier in a GCA to add efficient retrieval techniques, such as term indexing, of potentially unifiable terms. This reduces the number of unsuccessful unifications and matching failures; thereby, improving the efficiency of a theorem prover.

The main disadvantages of GCA with respect to IDDFS are:

- Storage requirements. Since derived clauses have to be stored, and generally there a large number of derived clauses, large memory capacities are required. Secondary storage, such as hard drives, can be used, but their slower access time can reduce the speed of the theorem prover substantially especially when simplification rules and term indexing have to performed on the secondary storage. To reduce the number of retained clauses, a number of strategies are used. For instance the weight limit and memory limit strategies are used in OTTER, and the limited resource strategy is used in VAMPIRE. Those strategies are effective in reducing the memory requirements but they compromise completeness in general[1].
- Redundancy control is an expensive procedure. The more clauses there are, the slower is the redundancy control process. A slow redundancy control

[1] According to [Riazanov & Voronkov 2000], any theorem that can be proved in VAMPIRE in time $t$ without using the limited resource strategy, the same theorem can be proved by VAMPIRE using the limited resource strategy in time less or equal to $t$. This does not imply that the limited resource strategies maintains completeness, but it ensures, at least in principle, that if a theorem can be proved without it, then the theorem can still be proved with it, provided that VAMPIRE is given the appropriate time limit for this theorem.

process can affect the performance of an ATP. According to [Riazanov & Voronkov 2000] when the number of retained clauses exceeds 100,000, it becomes very difficult to manage them efficiently even with state-of-the-art term indexing techniques. In GCA derived clauses are retained rather than derived again as in IDDFS. This usually leads to a retention of a large number of derived clauses which, consequently, can slow down the redundancy control process.

- The need to find good heuristics to select, if not the best clause, something close to the best. In order to build good heuristics, a lot of analysis (theoretical and experimental) should be done. Furthermore, a set of heuristics may work well on a category of theorems but not as well on another category of theorems (see Appendix A for a list of categories).
- The non-linearity of derivations performed by GCA does not allow for specific implementation optimizations such as the use of DCC. Every generated clause is constructed and then subjected to simplification rules and subsumption tests. If the clause passes the tests it is retained otherwise it is discarded. So if the implemented inference rules take small steps, such as binary resolution, then a lot of clauses may be constructed and discarded as shown in [Wos et al. 1992].

## 4.5 Semi-Linear Resolution (SLR)

Semi-linear resolution is mainly an IDDFS with some of the advantageous ideas from the best-search first included. It relies on DCC in the form of a mega-inference rule. The main SLR algorithm is shown in **Figure 4-3**, and the mega-inference rule part is shown in **Figure 4-4**. The list of parameters (in alphabetical order) with a brief description of each is given in **Table 4-1**. The parameters *S*, *I*, and *max_bound* are passed by value.

```
SLR(S:clauses, I:inference_rules, max_bound:integer)

1.  C_init:clause
2.  bound:integer
3.  S_factors, Goals, initial_clauses:clauses

4.  Goals := {}
5.  S_factors := FACTORS(S)
6.  for bound := 1 to max_bound
7.      initial_clauses := SELECTINITIALCLAUSES( S ∪ S_factors ∪ Goals )
8.      for each C_init in initial_clauses
9.          Goals := MIR(C_init, 1, bound, S ∪ S_factors ∪ Goals , {},
                         Goals, I)
10.     end for
11. end for
```

**Figure 4-3: An SLR algorithm.**

```
MIR(C:clause, depth:integer, bound:integer, potential_side_premises:clauses,
      T:clauses, goals:clauses, rules:inference_rules)

1.  side_premises, inferred:clauses
2.  applicable_rules:inference_rules
3.  rule:inference_rule
4.  new_clause:clause

5.  if depth > bound then return(goals)

6.  side_premises :=
        SELECTSIDEPREMISES(potential_side_premises ∪ goals ) ∪ T
7.  applicable_rules := SELECTAPPLICABLERULES(rules, C, side_premises)
8.  for each rule in applicable_rules
9.      inferred := INFER(C, side_premises, rule)
10.     if φ in inferred then TERMINATE("unsatisfiable")
11.     for each new_clause in inferred
12.         if PASSEVALUATION(new_clause) then
13.             CONSTRUCT(new_clause)
14.             ADDCLAUSE(new_clause, goals)
15.         end if
16.         if MERGECLAUSE(new_clause) then
17.             CONSTRUCT(new_clause)
18.             ADDCLAUSE(new_clause, T)
19.         end if
20.         goals := MIR(new_clause, depth+1, bound, side_premises, T,
                         goals, rules)
21.         if MERGECLAUSE(new_clause) then
22.             DELETECLAUSE(new_clause, T)
23.         end if
24.     end for
25. end for
26. return(goals)
```

**Figure 4-4: An MIR algorithm.**

**Table 4-1: List of variable parameters used in an implementation of SLR**

| Variable parameters | Description |
|---|---|
| *applicable_rules* | A subset of *I*. This set contains rules that are relevant to the clauses *C_init* and *side_premises*. For example, paramodulation is not applicable unless either *C_init* or one of the *side_premises* can be used as a paramodulator. |
| *bound* | This is the iteration depth bound. |
| *C_init* | Initial clause in a linear derivation. |
| *Goals* | Set of derived clauses that conform to criteria either set by a user or by an ATP. |
| *I* | Set of inference rules chosen by a user. |
| *inferred* | The set of all clauses that are inferred from the clause *C* and *side_premises* using the inference rule *rule*. |
| *max_bound* | The maximum iteration depth. |
| *potential_side_premises* | The set of clauses from which the side premises to be used in the mega-inference rule are selected. |
| *S* | Set of input clauses. |
| *S_factors* | Factors of the set of input clauses. |
| *side_premises* | Chosen clauses that can be used as side premises in the mega-inference rule. |
| *T* | Temporary set of constructed clauses used only during a derivation of a goal clause. For example, merge clauses obtained during the application of the mega-inference rule are constructed and added to this set but are deleted once the application of the mega-inference rule is over. |

A list of the procedures/functions (in alphabetical order) and a brief description of the functionality of each procedure/function is given in **Table 4-2**. Detailed description of the procedures SLR and MIR are given later.

**Table 4-2: A list of procedure/functions used in an implementation of SLR**

| Procedure/Function | Description |
|---|---|
| ADDCLAUSE | Adds a clause to a set of clauses. |
| ADDCLAUSES | Performs the union of two sets of clauses and returns the result in the second parameter. |
| CONSTRUCT | Constructs a clause. |
| MERGECLAUSE | Determines if the clause is a merge clause or not. Returns **true** if the clause is a merge clause and **false** otherwise. |
| MIR | Application of the mega-inference rule performed as a depth first search. |
| PASSEVALUATE | Determines if a clause conform to certain criteria set by the user or automatically determined by an ATP. Returns **true** if the clause conforms to the criteria and **false** otherwise. |
| SELECTAPPLICABLERULES | Selects applicable rules for *C_init* and *side_premises*. |
| SELECTINITIALCLAUSES | Selects initial clauses that have the potential to lead to a refutation. |
| SELECTSIDEPREMISES | Selects side premises that have potential complementary literals or can be used as paramodulators. |

| Procedure/Function | Description |
|---|---|
| SLR | Main procedure of semi-linear resolution. Performs an IDDFS using delayed-clause construction and adds some redundancy control. |

We now explain the SLR procedure and MIR function in detail.

SLR

Line 4 initializes the set *Goals*. This is the set that contains the conclusions of the mega-inference rule as long as the conclusion conforms to certain criteria set by the user or automatically determined by an ATP. Examples of goal clauses are given in Chapter 3. The purpose of this set is similar to lemmas and caching in METEOR except that it does not affect the refutation completeness of SLR. *Goals* can be viewed as a set whose functionality with respect to SLR is similar to the *passive* set in GCA.

Line 5 calculates the factors of the input clauses. This is necessary to maintain the completeness of SLR since input clauses are not factored before they are used as side premises.

Lines 6-11 is the main IDDFS loop.

Line 7 selects the initial clauses for a linear derivation. The selection criteria are based on heuristics that determine the "best" clauses to start with. The selection is also based on the values given in attribute sequences (presented in Chapter 5). Attribute sequences are used to prune the explorable search space. The initial clauses are selected from the input clauses, the set of factors of the input clauses and the set *Goals* which consists of derived constructed clauses. Initially the set

*Goals* is empty, but after each application of the mega-inference rule (line 9), the size of *Goals* increases when derived clauses conforming to criteria that are either set by a user or automatically determined by an ATP are constructed and added to *Goals*. The addition of new clauses to *Goals* is done in line 14 of the function MIR.

MIR

Line 5 checks if the depth has exceeded the bound and if so, it returns the set *goals* unchanged.

Line 6 selects the side premises by using lookup tables (see Chapter 6) and attribute sequences (see Chapter 5). Lookup tables help determine clauses that have potential complementary literals for resolution and unifiable literals for factoring. They also help in determining unifiable terms, instances of terms, or generalizations of terms between the initial clause and the side premises. The addition of the set $T$ as a union with the selected premises rather than as a union with the parameters of the function SELECTSIDEPREMISES ensures that ancestor clauses, such as merge clauses or *from* clauses, are added to the set of side premises. The selection of side premises is important because it influences the selection of applicable inference rules (line 7).

Line 7 selects applicable inference rules. Those are the rules that are relevant to the initial clause and the selected side premises. For instance if paramodulation is in *rules* but none of the side premises or the initial clause can be used as a paramodulator then there is no point in choosing paramodulation. In this case, paramodulation is not an applicable rule. Also attribute sequences (see in Chapter 5) affect the selection of the applicable rules as well as the number of applications of each rule by relying on the length of a clause.

Lines 8-25 is the main depth first search loop in which the application of the selected inference rules is performed in all possible ways at all depths up to the depth bound. If the application of an inference rule is successful, then the conclusions are returned in the set *inferred* (line 9). If one of the conclusions is the empty clause, then the search terminates and the string "unsatisfiable" is displayed (line 10). If the empty clause is not obtained, then every clause in the set *inferred* is evaluated with respect to the criteria that are either set by a user or automatically determined by an ATP (line 12). If the clause passes the evaluation, it is constructed (line 13) and added to the set of derived goals (line 14).

Forward subsumption, where if a clause in *S*, *S_factors*, or *Goals* subsumes *new_clause* then *new_clause* is not added, is performed within the procedure PASSEVALUATE. Back subsumption, where if *new_clause* subsumes a clause in *S*, *S_factors*, or *Goals* then the subsumed clause is deleted, can also be performed at this point.

Lines 16-19 determine if a conclusion is a merge clause and if so, it is added to the temporary set *T*. *T* is a set of constructed intermediate conclusions that are generally used as far parents, such as merge clauses and *from* clauses. However, there is a case where an intermediate conclusion is constructed and added to *T* and is not necessarily a far parent. This case is explained below as Case (2). Clauses in *T* are short lived clauses. They are useful within a linear derivation but once the end of a derivation is reached (see Chapter 3 for the definition of end of derivation), they are deleted. There are two cases when it is necessary to construct an intermediate conclusion and added to *T* during a linear deduction. The two cases occur when

(1) an intermediate conclusion is a merge clause, or

(2) the time it takes to extract the attribute of an intermediate conclusion takes longer when the intermediate conclusion is not constructed than when it is constructed.

Those cases are necessary to maintain completeness of SLR and the speed improvement gained by the employment of DCC. In **Figure 4-4** we show only the case of merge clauses to simplify the algorithm. It is easy to add a condition that tests for Case (2). We discuss each case in detail.

Case (1)

In [Andrews 1968] and [Anderson & Bledsoe 1970], it was shown that the resolution method remains refutation complete if ancestor resolutions are restricted to only resolutions with ancestor clauses that are merge clauses. Corollary 3.2 states that every clause that is a far parent must be constructed. A merge clause is a far parent, and thus must be constructed. If SLR is limited to constructing merge clauses and adding them to the set *T*, then SLR remains refutation complete. However, every time a clause is constructed, it is no longer a delayed clause. So if the number of merge clauses is large, then too many clauses would be constructed. This implies that only the construction of few clauses is delayed. The performance improvement of an ATP resulting from the employment of DCC relies on the number of delayed clauses; the more the delayed clauses the better the performance. Therefore, if the construction of only a few clauses is delayed, then the use of DCC leads to a negligible boost in performance. If the worst case occurs, where every clause is a merge clause, then DCC becomes useless.

We tested 2323 theorems to determine the percentage of merge clauses with respect to the total number of generated clauses. We found that, on average, only 0.57% of all generated clauses are merge clauses and none of the tests revealed a worst case scenario. The results are shown in **Figure 4-5**.

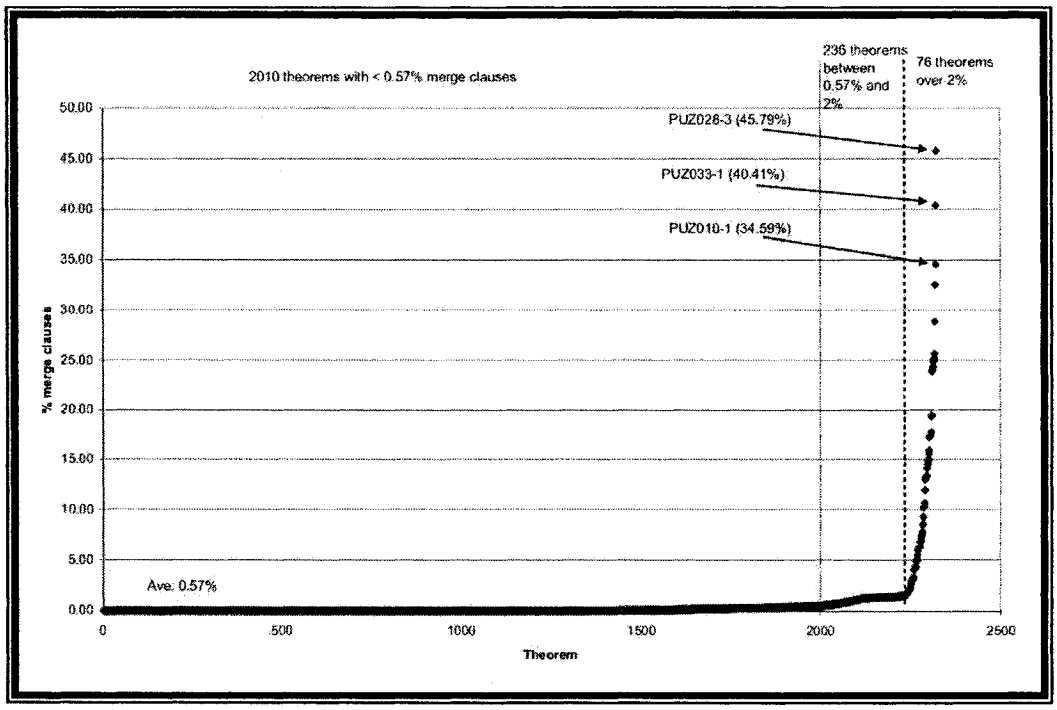



**Figure 4-5: Percentage of merge clauses with respect to the total number of generated clauses per theorem.**

The highest value is 45.73% and the lowest is 0%. Higher percentages of merge clauses normally occur in theorems where the maximal term depth is one and most of the terms are constants. For instance, PUZ028-3 has zero variables and a maximal term depth of one, PUZ033-1 contains only propositional clauses, and most of the input clauses of PUZ010-1 contain zero or one variable and their maximal term depth is one. The reason why merge clauses arise more in theorems where shallow ground terms occur often is because if the unification of literals with shallow terms is successful then it is more likely to produce an mgu that is an empty substitution set. Therefore, identical instances of literals are more likely to arise.

Since factoring of identical literals is the same as merging the identical literals, then there is only the need to check for merge clauses after performing a binary factoring. If the unification of two literals yields an mgu which is an empty

substitution set, then the literals are identical and the factor is a merge clause. A merge clause is constructed and added to set *T*.

We conclude from the result of the experiments that the percentage of merge clauses to the total number of generated clauses is low enough in general that constructing them does not affect the performance improvement gained by using DCC.

Case (2).

When an intermediate conclusion is generated, it is useful to determine its attributes in order to decide whether to continue along the path of this clause or to backtrack. For instance, an intermediate conclusion may be subsumed by a clause in *Goals*. In this case, there is no point in proceeding with the search. Clause attributes can also be used as part of heuristics upon which the decision of proceeding along a path or not is made. For instance, THEO [Newborn 2001] uses an extended search strategy, in which the search continues beyond the iteration bound if an intermediate conclusion satisfies certain criteria. The criteria are based on clause attributes. Therefore, clause attributes are important for an ATP to make a decision that may lead to the pruning of the explorable search space. When DCC is employed, determining an intermediate conclusion's attributes is not a straight forward process because an intermediate conclusion is generally not constructed. The attributes have to be determined from the expression that represent an intermediate conclusion. Generally, the time it takes to determine a non-constructed clause attributes is short enough that is does not affect the performance improvement gained by using DCC. However, it is sometimes better to construct an intermediate conclusion and then determine its attributes rather than determine its attributes in its non-constructed state. We derived a formula that an ATP can use to determine whether it is faster to construct an intermediate conclusion and then determine its attributes, or to determine its attributes without constructing it. The formula is

$$|\ddot{\sigma}_{1..i}| \leq \frac{-|C(I_i)| + \sqrt{|C(I_i)|^2 + 4 \cdot Weight(C(I_i))}}{2}, \qquad \textbf{E4.1}$$

where $\ddot{\sigma}_{1..i}$ is a p-idempotent substitution set which forms the union of all mgu's resulting from the inference rules applied up to an including depth $i$, and $C(I_i)$ is the intermediate conclusion at depth $i$. A detailed derivation and analysis of this formula is given in Appendix C. An ATP uses **E4.1** as follows. If the number of elements in $\ddot{\sigma}_{1..i}$ is less than or equal to the value obtained from the right hand side of the formula in **E4.1**, then it is faster to determine an intermediate conclusion's attribute without constructing it. Otherwise, it is faster to construct the intermediate conclusion and then determine its attributes.

## 4.6 Redundancy Control in SLR

Redundancy elimination is performed at the time when a goal is evaluated (line 12 in MIR procedure in **Figure 4-4**). If a new goal clause *new_clause* subsumes clauses from *goals*, then the clauses are removed from *goals* (i.e. backward subsumption), but if *new_clause* is subsumed by a clause in *goals*, then *new_clause* is not added to *goals* (i.e., forward subsumption). Forward and backward demodulation can be performed in the same manner between a *new_clause* and *goals*.

We conducted an experiment in our experimental ATP CARINE to determine the number of redundant goals eliminated due to forward subsumption. We restricted the goals to unit clauses. We ran CARINE over the CNF theorems in TPTP library v.2.6.0 (see Appendix A). We counted the number of generated unit clauses and retained unit clauses in each theorem that was proved by CARINE (see Appendix H

for a list of theorems proved by CARINE). A unit clause is retained if it is not subsumed by an already retained clause. We found that on average less than 20% of the unit clauses generated were retained.

Forward and backward subsumption can also be performed between a new goal and the input set or the factors of the input set without affecting the completeness of SLR, but forward and backward demodulations between the a new goal and the input set can affect the completeness in SLR.

Tautology deletion is performed on every constructed clause whether it is in the temporary set *T* or in *Goals*. Tautology deletion is performed at the time of construction of a clause (lines 13 and 17 in MIR) .

# 4.7 Advantages and disadvantages of SLR

## 4.7.1 Comparison between SLR and GCA

In addition to the advantages of iteratively-deepening depth-first search over the given-clause algorithm, SLR uses DCC which results in a much faster derivation of clauses and hence, SLR can achieve a much higher inference rate than GCA. Furthermore, the disadvantages of the IDDFS (shown in **Figure 5-2**) are not as strong in SLR because of the following additional strategies.

- The selection of the "best" initial clauses (line 8 in SLR).
- The use of the set *Goals* which although is much smaller than either the passive or active sets used in GCA to store derived clauses that can be used in the search. The use of *Goals* reduces the number of repetitive computations. The set *Goals* acts like the set *passive* in GCA at the time of initial clause (line 8 in SLR) selection but acts like the set *active* when the side premises are selected (line 2 in MIR).
- Simplification and redundancy control can be integrated easily into SLR without affecting DCC. However, the range of simplifications and

redundancy control is not as wide as it is in GCA because much fewer derived clauses are retained.

### 4.7.2 Comparison between SLR and Model Elimination

In this section we compare SLR with the Model Elimination (ME) implementation that is used in high performance ATPs such as SETHEO and METEOR. Henceforth, any mention of ME implies the implementation of ME in an iteratively-deepening depth-first search with implementation techniques that result in high inference rate. Since both SLR and ME follow an IDDFS, we focus our comparison on the implementations issues that result in high inference rate and flexibility to include additional strategies and equality rules.

The main difference between SLR and ME is in the way the high inference rate is achieved. An ATP based on SLR achieves its high inference rate through DCC, whereas an ATP based on ME achieves its high inference rates based on either clause compilation following the PTTP approach or on a *data oriented architecture* with a reuse of input clauses as proposed in [Letz & Stenz 2001].

PTTP first appeared in [Stickel 1984]. PTTP solved the weakness found in SLD-resolution of Prolog systems. The weaknesses were lack of completeness for non-Horn formulas, unsound unification, and unbounded depth-first search. In addition, PTTP provided a high inference rate through the compilation of input clauses into procedures of either an actual or abstract machine. The compilation is possible because of the linearity in the derivation of clauses. Since ME does not perform ancestor resolutions (i.e. no far parents are explicitly involved) in a derivation, then only input clauses are used as side premises. Since the contents of input clauses are known before the search starts, the input clauses can be compiled in a way that will make the application of the *extension* and *reduction* rules of ME performed efficiently. This leads to the integration of the input clauses with the search process resulting in a tight relation between the two. The

drawback resulting from this tight relationship was reported in [Letz & Stenz 2001 p.2086] as follows

"Changing the unification such as to add sorts, for example, or adding new inferences, e.g. equality handling, or generalizing the backtracking procedure becomes extremely cumbersome if not impossible in such an architecture."

Unlike PTTP, in a data oriented architecture clauses are viewed as data structures that are separate from the search process which allows an easy integration of new strategies and the addition of new inference rules, simplification rules, and redundancy control. This is similar to SLR but instead of binary resolution and binary factoring, the extension and reduction rules are used and instead of DCC, a *reuse* of copies of input clauses is performed. The motivation behind clause reuse is the same as that of DCC, namely, the time consuming operations for constructing clauses. Input clauses are copied (i.e., constructed) and their variables renamed, then they are used as side premises. When backtracking is performed, a clause which was used as a side premise is not deleted but only its instantiated variables are *de-instantiated.* The next time this input clause is needed as a side premise, one of its copies that is still remaining in memory is used. This reduces the number of clause constructions which ultimately leads an ATP to achieve a high inference rate. The drawback is the memory requirement. Since the copies of side premises remain in memory after backtracking, then the reuse approach requires more memory than DCC. Furthermore, DCC does not waste time making copies. Therefore, SLR which relies on DCC uses memory more efficiently than the reuse approach, and it performs less operations in generating new clauses because it does not make copies of input clauses.

## 4.8 Completeness of SLR

If the selection of initial clauses is fair (i.e., any clause or its factor from the input clauses eventually will be selected) and the inference rules are refutation complete, then SLR will behave like linear-input resolution performed in an iteratively-deepening manner. If only merge clauses are added to the set *T*, then SLR will behave like linear resolution. The main difference between SLR and linear resolution is the set *Goals*. If *Goals* is left empty, then SLR will become a linear resolution performed in an iteratively-deepening manner. The set *Goals* is used to reduce redundancy and repetitive derivations. Therefore, it does not affect the completeness of SLR. Since linear resolution is refutation complete, then so is SLR.

We conclude that SLR is refutation complete if

- the procedure SELECTINITIALCLAUSES is fair, and
- the selected inference rules *I* are refutation complete, and
- merge clauses are constructed and used in ancestor resolutions, and
- *max_bound* is infinity (since SLR is an IDDFS).

## 4.9 Summary

In this chapter we described the given-clause algorithm and iteratively-deepening depth-first search algorithm briefly. We listed the advantages and disadvantages of each algorithm. We then presented semi-linear resolution; an iteratively-deepening depth-first search that incorporates delayed-clause construction and includes some of the advantages of GCA. We listed the advantages of SLR over a regular IDDFS and compared the shared strategies between SLR and GCA. Finally, we listed the conditions required for SLR to be refutation complete.

CHAPTER

# 5

# Attribute Sequences

Semi-linear resolution performs a selection on the initial clauses and side premises. The selection process is based on two criteria. One relies on lookup tables and the other on clause attributes. Lookup tables are used to reduce the number of unsuccessful applications of inference rules, thereby increasing the inference rate of SLR. Clause attributes are used to reduce the explorable search space and improve the efficiency of SLR. Lookup tables are discussed in Chapter 6. In this chapter, we discuss clause attributes and we concentrate mainly on the length attribute.

Since SLR seeks a refutation, it is possible to reduce the explorable search space substantially by relying on the relationship between the attributes of input clauses and the attributes of the empty clause. For instance, the length of the empty clause is zero. Therefore, the length of the derived clauses must eventually decrease as the search depth increases, otherwise the empty clause cannot be obtained. In a linear resolution the relationship between the initial clause, the side premises and the empty clause can be represented by attribute sequences. An attribute sequence is a sequence of tuples of clause attributes where each tuple contains attributes related to the input clauses and the generated clauses.

In this, chapter we analyze the size of the search space of SLR from two perspectives, the number of generated clauses and the number of attribute sequences where the attribute is the length of a clause. In the former, we derive an upper bound on the number of generated clauses, whereas in the latter we

compute the number of attribute sequences that can be used as a guide to prune the search space without compromising completeness. To simplify the analysis of the number of attribute sequences, we proceed from the simplest case, where binary resolution is the only inference rule, and then include other inference rules, such as binary factoring, demodulation, and paramodulation. Finally, we construct the minimum subset of attribute sequences that can be used as a guide to prune the search without compromising completeness.

# 5.1 Number of generated clauses in SLR

In this section, we compute an upper bound on the number of generated clauses from the SLR algorithm presented in Chapter 4 (**Figure 4-3**). Let $n_i(X)$ be a function that returns the number of elements in a set $X$ at iteration $i$. For example, $n_1(initial\_clauses)$ is the number of initial clauses selected (see **Figure 4-3** line 7 in the SLR procedure) at iteration 1. Let $\lambda_i \subseteq applicable\_rules$ be the set of inference rules at iteration $i > 0$ that require only one premise (e.g., binary factoring, equality resolution equality factoring). Suppose that a $j^{th}$ derived clause $C_{j,i-1}$ at depth $i-1$ generates some number of clauses using the inference rules $\lambda_i$, then let $f(C_{j,i-1})$ be this set of generated clauses. $f(C_{j,i-1})$ is shown in **Figure 5-1** as a rectangle.

In **Figure 5-1**, at depth 0, the circles represent initial clauses. At depth 1, the circles represent clauses generated from the initial clauses using *applicable_rules*. At depth 2 the circles represent clauses generated from the clauses at depth 1 using the inference rules in $\lambda_1$. To reduce a crowded representation, we did not add in **Figure 5-1** the clauses generated at depth 2 by the rules from the set $applicable\_rules \setminus \lambda_1$.

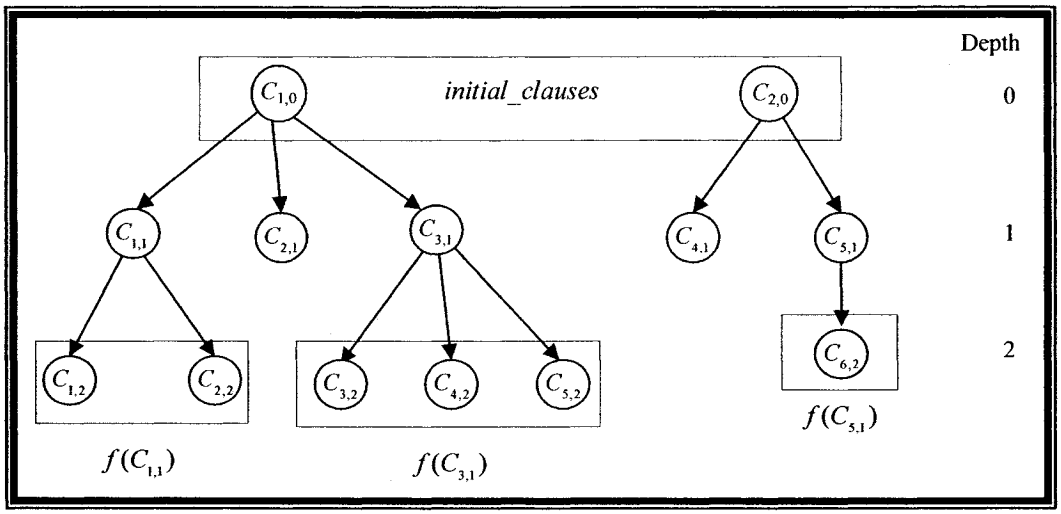



**Figure 5-1: An example of search tree showing the set $f(C_{j,i-1})$ of clauses generated using inference rules that require only one premise.**

**Table 5-1** shows the maximum number $b_{i,j}$ of clauses that can be generated at depth $i$ in iteration $j$. The first column is the iteration depth bound and the second column is the depth. The total number of generated clauses at each iteration $j$ is

$$\sum_{i=1}^{j} b_{i,j} . \qquad \textbf{(E5.1)}$$

The total number of clauses generated up to and including iteration $d_{\max}$, denoted by $ngen(d_{\max})$, is the sum of the total number of generated clause at each iteration $1 \le j \le d_{\max}$. Therefore, using **E5.1**, $ngen(d_{\max})$ is calculated by the formula

$$ngen(d_{\max}) = \sum_{j=1}^{d_{\max}} \sum_{i=1}^{j} b_{i,j} . \qquad \textbf{(E5.2)}$$

If we assume a uniform branching factor of $b$ and an initial number of clauses $m$, then **E5.1** becomes

$$\sum_{i=1}^{j} m \cdot b^{i} . \qquad \textbf{(E5.3)}$$

With a uniform branching factor $b$ and an initial number of clauses $m$, **E5.2** becomes

$$ngen(m,b,d_{\max}) = \sum_{j=1}^{d_{\max}} \sum_{i=1}^{j} m \cdot b^i \,. \qquad \textbf{(E5.4)}$$

**Table 5-1: Maximum number of generated clauses in SLR at each iteration**

| Bound | Depth | Maximum number of generated clauses |
|---|---|---|
| 1 | 1 | $b_{1,1} = n_1(initial\_clauses) \times n_1(side\_premises) \times n_1(applicable\_rules)$ |
| 2 | 1 | $b_{1,2} = n_2(initial\_clauses) \times n_2(side\_premises) \times n_2(applicable\_rules)$ |
| | 2 | $b_{2,2} = b_{1,2} \times (n_2(side\_premises) - 1) \times n_1(applicable\_rules \setminus \lambda_2) + \sum_{j=1}^{b_{1,2}} n_2(f(C_{j,1}))$ |
| 3 | 1 | $b_{1,3} = n_3(initial\_clauses) \times n_3(side\_premises) \times n_3(applicable\_rules)$ |
| | 2 | $b_{2,3} = b_{1,3} \times (n_3(side\_premises) - 1) \times n_3(applicable\_rules \setminus \lambda_3) + \sum_{j=1}^{b_{1,3}} n_3(f(C_{j,1}))$ |
| | 3 | $b_{3,3} = b_{2,3} \times (n_3(side\_premises) - 2) \times n_3(applicable\_rules \setminus \lambda_3) + \sum_{j=1}^{b_{2,3}} n_3(f(C_{j,2}))$ |
| ... | ... | ... |
| $d_{\max}$ | 1 | $b_{1,d_{\max}} = n_{d_{\max}}(initial\_clauses) \times n_{d_{\max}}(side\_premises) \times n_{d_{\max}}(applicable\_rules)$ |
| | 2 | $b_{2,d_{\max}} = b_{1,d_{\max}} \times (n_{d_{\max}}(side\_premises) - 1) \times n_{d_{\max}}(applicable\_rules \setminus \lambda_{d_{\max}}) + \sum_{j=1}^{b_{1,d_{\max}}} n_{d_{\max}}(f(C_{j,1}))$ |
| | ... | |
| | $d_{\max}$ | $b_{d_{\max},d_{\max}} = b_{d_{\max}-1,d_{\max}} \times (n_{d_{\max}}(side\_premises) - d_{\max} + 1) \times n_{d_{\max}}(applicable\_rules \setminus \lambda_{d_{\max}}) + \sum_{j=1}^{b_{d_{\max}-1,d_{\max}}} n_{d_{\max}}(f(C_{j,d_{\max}-1}))$ |

**E5.3** is a geometric series and can be written as $m \cdot (b^{j+1} - b)/(b-1)$. If we substitute this expression in **E5.4**, then expand and simplify the resulting expression, we get

$$ngen(m,b,d_{\max}) = m \cdot b \cdot \frac{b^{d_{\max}+1} - b \cdot d_{\max} - b + d_{\max}}{(b-1)^2}. \qquad \textbf{(E5.5)}$$

In practice, it is usually sufficient to set $d_{\max} \leq 30$ when SLR is used because when the depth reaches $d_{\max} = 30$, even for values as small as $b = 3$ and $m = 2$, SLR would have to generate (using **E5.5**) approximately $10^{15}$ clauses (see **Figure E-2** in Appendix E for values of $m$ and $b$ in practice).

## 5.2 Search Paths

If $\Delta = \langle \mathcal{I}_1, \ldots, \mathcal{I}_n \rangle$ is a linear derivation of length $n > 0$, then $SP(\Delta)$ is called a **search path** and is defined as the sequence

$$SP(\Delta) = \langle (C_{init}, \mathcal{D}(\mathcal{I}_1)), \ldots, (C(\mathcal{I}_{n-1}), \mathcal{D}(\mathcal{I}_n)) \rangle,$$

where

$C_{init}$ is the initial clause of the derivation $\Delta$,

for all $1 \leq i \leq n-1$, $C(\mathcal{I}_i)$ is an intermediate conclusion,

for all $1 \leq i \leq n$, $\mathcal{D}(\mathcal{I}_i)$ are the side premises.

A **refutation search path** (*RSP*) is search path that leads to the empty clause.

In the case where for all $1 \leq i \leq n$, $1 \leq |Prem(\mathcal{I}_i)| \leq 2$, we write $SP(\Delta)$ as

$$SP(\Delta) = \langle (R_0, D_0), \ldots, (R_{n-1}, D_{n-1}) \rangle,$$

where

$R_0 = C_{init}$,

for all $0 \leq i \leq n-1$, $D_{i-1} \in \mathcal{D}(\mathcal{I}_i)$ or $D_i = R_i$,

for all $1 \le i \le n-1$, $R_i = C(I_i)$.

If $D_i = R_i$ then the inference rule requires only one premise, such as binary factoring.

**Example 5.1:**

$S = \{B_1, B_2, B_3, B_4\}$ is set of constructed clauses.

$B_1 = \neg P \vee \neg Q \vee D$, $B_2 = \neg D \vee \neg Q$, $B_3 = P$, $B_4 = Q$.

$B_1 : \neg P \vee \neg Q \vee D$

$B_2 : \neg D \vee \neg Q$

$R_1 : \neg P \vee \neg Q \vee \neg Q$ (resolvent)

$R_2 : \neg P \vee \neg Q$ (factor/merge clause)

$B_3 : P$

$R_3 : \neg Q$ (resolvent)

$B_4 : Q$

$R_4 : \phi$ (resolvent)

The derivation $\Delta$ in Example 5.1 is a linear derivation where the number of premises in each inference is either 1 or 2. Therefore, the search path $SP(\Delta)$ is expressed as $\langle (B_1, B_2), (R_1, R_1), (R_2, B_3), (R_3, B_4) \rangle$. The length of this sequence is 4 and it contains one application of binary factoring and three applications of binary resolutions. The binary factoring is indicated by the pair $(R_1, R_1)$.

SLR explores many search paths where some of them lead to an empty clause while others don't. Also, the search paths that lead to the empty clause differ in length. It is not possible to determine ahead of time the search paths that lead to

the empty clause; otherwise proving unsatisfiability would be relatively easy. Instead, we have to explore all possible paths or at least the potential ones that may lead to the empty clause. The potential paths are the paths that remain after eliminating the ones that would certainly not lead to the empty clause. By relying on the relationship between the clause attributes, the inference rules involved, and the bound set for every iteration, we can eliminate many of the unnecessary paths. Using such relationship, we study sequences of clause attributes and in particular, the lengths of clauses, at each level in the derivation. By relying on those sequences, we can reduce the search space explored by SLR independent of the semantics of the clauses and without compromising completeness.

## 5.3 Attribute Sequences

An **attribute sequence** is a sequence of tuples of numbers where each number within the tuple represents one attribute of one premise of the applied inference rule in a linear derivation. Let $f(C) \in \mathcal{A}_{\mathbb{R}}(C)$ denote a function that returns some real valued attribute of a clause $C$, and if $S = \{D_1,\ldots,D_n\}$ is a multiset of clauses, then $f(S) = (f(D_1),\ldots,f(D_n))$, where $f(D_i) \in \mathcal{A}_{\mathbb{R}}(D_i)$ for $1 \le i \le n$. If $\Delta = \langle \mathcal{I}_1,\ldots,\mathcal{I}_n \rangle$ is a linear derivation of length $n > 0$, then

$$SP(\Delta) = \langle (C_{init}, \mathcal{D}(\mathcal{I}_1)),\ldots,(C(\mathcal{I}_{n-1}), \mathcal{D}(\mathcal{I}_n)) \rangle$$

is the search path corresponding to $\Delta$ and

$$ATS(f,\Delta) = \langle (f(C_{init}), f(\mathcal{D}(\mathcal{I}_1))),\ldots,(f(C(\mathcal{I}_{n-1})), f(\mathcal{D}(\mathcal{I}_n))) \rangle,$$

is the attribute sequence corresponding to $SP(\Delta)$ with respect to the attribute $f$. For instance, an attribute sequence, where the attribute is the length of the clause, of the refutation search path $\langle (B_1,B_2),(R_1,R_1),(R_2,B_3),(R_3,B_4) \rangle$ (from Example 5.1) is

$$\langle (Len(B_1), Len(B_2)),(Len(R_1), Len(R_1)),(Len(R_2), Len(B_3)),(Len(R_3), Len(B_4)) \rangle.$$

The set of attribute sequences, *ATS*, forms the co-domain of a mapping, $\mathcal{M}$, between search paths, *SP* (domain), and *ATS*. The mapping $\mathcal{M}$ is expressed as

$$\mathcal{M}: SP \rightarrow ATS .$$

Let $\Delta(S)$ be set of all derivations that can be formed from a set of clauses *S*, then the set of all derivations of length less than $k > 0$ that can be formed from a set of clauses *S* is denoted by

$$\Delta(S,1..k) = \{\Delta : \Delta \in \Delta(S) \text{ and } 1 \leq |\Delta| \leq k\} .$$

The set of all search paths that can be formed from $\Delta(S,1..k)$ is denoted by $SP(\Delta(S,1..k))$. The corresponding set of attribute sequences with respect to an attribute $f$ is denoted by $ATS(f, \Delta(S,1..k))$. The number of attribute sequences of lengths between 1 and $k > 0$ with respect to an attribute $f$ that can be formed from a given set of clauses *S* is

$$|ATS(f, \Delta(S,1..k))| . \qquad \textbf{(E5.6)}$$

In the following sections, we present an algorithm that generates all attribute sequences, where the attribute is the length of a clause, which correspond to refutation search paths for a given set of clauses. We derive a formula for computing the expression in **E5.6** when $f$ is the length of a clause and the inference rule used is binary resolution. We then determine the minimum set of attribute sequences that can be used as a guide in the selection of clauses in SLR without compromising completeness and we provide an algorithm that generates the minimized set of attribute sequences.

We begin our presentation by assuming that binary resolution is the only inference rule applied in a derivation (other rules are discussed afterwards). We demonstrate how attribute sequences that correspond to refutation search paths

can be used as a guide to reduce the number of search paths. In other words, since we don't know which search paths are refutation search paths, we rely on attribute sequences to find out which ones may lead to a refutation. This is illustrated in **Figure 5-2.** The set *SP* is the set of search paths that can be formed from a given set of clauses. *RSP* is the subset of *SP* where each element of *RSP* leads to a refutation. *ATS* is the set of attribute sequences corresponding to *SP*. *W* is the set of attribute sequences corresponding to *RSP*. When a search is conducted by an ATP, the set *RSP* is not known. By using the set *W* as a guide, an ATP can reduce the explorable search space from *SP* to *RSP*. This is what SLR does when it selects initial clauses and side premises. It relies on attribute sequences to reduce the size of the explorable search space.

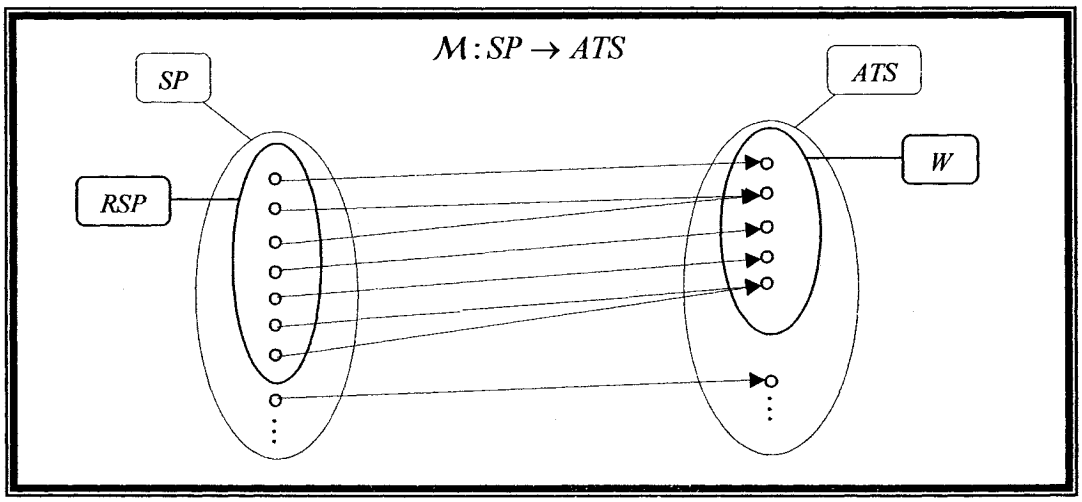



**Figure 5-2: An example of a relationship between search paths and attribute sequences.**

### 5.3.1 Restricting the number of search paths under binary resolution

When searching for a refutation and the bound is set to 1, we are actually seeking a derivation of the empty clause in one step. It is readily noticeable that we do not

need to resolve clauses whose lengths are greater than one, because the length of a resolvent $R$ resulting from the binary resolution of the two clauses, $C_1$ and $C_2$, is

$$Len(R) = Len(C_1) + Len(C_2) - 2$$

and if $Len(C_1) + Len(C_2) > 2$, then $Len(R) > 0$ which is greater than the length of the empty clause; $Len(\phi) = 0$. Therefore, when seeking a derivation of length 1 of the empty clause, the sum of the lengths of the clauses forming the premises of binary resolution must be less or equal to the length of the resolvent (the empty clause) which is zero, plus two;

$$Len(C_1) + Len(C_2) \leq Len(R) + 2 = Len(\phi) + 2 = 0 + 2 = 2.$$

In Example 5.1, there is no need to attempt a resolution between $B_1$ and $B_2$ when looking for a search path of length 1 because $Len(B_1) + Len(B_2) = 5$ and it is greater than 2. Only clauses $B_3$ and $B_4$ are potential candidates at this point because the sum of their lengths is 2. However, since they do not resolve together, there is no derivation of length 1 that results in the empty clause and the search for a proof within iteration 1 is over. SLR proceeds with an attempt to find a derivation of the empty clause in two steps, so the bound is incremented to 2.

Within iteration 2, the sum of the lengths of the premises at depth 0, i.e., the sum of the lengths of the initial input clauses, must be less or equal to 4. In Example 5.1 the choice for the initial clauses, based solely on their lengths, can be any of the pairs from the set

$$\{(B_1, B_3), (B_1, B_4), (B_2, B_3), (B_2, B_4), (B_3, B_4),$$
$$(B_3, B_1), (B_4, B_1), (B_3, B_2), (B_4, B_2), (B_4, B_3)\}.$$

Some of these pairs of clauses resolve while others don't. By eliminating all the pairs of clauses from the above set that don't resolve, we are left with the set

$$\{(B_1, B_3), (B_1, B_4), (B_2, B_4), (B_3, B_1), (B_4, B_1), (B_4, B_2)\}.$$

Since the bound is set to 2, then at depth 1 there is only one step left and hence, the sum of the length of the resolvent $R_1$ and the length of a clause $B_i$ for some $1 \le i \le 4$, must be less or equal to 2 , i.e., $Len(R_1) + Len(B_i) \le 2$, if the empty clause is to be produced within a total of 2 steps. The minimum length for any of the $B_i$'s is 1, so the maximum length for $R_1$ must be $Len(R_1) = 2 - Len(B_i) = 2 - 1 = 1$. Any of the pairs $(B_1, B_3)$, $(B_1, B_4)$, $(B_3, B_1)$ or $(B_4, B_1)$ produces a resolvent of length 2 which is greater than 1, the maximum length allowed for $R_1$. In this case, the resolvent is said to be **oversized for the current depth**. In general, when a refutation search path of length *k* is sought, a resolvent $R_d$ at depth $d > 0$ within iteration *k* is said to be **oversized for depth** $d$ if $Len(R_d) > k - d$.

In SLR, proceeding with the search from an oversized resolvent is a waste of time, as shown above. Therefore, all paths leading to an oversized resolvent must be avoided. This implies that at depth $d > 0$ within iteration *k*, the length of the resolvent $R_d$ must be restricted by the expression

$$Len(R_d) \le k - d . \qquad \textbf{(E5.7)}$$

We can compute the length of any resolvent $R_d$, for $d > 0$, in a refutation search path $\langle (R_0, D_0), \ldots, (R_{k-1}, D_{k-1}) \rangle$ recursively using the formula

$$Len(R_d) = Len(R_{d-1}) + Len(D_{d-1}) - 2 . \qquad \textbf{(E5.8)}$$

From **E5.7** and **E5.8** we conclude that

$$Len(R_{d-1}) + Len(D_{d-1}) \le k - d + 2 . \qquad \textbf{(E5.9)}$$

In Example 5.1 the pairs $(B_1, B_3)$, $(B_1, B_4)$, $(B_3, B_1)$ and $(B_4, B_1)$ produce oversized resolvents for depth 1, and so they are eliminated from the set $\{(B_1, B_3), (B_1, B_4), (B_2, B_4), (B_3, B_1), (B_4, B_1), (B_4, B_2)\}$. This set is thus reduced to

the set $\{(B_2, B_4), (B_4, B_2)\}$. $B_2$ and $B_4$ produce the resolvent $R_1 = \neg D$. At iteration 2 and depth 2, $k = 2$ and $d = 2$. By using **E5.9**, and knowing that $Len(R_1) = 1$, we look for a clause in $S$ that resolves with $R_1$, such that $1 + Len(B_i) \leq 2 - 2 + 2 = 2$, for some $1 \leq i \leq 4$.In other words, we look for a unit clause that resolves with $R_1$. Since there aren't any, then iteration 2 is over without resulting in a proof of unsatisfiability. Adding the restriction of **E5.9** to SLR, reduces the number of search paths explored by SLR from six to three as shown in **Figure 5-3**. The search paths and attribute sequences are indicated below the leaves of the tree.

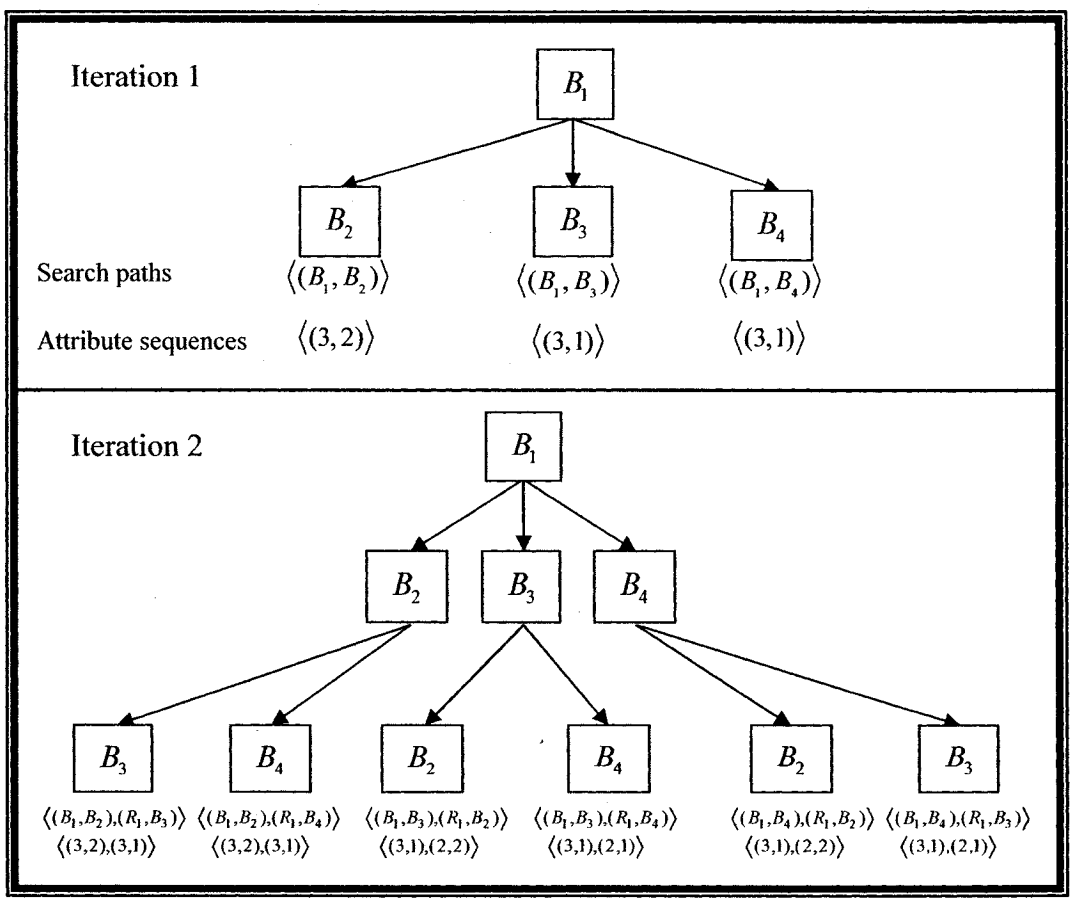



**Figure 5-3: Search paths and attribute sequences for iteration 1 and iteration 2 of Example 5.1.**

It is clear from **Figure 5-3** that by applying the restriction **E5.9** on the lengths of the clauses many search paths can be avoided. By using **E5.9** to reduce the number of attribute sequences explored, we actually reduce the number of search paths explored.

### 5.3.2 Constructing attribute sequences

From a set of given clauses, we can determine all the attribute sequences corresponding to all the refutation search paths of the form $\langle (R_0, D_0), \ldots, (R_{k-1}, D_{k-1}) \rangle$. We begin by selecting a length for an initial clause $R_0$ such that $Len(R_0) \leq k$ (from **E5.7**). Then, iteratively, for every $0 \leq i \leq k-1$, we compute first the possible range of lengths for $D_i$ by the formula

$$Len(D_i) \leq k - i + 1 - Len(R_i) \qquad \textbf{(E5.10)}$$

and then for each of the possible values for $Len(D_i)$, we calculate $Len(R_{i+1})$ using the formula

$$Len(R_{i+1}) = Len(R_i) + Len(D_i) - 2. \qquad \textbf{(E5.11)}$$

The following function, CONSTRUCTATSBR, computes the attribute sequences for iteration $k$ using **E5.10** and **E5.11** and assuming that only binary resolution is used. It then returns the results in the array *atsequences*.

```
CONSTRUCTATSBR(k:integer):array

1.  atsequences:array
2.  current_ats:array
3.  n_seq:array

4.  for initial_length := 1 to k
5.      current_ats[1] := initial_length
6.      CONSTRUCTRESTOFATSBR(current_ats, 1, 1, k, initial_length,
                                    atsequences, n_seq)
7.  end for
8.  return (atsequences)
```

Line 4 of CONSTRUCTATSBR loops over the range of possible values for the length of the initial clause $R_0$. Since there is no need to store $Len(R_i)$ because it is implied, the three dimensional array *atsequences* stores all the attribute sequences in a compact format as $\langle Len(R_0), Len(D_0), Len(D_1), \ldots, Len(D_{k-1}) \rangle$. For example, for $k = 4$, one of the attribute sequences is $\langle (1,4),(3,1),(2,1),(1,1) \rangle$ in full format. It is stored in the compact format in *atsequences*[4][5] as $\langle 1,4,1,1,1 \rangle$ such that *atsequences*[4][5][1]=1, *atsequences*[4][5][2]=4, and so on. The array *n_seq* is a counter of the number of attribute sequences for each length for iteration *k*. For example, for $k = 4$, *n_seq*[5]=14 and that is there are 14 attribute sequences of length 5 (the length is measured over the compact format).

```
CONSTRUCTRESTOFATSBR (current_ats:array, ats_length:integer,
                      depth:integer, k:integer,
                      resolvent_length:integer, atsequences:array,
                      n_seq:array):integer

1.  if resolvent_length = 0 then
2.      INCREMENT(n_seq[ats_length])
3.      ats_number := n_seq[ats_length]
4.      for ats_entry := 1 to ats_length
5.          atsequences[ats_length][ats_number][ats_entry] :=
            current_ats[ats_entry]
6.      end for
7.  end if
8.  return
9.  for input_length := 1 to k – depth + 2 – resolvent_length
10.     current_ats[depth + 1] := input_length
11.     CONSTRUCTRESTOFATSBR(current_ats, ats_length + 1, depth + 1, k,
                             resolvent_length + input_length – 2,
                             atsequences, n_seq)
12. end for
```

Line 5 of CONSTRUCTATSBR sets the first element of the sequence to the length of $R_0$ and the rest of the sequence is determined by the function

CONSTRUCTRESTOFATSBR. CONSTRUCTRESTOFATSBR is called recursively and the recursive termination condition at line 1 checks if the resolvent's length is zero to ensure that no sequences longer than necessary are generated. When the resolvent's length is zero, the number of attribute sequences of the current length is incremented and the current sequence is copied into the array *atsequences*. In other words, CONSTRUCTRESTOFATSBR adds the current sequence to the set of sequences of length, *ats_length*. Line 9 of CONSTRUCTRESTOFATSBR loops over the entire range of lengths that the clauses from the input set of clauses can have based on the restriction of **E5.10**[1].

**Table 5-2** lists all the 22 attribute sequences in compact notation for $k = 4$ in the order they are generated by CONSTRUCTATSBR and not by their lengths.

**Table 5-2: Attribute sequences for *k*=4**

| | | | |
|---|---|---|---|
| ⟨1,1⟩ 1 | ⟨1,2,1⟩ 2 | ⟨1,2,2,1⟩ 3 | ⟨1,2,2,2,1⟩ 4 |
| ⟨1,2,3,1,1⟩ 5 | ⟨1,3,1,1⟩ 6 | ⟨1,3,1,2,1⟩ 7 | ⟨1,3,2,1,1⟩ 8 |
| ⟨1,4,1,1,1⟩ 9 | ⟨2,1,1⟩ 10 | ⟨2,1,2,1⟩ 11 | ⟨2,1,2,2,1⟩ 12 |
| ⟨2,1,3,1,1⟩ 13 | ⟨2,2,1,1⟩ 14 | ⟨2,2,1,2,1⟩ 15 | ⟨2,2,2,1,1⟩ 16 |
| ⟨2,3,1,1,1⟩ 17 | ⟨3,1,1,1⟩ 18 | ⟨3,1,1,2,1⟩ 19 | ⟨3,1,2,1,1⟩ 20 |
| ⟨3,2,1,1,1⟩ 21 | ⟨4,1,1,1,1⟩ 22 | | |

**Figure 5-4** shows a graph of the 22 attribute sequences for $k = 4$. Every vertex in the graph contains a pair $(Len(R_i), Len(D_i))$ for $0 \le i \le 4$. At depth 0, the vertices represent the possible lengths for the initial pair of clauses $(R_0, D_0)$. The subsequent depths indicate the possible lengths for the pairs $(R_i, D_i)$ for $1 \le i \le 4$.

[1] Notice that Line 9 the range goes up to $k$ – *depth* + 2 – *resolvent_length* instead of $k$ – *depth* + 1 – *resolvent_length*. This is because we start at depth 1 instead of 0. The depth is used as an index counter for the array. We prefer not to use the zero index in an array for reasons that are not related to this thesis.

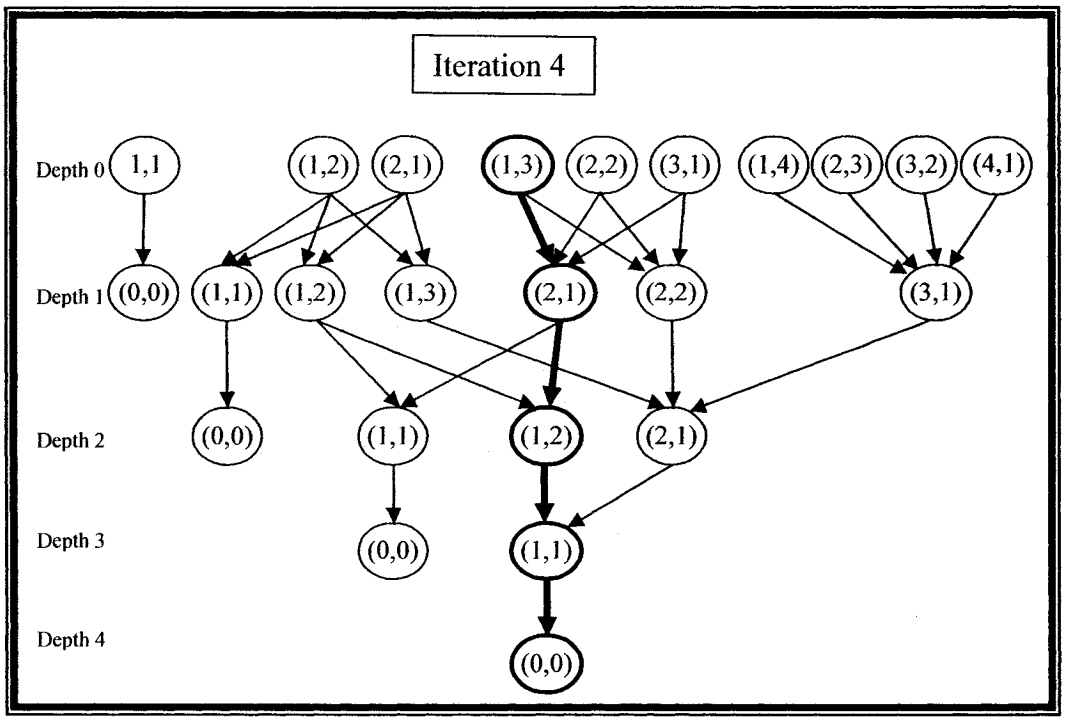



**Figure 5-4: A graph of the attribute sequences for iteration 4.**

Any path that starts with a vertex from depth 0 and ends with a vertex containing (0,0) is an attribute sequence. For example, the highlighted path in the figure is the sequence $\langle(1,3),(2,1),(1,2),(1,1)\rangle$ or simply $\langle 1,3,1,2,1\rangle$ which is sequence number 7 in **Table 5-2**. The vertices containing (0,0) indicate that the resolvent is the empty clause.

Since the number of attribute sequences can be reduced by imposing the restriction of **E5.9**, it is important to determine the ratio between the size of the set of the attribute sequences without the restriction of **E5.9** and with the restriction of **E5.9**. This ratio reveals the gain in efficiency obtained by imposing the restriction of **E5.9**.

### 5.3.3 Calculating the number of attribute sequences

Determining the number of attribute sequences with the restriction imposed by **E5.9** requires an observation of the pattern existing within the generated sequences. For instance, every iteration contains all the attribute sequences of the previous iterations as well as additional paths, as shown in **Figure 5-5**. At iteration $k$, let the number of paths to a node (0,0) from the vertices (1,$x$) (i.e., whose first entry is 1) at depth $d$ be $p_1$. Let the number of paths to a node (0,0) from all the vertices ($y$,$z$) at $d+1$ where $0 \le y \le k-(d+1)$ be $p_2$. Then $p_1 = p_2$. For example, the number of paths starting from the vertices (1,1), (1,2), (1,3) and (1,4) at depth 0 is 9, which is exactly the number of paths starting from the vertices (0,0), (1,1), (1,2), (2,1), (1,3), (2,2), (3,1) at depth 1 and ending at the node (0,0) as displayed in **Figure 5-5** (we count the empty path from (0,0) to (0,0) as one path).

In general, for any $1 \le r \le k$ where $k > 0$ is the iteration number, the number of paths from the vertices ($r$,$x$) for all $1 \le x \le k-r+1$ at depth $d$ is equal to the total number of paths from the vertices ($y$,$z$) at depth $d+1$, where $r-1 \le y \le k-(d+1)$ for all $0 \le z \le k$. We can now write a formula for calculating the total number of attribute sequences within iteration $k$.

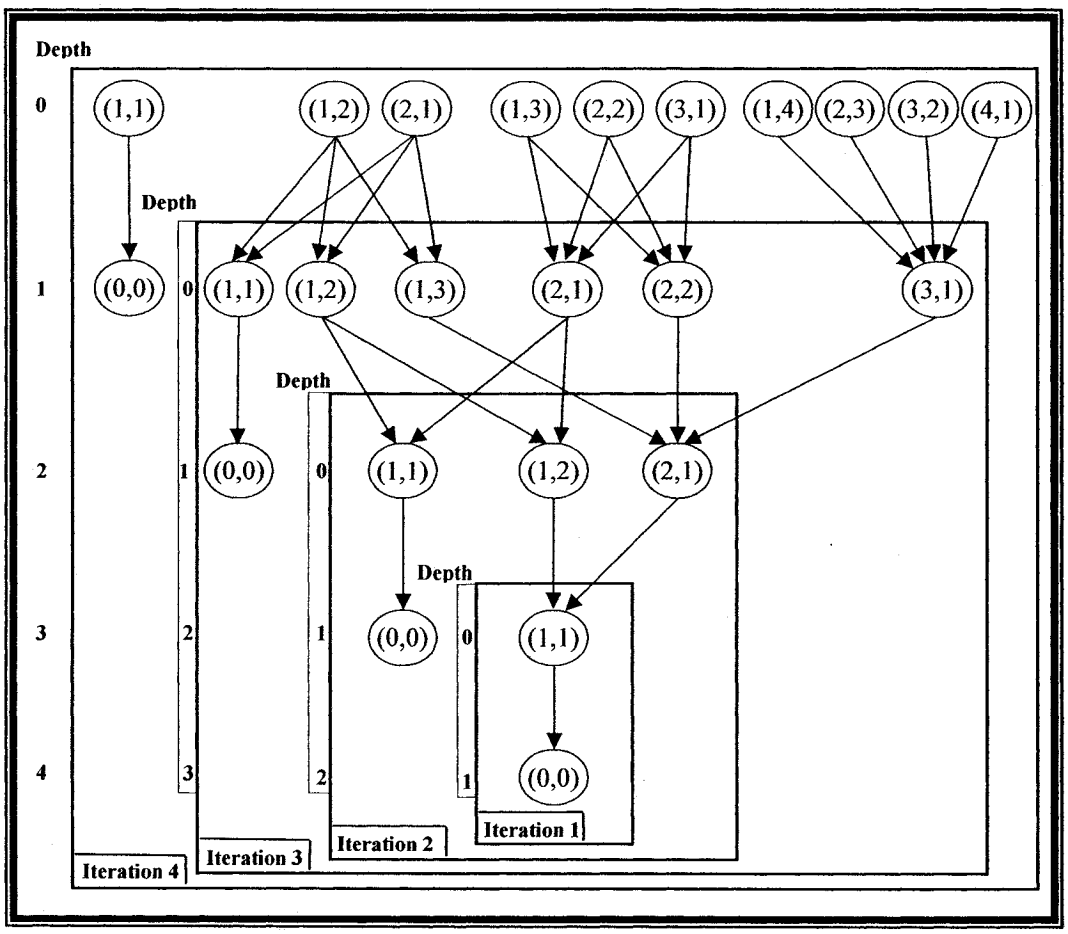



**Figure 5-5: Attribute sequences for iterations 1 to 4.**

Let the total number of attribute sequences with restriction **E5.10** on the lengths of the input clauses from the vertices at depth $d$ within iteration $k$ whose resolvent's length is $r$ be denoted by $nrp(k,d,r)$, then the total number of attribute sequences with restriction **E5.10** on the length of the input clauses within iteration $k$ is

$$tnrp(k) = \sum_{r=1}^{k} nrp(k,0,r)\,. \qquad \textbf{(E5.12)}$$

The value for $nrp(k,d,r)$ can be computed recursively by

$$nrp(k,d,r) = \begin{cases} 1 & \text{if } r = 0, \\ \sum_{i=r-1}^{k-(d+1)} nrp(k,d+1,i) & \text{if } r > 0. \end{cases} \quad \textbf{(E5.13)}$$

where $r$ is the length of a resolvent and $0 \le d \le k-1$ is the depth.

The value produced by evaluating $tnrp(k)$ is the number of attribute sequences at iteration $k$. The total number of attribute sequences from 1 to $k$ is sum $tnrp(1)+\cdots+tnrp(k)$. Therefore, the value for $|ATS(f,\Delta(S,1..k))|$ (from **E5.6**) when $f$ is the length of a clause, $S$ is a set of input clauses, $1..k$ is the range of iterations, is computed as

$$|ATS(Len,\Delta(S,1..k))| = \sum_{i=1}^{k} tnrp(i). \quad \textbf{(E5.14)}$$

To compare **E5.12** to the number of attribute sequences in each iteration without imposing the restriction of **E5.10**, we first determine the number of attribute sequences without in each iteration without imposing the restriction of **E5.10**. Let the length of the longest input clause within a given set of clause be $s_{\max}$. We assume that there are clauses in the input set of all lengths between 1 and $s_{\max}$. Since we assume that the restriction of **E5.10** are is not imposed, then any combination of lengths is possible as shown in **Figure 5-6**.

We notice from **Figure 5-6** that for $k > 1$ not all paths starting from a vertex at depth 0 continue till depth $k$. For example, when $k = 2$, the path starting with the vertex containing the pair (1,1) ends with the vertex containing the pair (0,0) at depth 1 and not at the iteration depth $k$. When $k = 3$, certain paths starting with the vertices containing either (1,2) or (2,1) end at depth 2 instead of depth 3,

which is the iteration depth. The reason is because those paths arrive at vertices containing (0,0) that marks the end of an attribute sequence.

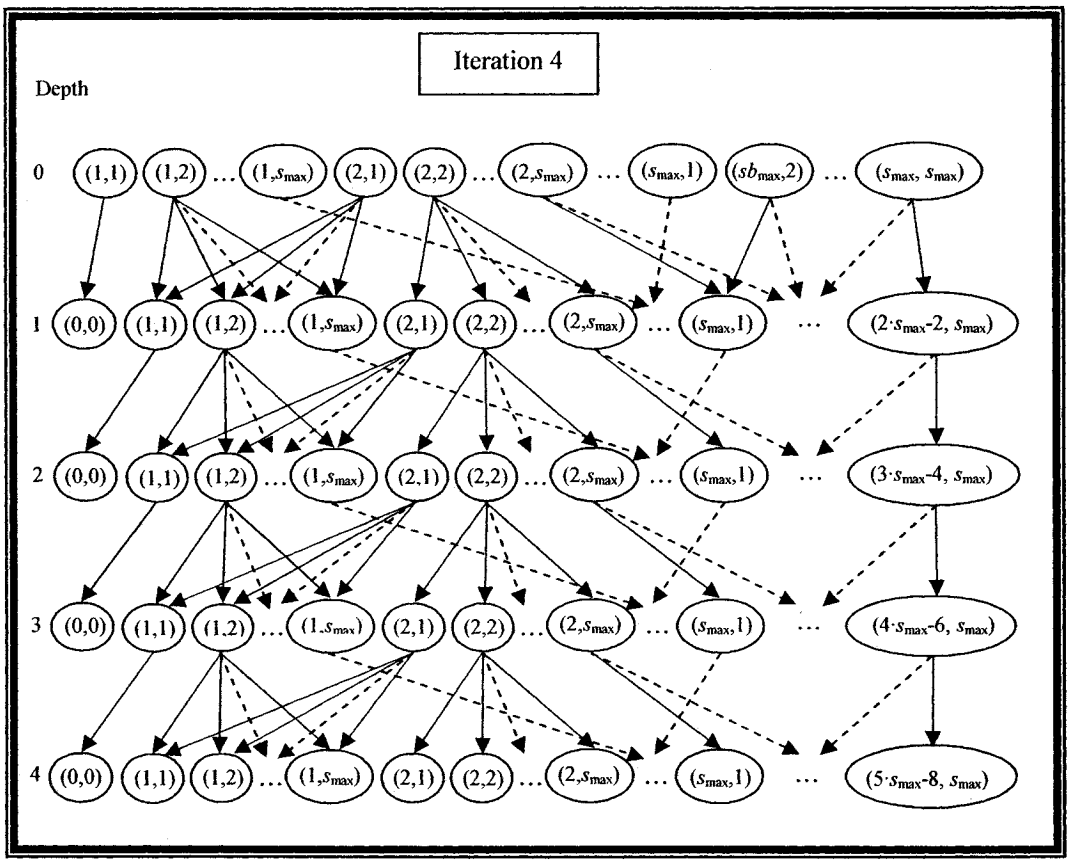



**Figure 5-6: A graph of attribute sequences of iteration 4 without restrictions on the lengths of the resolvents.**

In order to calculate the exact total number of attribute sequences that may be explored by SLR without any restriction on the lengths of the clauses, we have to take into consideration the fact that certain paths are not of length $k$. A simple observation of the graph in **Figure 5-6** reveals that the number of vertices at depth $d = 0$ is ${s_{\max}}^2$. The number of attribute sequences when $k = 1$ is $({s_{\max}}^2 - 1) \cdot s_{\max} + 1$ because every vertex at depth 0 has an out-degree of $s_{\max}$ except the vertex containing the pair (1,1) which has an out-degree of one. Let

$np(0) = {s_{\max}}^2$. Let $np(1) = ({s_{\max}}^2 - 1) \cdot s_{\max} + 1$. When $k = 2$, the number of paths, $np(2)$, from the vertices at depth 0 to the vertices at depth 2, including the paths of the vertices that end at (0,0), is $np(2) = (np(1) - 2) \cdot s_{\max} + 2$. When $k = 3$, the number of paths, $np(3)$, from the vertices at depth 0 to the vertices at depth 3 is $np(3) = (np(2) - 4) \cdot s_{\max} + 4$. In general, for $k > 0$, the number of paths, $np(k)$, in the graph of **Figure 5-6** starting from the vertices at depth $d = 0$ and ending either with a vertex containing the pair (0,0), or at depth $d = k$ is $(np(k-1) - 2^{k-1}) \cdot s_{\max} + 2^{k-1}$.. The total number of attribute sequences within iteration $k$ without restriction on the lengths of the clauses is $tnp(k) = \sum_{i=1}^{k} np(i)$.

By expanding the expressions of $np(i)$, we get

$$tnp(k) = \sum_{d=1}^{k} \left( {s_{\max}}^{d+2} - {s_{\max}}^{d} - \sum_{i=1}^{d-1} \left( 2^{i-1} \cdot {s_{\max}}^{d-i} \right) + 2^{d-1} \right). \qquad \textbf{(E5.15)}$$

The summation in **E5.15** can be expanded, factored and simplified through algebraic manipulations to reach the formula

$$tnp(k) = \frac{{s_{\max}}^{k+4} - 2 \cdot {s_{\max}}^{k+3} - {s_{\max}}^{k+2} + {s_{\max}}^{k+1}}{(s_{\max} - 2)(s_{\max} - 1)} - \frac{{s_{\max}}^{4} + 2 \cdot {s_{\max}}^{3} + (2^{k+1} - 1) \cdot {s_{\max}}^{2} - (2^{k+2} - 3) \cdot s_{\max} + 2^{k+1} - 2}{(s_{\max} - 2)(s_{\max} - 1)}. \qquad \textbf{(E5.16)}$$

**Table 5-3** shows the values for the total number of attribute sequences for the first fifteen iterations with and without the restriction on the lengths of the input clauses and the ratio, $q_{np}$, of the two totals. The largest input clause is assumed to

be 7, i.e., $s_{\max} = 7$, which is equal to the rounded average of the longest clauses in all theorems within the TPTP version 2.6.0 library.

**Table 5-3: Number of attribute sequences for the first 15 iterations**

| Iteration | $tnrp(k)$ | $tnp(k)$ | $q_{np} = tnrp(k)/tnp(k)$ |
|---|---|---|---|
| 1 | 1 | 337 | 0.002967359 |
| 2 | 3 | 2,684 | 0.001117735 |
| 3 | 8 | 19,089 | 0.000419090 |
| 4 | 22 | 133,876 | 0.000164331 |
| 5 | 64 | 937,289 | 0.000068282 |
| 6 | 196 | 6,560,988 | 0.000029874 |
| 7 | 625 | 45,926,497 | 0.000013609 |
| 8 | 2055 | 321,484,292 | 0.000006392 |
| 9 | 6917 | 2,250,387,321 | 0.000003074 |
| 10 | 23,713 | 15,752,705,452 | 0.000001505 |
| 11 | 82,499 | 110,268,926,225 | 0.000000748 |
| 12 | 290,511 | 771,882,459,348 | 0.000000376 |
| 13 | 1,033,411 | 5,403,177,166,633 | 0.000000191 |
| 14 | 3,707,851 | 37,822,240,068,478 | 0.000000098 |
| 15 | 13,402,696 | 264,755,680,283,073 | 0.000000051 |

It is obvious from **Table 5-3** that without the restriction on the lengths of the input clauses, the total number of attribute sequences is much larger than with the restriction of **E5.10**. Notice that the ratio $q_{np}$ decreases by almost a half with every iteration. Consequently, the higher the iteration, the more apparent is the advantage of restricting the length of the initial clause and side premises when conducting a search in SLR. However, even with the restriction imposed by **E5.10**, the total number of attribute sequences is still large for iteration 13 and above. Recall that those are the number of attribute sequences and not number of search paths. Every attribute sequences corresponds to one or more search paths. For example, the attribute sequence $\langle(1,3),(2,1),(1,1)\rangle$ corresponds to all search paths that start with two clauses, where the first has a length of 1 and the second has a length 3. The pair (2,1) implies that binary resolution is attempted between all unit clauses with the resolvent of length 2. So the if the number attribute

sequences for iteration 13 is 1,033,411, then it does not mean that 1,033,411 search paths are explored. The number of search paths can be more than that. Fortunately, we can reduce the number of attribute sequences further which implies that we can reduce the number of refutation search paths.

### 5.3.4 Minimizing the number of attribute sequences

By applying the restriction on the lengths of the input clauses, we have reduced the total number of attribute sequences substantially. Consequently, we have reduced the number of explorable search paths. However, it is possible to optimize this number further due to the fact that the order of the initial clauses does not affect the search path. Therefore, a derivation represented by the sequence $\langle (R_0, D_0), \ldots, (R_{k-1}, D_{k-1}) \rangle$ is same as the derivation represented by the sequence $\langle (D_0, R_0), \ldots, (R_{k-1}, D_{k-1}) \rangle$ and the two sequences are equivalent. This would reduce the total number of attribute sequences by almost a half.

If we view the initial pair of an attribute sequences as shown in **Figure 5-7**, then each element $n_{ij}$ within the table represents the number of attribute sequences that start with the pair $(Len(R_0), Len(D_0)) = (i, j)$. For example, $n_{11}$ represents the number of attribute sequences that start with (1,1). Since there is only one attribute sequence that starts with (1,1), then $n_{11} = 1$.

| $Len(R_0)$ \ $Len(D_0)$ | 1 | 2 | ⋯ | $k$ |
|---|---|---|---|---|
| 1 | $n_{11}$ | $n_{12}$ | ⋯ | $n_{1k}$ |
| 2 | $n_{21}$ | $n_{22}$ | ⋯ | $n_{2k}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $k$ | $n_{k1}$ | $n_{k2}$ | ⋯ | $n_{kk}$ |

**Figure 5-7: The number of attribute sequences viewed in table form.**

Within iteration $k$, $n_{ij} = nrp(k,1,i+j-2)$ and the sum of the values in a row $i$ is equal to the number of attribute sequences that start with $Len(R_0) = i$. Therefore, $\sum_{j=1}^{k} n_{ij} = nrp(k,0,i)$ and $tnrp(k) = \sum_{i=1}^{k} n_{ii} + 2 \cdot \sum_{j=1}^{i-1} n_{ij}$. Since reversing the order of the initial clauses produces equivalent derivations and consequently, equivalent attribute sequences, then $n_{ij} = n_{ji}$. Therefore, all attribute sequences that start with $(j,i)$ can be removed from the restricted attribute sequences search space leading to a total number of attribute sequences (the highlighted section in **Figure 5-7**)

$$tnrp'(k) = \sum_{i=1}^{k} n_{ii} + \sum_{j=1}^{i-1} n_{ij} = \sum_{i=1}^{k} nrp(k,1,2 \cdot i-2) + \sum_{j=1}^{i-1} nrp(k,1,i+j-2) \qquad \textbf{(E5.17)}$$

Furthermore, the attribute sequences of length less than $k$ need not be explored at iteration $k$ because they have been explored within previous iterations. For example, the attribute sequence $\langle (1,2),(1,1) \rangle$ of length 2 need not be explored within iteration 3 and above because it has been explored at iteration 2. Therefore, the total number of attribute sequences can be reduced even further.

The **minimized total number of attribute sequences** at iteration $k$ with restriction **E5.10** is

$$tnrp''(k) = tnrp'(k) - tnrp'(k-1) \qquad \textbf{(E5.18)}$$

**Table 5-4** lists the values for the minimized total, $tnrp''(k)$, and the restricted total, $tnrp(k)$, for the first fifteen iterations. It also shows the ratio $q'_{np} = tnrp''(k)/tnrp(k)$, which reflects how much the number of attribute sequences is reduced when the minimized set of attribute sequences is explored.

**Table 5-4: Comparison between the total number of attribute sequences restricted by E5.10 and the optimized total number of attribute sequences for the first fifteen iterations**

| Iteration | $tnrp''(k)$ | $tnrp(k)$ | $q'_{np} = tnrp''(k)/tnrp(k)$ |
|---|---|---|---|
| 1 | 1 | 1 | 1.000000000 |
| 2 | 1 | 3 | 0.333333333 |
| 3 | 3 | 8 | 0.375000000 |
| 4 | 8 | 22 | 0.363636364 |
| 5 | 24 | 64 | 0.375000000 |
| 6 | 75 | 196 | 0.382653061 |
| 7 | 243 | 625 | 0.388800000 |
| 8 | 808 | 2055 | 0.393187348 |
| 9 | 2742 | 6917 | 0.396414631 |
| 10 | 9458 | 23,713 | 0.398852950 |
| 11 | 33,062 | 82,499 | 0.400756373 |
| 12 | 116,868 | 290,511 | 0.402284251 |
| 13 | 417,022 | 1,033,411 | 0.403539347 |
| 14 | 1,500,159 | 3,707,851 | 0.404589882 |
| 15 | 5,434,563 | 13,402,696 | 0.405482822 |

As we can see from **Table 5-4**, the number of attribute sequences can be reduced by an average of 60% of $tnrp(k)$.

The algorithm to construct the **minimized set of attribute sequences** (MATS) of all lengths up to $k$ is described by the procedure CONSTRUCTATSBROPT, which is similar to CONSTRUCTATSBR but with one modification done to the range of the loop in line 4. The upper limit for $Len(R_0)$ is set to the ceiling of $k/2$ so that only one of the sequences starting with the pair $(Len(R_0), Len(D_0))$ and $(Len(D_0), Len(R_0))$ is constructed, since such sequences are equivalent.

The recursive procedure CONSTRUCTRESTOFATSBROPT is similar to the procedure CONSTRUCTRESTOFATSBROPT with a conditional statement (lines 9-10) added to it. The condition in line 9 ensures that $Len(D_0)$ is at least as long as $Len(R_0)$, in order to avoid constructing attribute sequences that are equivalent.

```
CONSTRUCTATSBROPT (k:integer):array

1.  atsequences:array
2.  current_ats:array
3.  n_seq:array

4.  for initial_length := 1 to CEILING(k/2)
5.      current_ats[1] := initial_length
6.      CONSTRUCTRESTOFATSBROPT(current_ats, 1, 1, k, initial_length,
                                atsequences, n_seq)
7.  end for
8.  return (atsequences)
```

```
CONSTRUCTRESTOFATSBROPT( current_ats:array, ats_length:integer,
                         depth:integer, k:integer,
                         resolvent_length:integer, atsequences:array,
                         n_seq:array):integer

1.  if  resolvent_length = 0 then
2.  INCREMENT(n_seq[ats_length])
3.  ats_number := n_seq[ats_length]
4.  for ats_entry := 1 to ats_length
5.      atsequences[ats_length][ats_number][ats_entry] :=
                            current_ats[ats_entry]
6.  end for
7.  return
8.  end if
9.  if depth = 1 then start_base_length := resolvent_length
10. else start_base_length := 1
11. for base_length := start_base_length to k – depth + 2 – resolvent_length
12.     current_ats[depth + 1] := base_length
13.     CONSTRUCTRESTOFATSBROPT(current_ats, ats_length + 1, depth +1,
                                k, resolvent_length + base_length – 2,
                                atsequences, n_seq)
14. end for
```

**Table 5-5** lists the attribute sequences of MATS for iterations 1 to 4 where each set contains the sequences of length $k+1$ for $1 \le k \le 4$. For example, sequences number 3 to 5 belong to the set of sequences for iteration 3. The thirteen sequences are sorted by length.

**Table 5-5: Minimized set of attribute sequences up to iteration 4**

| | | | |
|---|---|---|---|
| ⟨1,1⟩ 1 | ⟨1,2,1⟩ 2 | ⟨1,2,2,1⟩ 3 | ⟨1,3,1,1⟩ 4 |
| ⟨2,2,1,1⟩ 5 | ⟨1,2,2,2,1⟩ 6 | ⟨1,2,3,1,1⟩ 7 | ⟨1,3,1,2,1⟩ 8 |
| ⟨1,3,2,1,1⟩ 9 | ⟨1,4,1,1,1⟩ 10 | ⟨2,2,1,2,1⟩ 11 | ⟨2,2,2,1,1⟩ 12 |
| ⟨2,3,1,1,1⟩ 13 | | | |

The value produced by evaluating $tnrp''(k)$ is the minimized number of attribute sequences at iteration $k$. The minimized total number of attribute sequences from 1 to $k$ is sum $tnrp''(1)+\cdots+tnrp''(k)$. The set $MATS(f,\Delta(S,1..k))$ is the minimum subset of the set $ATS(f,\Delta(S,1..k))$ that can be used as a guide to reduce the search paths without compromising completeness. Therefore, the value for $|MATS(f,\Delta(S,1..k))|$ when $f$ is the length of a clause, $S$ is a set of input clauses, $1..k$ is the range of iterations, is computed as

$$|MATS(Size,\Delta(S,1..k))| = \sum_{i=1}^{k} tnrp''(i). \qquad \textbf{(E5.19)}$$

### 5.3.5 Attribute sequences and binary factoring

The inclusion of binary factoring does not affect the number of attribute sequences because with respect to attribute sequences, the application of binary factoring is similar to performing a unit resolution. For example, consider the refutation search path, $sp = \langle (R_0, D_0), (R_1, D_1) \rangle$, of length 2. The corresponding attribute sequence can be either $ats_1 = \langle (1,2), (1,1) \rangle$ or $ats_2 = \langle (2,1), (1,1) \rangle$. In either case, one of either $R_0$ or $D_0$ is a unit clause. If $ats_2$ is the attribute sequence corresponding to $sp$, then $Len(R_0) = 2$. If $R_0$ has a factor, say $F_{R_0}$, then $Size(F_{R_0}) = 1$. Suppose that the search path $sp' = \langle (R_0, R_0), (F_{R_0}, D_1') \rangle$ is a refutation search path. Since the first pair, $(R_0, R_0)$, contains the same clause, it

implies by notational convention that this pair represents the application of binary factoring. We write the corresponding attribute sequence for $sp'$ as

$$ats' = \langle (Len(R_0), -1), (Len(F_{R_0}), Len(D_1')) \rangle = \langle (2,-1),(1,1) \rangle .$$

We use the notation $\langle (2,-1),(1,1) \rangle$ to indicate that the length of the first element of the first pair is reduced by one. By comparing $ats'$ with $ats_2$, we notice that the sequences are similar except for the negative sign. We extend the domain of the *absolute* function, ABS, to include attribute sequences as follows.

| If $ats = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$ is an attribute sequence, where $x_i$ and $y_i$ are integers, then $\text{ABS}(ats) = \langle (\text{ABS}(x_i), \text{ABS}(y_i)) \rangle$ for all $1 \le i \le n$. |
|---|

We can now write $\text{ABS}(ats') = \text{ABS}(ats_2)$. We call those attributes sequences similar. In general, **similar attribute sequences** are attribute sequences whose absolute values are equal.

With the use of the definition of similar attribute sequences, we can show that the attribute sequences generated by the CONSTRUCTATSBR procedure are effective even when binary factoring is employed.

**Theorem 5.1:**

*If an attribute sequence, ats, corresponds to a refutation search path, sp, within iteration k such that sp includes binary factoring, then ABS(ats) belongs to the minimized set of attribute sequences, MATS, of length k.*

**Proof:**

Let $sp = \langle (R_0, D_0), \ldots, (R_{k-1}, D_{k-1}) \rangle$, where $R_i$, for $1 \le i \le k-1$, is either a resolvent or a factor, such that if $R_1$ is not a factor of $R_0$, then $Size(R_0) \le Size(D_0)$. In other words, if the first pair represents a binary resolution, then $R_0$ is the smaller clause between $R_0$ and $D_0$. This does not affect the completeness of any search strategy employing the attribute sequences concept to reduce the search space because as we have demonstrated, reversing the order of the initial clauses produce equivalent search paths. Let *ats* be the corresponding attribute sequence to *sp*, then $ats = \langle (Len(R_0), Len(D_0)), \ldots, (Len(R_{k-1}), Len(D_{k-1})) \rangle$. If $R_i$ is a factor, then $(R_{i-1}, D_{i-1})$ represents binary factoring and thus, $D_{i-1} = R_{i-1}$. The corresponding pair in the attribute sequence is $(Len(R_{i-1}), -1)$. Let $ats' = \text{ABS}(ats)$. We want to prove that $ats' \in \text{MATS}$. We know that, as long as $Len(R_0) \le Len(D_0)$, the set MATS contains all the attribute sequences corresponding to refutation search paths. Suppose that $ats' \notin \text{MATS}$, then $ats'$ is an attribute sequence not in MATS but corresponds to some refutation search path. This implies that MATS does not contain all the attribute sequences that correspond to refutation search paths which contradicts the fact that MATS contains all the attribute sequences corresponding to the refutation search paths. Therefore, $ats' \in \text{MATS}$. □

From Theorem 5.1 we conclude that we do not need to explicitly generate the attribute sequences that include binary factoring. For every pair, except for the last pair, within an attribute sequence where there is a 1 as the second element, i.e., pair of the form $(*,1)$, either a unit resolution or binary factoring may be selected as the inference rule to be applied. Even though the sequences need not be explicitly generated, this does not mean that the possible search paths and consequently, the explorable search space does not increase.

### 5.3.6 Attribute sequences and other inference rules

We have demonstrated that when using only binary resolution and binary factoring as inference rules, the number of attribute sequences can be reduced significantly by restricting the lengths of the initial clause and side premises. We now show how other inference rules, such as demodulation and paramodulation, can be added.

In demodulation, the length of the conclusion is equal to the length of the demodulated clause. With respect to an attribute sequence, this is similar to a binary resolution of two clauses where one of them has a length of 2. For example, suppose that the attribute sequence $ats_1 = \langle (2,3),(3,1),(2,1),(1,1) \rangle$ corresponds to a refutation search path where only binary resolution is performed. Suppose the attribute sequence $ats_2 = \langle (1,3),(3,1),(2,1),(1,1) \rangle$ corresponds to a refutation search path where the first inference rule is demodulation and the next three inference rules are binary resolutions. The 1 in the pair (1,3) is the length of the demodulator. The two sequences $ats_1$ and $ats_2$ are identical everywhere except for the first integer of the first pair. In binary resolution, if one of the premises is a clause of length 2, then the resolvent maintains the length of the other premise. In demodulation, the demodulator's length is always 1 and the demodulant's length is always the same length as the demodulated clause. Therefore, we can construct all the attribute sequences that include demodulation by simply copying all the sequences in MATS and then replacing the second element of every pair $(*,2)$ by 1 as shown in **Table 5-6**.

**Table 5-6: Attribute sequences up to iteration 4 with binary resolution and demodulation as inference rule**

| Binary Resolution | Binary Resolution and Demodulation |
|---|---|
| ⟨(1,1)⟩ | |
| ⟨(1,2),(1,1)⟩ | ⟨(1,1),(1,1)⟩ |
| ⟨(1,2),(1,2),(1,1)⟩ | ⟨(1,1),(1,2),(1,1)⟩ , ⟨(1,2),(1,1),(1,1)⟩ , ⟨(1,1),(1,1),(1,1)⟩ |
| ⟨(1,3),(2,1),(1,1)⟩ | |
| ⟨(2,2),(2,1),(1,1)⟩ | ⟨(2,1),(2,1),(1,1)⟩ |
| ⟨(1,2),(1,2),(1,2),(1,1)⟩ | ⟨(1,1),(1,2),(1,2),(1,1)⟩ , ⟨(1,2),(1,1),(1,2),(1,1)⟩ , ⟨(1,2),(1,2),(1,1),(1,1)⟩ , ⟨(1,1),(1,1),(1,2),(1,1)⟩ , ⟨(1,1),(1,2),(1,1),(1,1)⟩ , ⟨(1,2),(1,1),(1,1),(1,1)⟩ , ⟨(1,1),(1,1),(1,1),(1,1)⟩ |
| ⟨(1,2),(1,3),(2,1),(1,1)⟩ | ⟨(1,1),(1,3),(2,1),(1,1)⟩ |
| ⟨(1,3),(2,1),(1,2),(1,1)⟩ | ⟨(1,3),(2,1),(1,1),(1,1)⟩ |
| ⟨(1,3),(2,2),(2,1),(1,1)⟩ | ⟨(1,3),(2,1),(2,1),(1,1)⟩ |
| ⟨(1,4),(3,1),(2,1),(1,1)⟩ | |
| ⟨(2,2),(2,1),(1,2),(1,1)⟩ | ⟨(2,1),(2,1),(1,2),(1,1)⟩ , ⟨(2,2),(2,1),(1,1),(1,1)⟩ , ⟨(2,1),(2,1),(1,1),(1,1)⟩ |
| ⟨(2,2),(2,2),(2,1),(1,1)⟩ | ⟨(2,1),(2,2),(2,1),(1,1)⟩ , ⟨(2,2),(2,1),(2,1),(1,1)⟩ , ⟨(2,1),(2,1),(2,1),(1,1)⟩ |
| ⟨(2,3),(3,1),(2,1),(1,1)⟩ | |

**Table 5-6** shows all the attribute sequences when binary resolution and demodulation are employed. The column labeled "Binary Resolution" contains a list of the attribute sequences from MATS where binary resolution is the only inference rule. The second column lists all the attribute sequences that can be obtained from the sequence in the first column by replacing the occurrences of the value 2 by 1 in the second element of a pair. We conclude from this table and from the relation between the length 2 of an input clause in binary resolution and the length of the demodulator that, even though the number of attribute sequences increases in proportion of the number of pairs that contain the value 2 as the second element, it is not necessary to explicitly construct the attribute sequences for the inclusion of demodulation. Every time the value 2 occurs as the second element of a pair, we can simply chose either to perform a binary resolution with a input clause of length 2 or perform a demodulation. The inclusion of binary factoring does not affect the inclusion of demodulation so binary factoring can

still be used with demodulation without the need to construct additional attribute sequences.

The application of paramodulation results in a paramodulant's length that is equal to the sum of the lengths of the premises minus one. If the paramodulator's length is 1, then we can use the same attribute sequences for demodulation, otherwise the possible attribute sequences must be added.

Attribute sequences help restrict the search to paths that may lead to a refutation and avoid the paths that definitely cannot reach the empty clause. Attribute sequences also help in controlling the amount of application of particular inference rules.

## 5.4 Summary

In this chapter we analyzed the size of the explorable search space of SLR from two perspectives:

1. the maximum number of generated clauses (**E5.5**).
2. the number of attribute sequences where the attribute is the clause length.

We derived two formulas that can be used to calculate the number of attribute sequences in two cases. The first formula, **E5.14**, gives the number of attribute sequences when a restriction is imposed on the length of a side premise such that completeness is not compromised. The second formula, **E5.16**, gives the number of attribute sequences when no restrictions are imposed on the length of a side premise. We analyzed the values from those formulas for the first 15 iterations. We found that when restrictions are imposed on the lengths of the clauses, the reduction in the size of the explorable search space is exponential in the iteration depth. Therefore, using ATS with length restriction imposed on the side premises improves the efficiency of SLR substantially.

CHAPTER

# 6

# CARINE: An Implementation of SLR

CARINE is an ATP that implements SLR. We developed CARINE to study the performance of SLR in practice. In this chapter, we present CARINE and discuss briefly the data structures used in it. We provide examples that demonstrate how delayed clause-construction and attribute sequences can be used to improve the inference rate and prune the search space. We then provide test results from experiments that we conducted to determine the effect of DCC in practice. The experiments produced results on the percentage of time spent constructing clauses, the percentage of successful unifications, and the ratio of unit conflict tests in a selected number of theorems from the TPTP v2.6.0 library. We analyze those results and discuss their relationship with the inference rate speedup. We provide remarks on when the use of DCC can be most effective in practice. Finally, we compare the effects of DCC with ATS on SLR.

## 6.1 Overview

CARINE is an experimental resolution-based automated theorem prover developed for the following reasons:

- to demonstrate how delayed clause-construction may be implemented efficiently using simple data structures

- to empirically show that the performance gained by using delayed clause construction is significant
- to show how simple it is to integrate DCC within semi-linear resolution
- to depict the potential of semi-linear resolution when attribute sequences are used to restrict the search and improve the overall efficiency of the theorem prover

At the highest abstraction level, the design of CARINE, as shown in **Figure 6-1**, is quite simple.

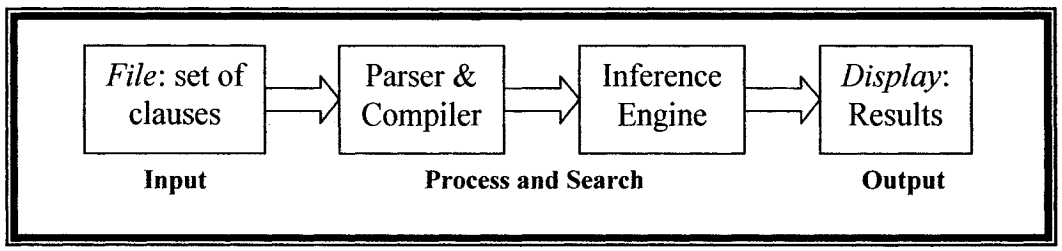



**Figure 6-1: Design of CARINE.**

The input is a file containing a set of input clauses. Once the clauses are read, they are parsed and compiled into the appropriate data structures and presented to the inference engine. Then the inference engine searches for a proof within a user defined time limit. The search terminates either when a proof is found or when the time limit expires. At that point, performance and statistical results, as well as a proof (if one is found) are displayed.

## 6.2 Definitions

We use the following definitions in this chapter to describe certain aspects of the implementation in a formal manner.

An **input file** is a text file containing a set of input clauses that CARINE reads, parses, compiles and uses to derive the empty clause. An ***obj*** can represent any of the following: a term, a literal, a clause or a substitution set. A **container** can be any of the following: an input file, a list of *obj*s, a single or a multi-dimensional array, or a string.

| | |
|---|---|
| $NArgs(L)$ | is a function that returns the number of arguments (arity) of the predicate in the literal $L$. |
| $NArgs(t)$ | is a function that returns the arity of a term $t$. |
| $Sign(L)$ | is a function that returns either -1 or 1 depending on whether the literal $L$ is either negative or positive respectively. |
| $Pred(L)$ | is a function that returns the predicate symbol of the literal $L$. |
| $Index(cont,obj)$ | is a function that returns the index of *obj* within the container *cont* such that *cont* exists as a structure in memory and not in the input file. The index may be a single integer or a tuple of integers depending on the type of *cont*. |
| $InpOcc(cont,obj)$ | a function that returns the occurrence of *obj* within the container *cont* such that *cont* is part of the input file. |

The difference between $Index(cont,obj)$ and $InpOcc(cont,obj)$ is the location of cont. In $Index(cont,obj)$, *cont* is in memory whereas in $InpOcc(cont,obj)$, *cont* is in a file on an external storage.

**Example 6.1:**

$NArgs(B(x,a,y))=3$, $NArgs(\neg Q)=0$,

$NArgs(a)=0$, $NArgs(f(a,b,x))=3$, $NArgs(x)=0$.

$Sign(D(x,y,z,z)=1$, $Sign(\neg Q)=-1$.

$Pred(B(x,y,z))=B$, $Pred(\neg P)=P$.

Given the array *predicates* = [*B*,*Q*,*P*,*R*]

$Index(predicates, P) = 3$,

$Index(predicates, B) = 1$,

$Index(predicates, G) = 0$, since $G$ is not in *predicates*.

Given the clause $C = \neg Q \vee P \vee W$

$Index(C, P) = 2$, $Index(C, \neg Q) = 1$.

Given the input file *IFile* containing the clauses:

$\neg P \vee Q$, $P \vee R$, $P \vee Q \vee W$

$InpOcc(IFile, \neg P \vee Q) = 1$,

$InpOcc(IFile, P \vee Q \vee W) = 3$,

$InpOcc(IFile, P \vee Q) = 0$,

$InpOcc(P \vee Q \vee W, Q) = 2$,

$InpOcc(\neg P \vee Q, \neg P) = 1$.

## 6.3 Data Structures

CARINE is implemented in ANSI C and therefore our description of its data structures and its algorithms is closely related to a procedural language such as C. We divide the presentation of CARINE's data structures into two sections. The first section describes the data structures for the essential elements that are common to almost all ATPs, including terms, literals, clauses and substitution sets. The second section describes the elements, which may or may not exist in other ATPs, that are related to the enhancement of the search procedures used in CARINE, including, the path table, the lookup tables, clause partitioning lists, and clause grouping lists.

### 6.3.1 Terms, literals, clauses and substitution sets

The data structures used in CARINE for the essential elements found in most theorem provers are quite simple. Terms are stored in a variation of the flatterm representation. We use an array of elements each having two fields. Every element contains a reference code to the term symbol it represents and a pointer to the next argument of a function or a literal. The last argument points to NULL.

**Figure 6-2** shows the term representation of $g(x,h(a),f(x,h(a)))$ in CARINE (the circle with a cross in it represents NULL).

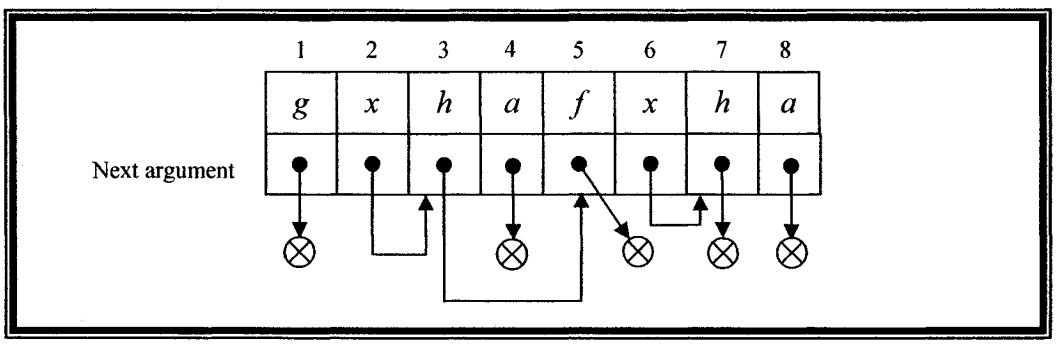



**Figure 6-2: Flatterm representation of $g(x,h(a),f(x,h(a)))$ in CARINE.**

Literals are stored as arrays of terms. **Figure 6-3** shows an example of a literal representation in CARINE.

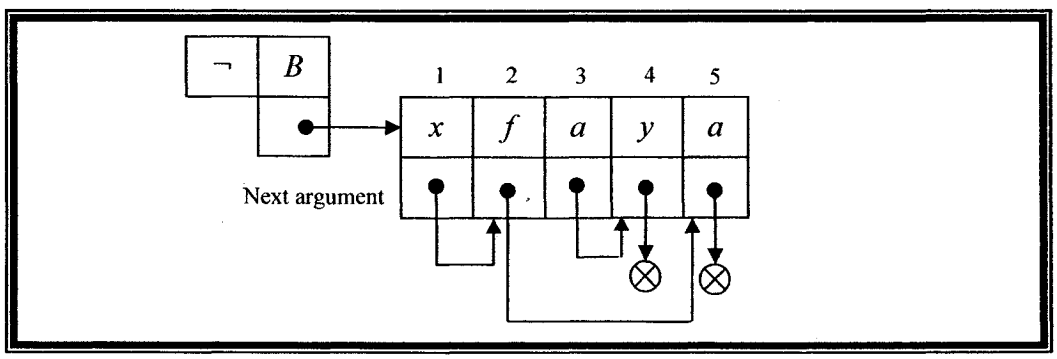



**Figure 6-3: Literal representation of $\neg B(x,f(a,y),a)$ in CARINE.**

Input clauses are read from the input file, parsed, compiled and stored as arrays of pointers to literals. With such representation, literals can be ordered and accessed more quickly by comparison with a linked list representation. **Figure 6-4** shows an example of a clause representation. The numbers under the variables in **Figure 6-4** are tags used by the ATP to identify the distinct variables within a clause. The tags are determined based on the first occurrence (when the clause is read from left to right) of a variable in a clause. For example, in **Figure 6-4**, $x$ is the first variable in clause $C$ so it is given the tag 1. Every other occurrence of $x$ is also given the tag 1. The variable $y$ is the second variable in $C$ so it is given the tag 2.

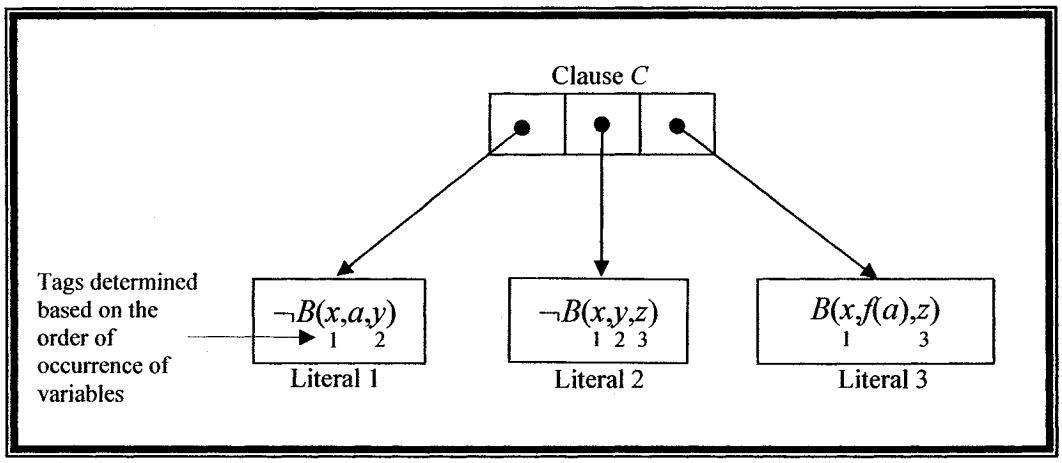



**Figure 6-4: Clause representation in CARINE.**

The tags play an important role in renaming the variables of a clause as we shall demonstrate later.

Recall that one of the conditions for DCC to be sound is to rename the variables of the input clauses used in a derivation so that no variable appears in more than one side premises. This implies that every side premise in a linear derivation should be a variant of an input clause. In an actual implementation, a variant is not constructed because the construction is time consuming.

Instead of constructing a variant of an input clause that is selected to participate in a linear derivation, a temporary integer code, called a **relative clause identification code** denoted by *RCid*, is assigned to the input clause to

identify the clause within the derivation. The renaming of variables is performed on a *as-needed* basis, and is achieved as follows. When a unification of terms or literals is performed, the tag attached to a variable is added to the *RCid* to obtain a unique identification code for the variable. This unique code, called the **variable identification code** denoted by *Vid*, makes the variable distinct from all the variables in the rest of the side premises in a linear derivation. This way there is no need to construct a variant of an input clause. We now formally define *RCid* and *Vid* and then provide examples to clarify their purpose.

Any input clause, a clause from *Goals* (see SLR in Chapter 4), or a clause from the set $T$ (see SLR in Chapter 4) is a constructed clause. Let *MVC* be a constant, set by the user or automatically determined by an ATP, that denotes the maximum number of variables per clause that an ATP can handle. Any constructed clause $C$ that is introduced into a derivation at depth $d$ is given a **relative clause identification code**, denoted by $RCid_C(d)$, which is unique with respect to all the clauses within the derivation and is determined by the formula

$$RCid_C(d) = \begin{cases} 0 & \text{if } C \text{ is an initial clause (i.e., } C_{init}\text{),} \\ MVC \times (d+1) & \text{otherwise,} \end{cases} \qquad \textbf{(E6.1)}$$

Currently, we set $MVC = 32$ because most theorems in the TPTP version 2.6.0 problem library do not contain any clauses containing more than 32 distinct variables.

Let $VOffset(v, C)$ be the offset code tagged to a distinct variable $v$ in a clause $C$ based on the order of occurrence of $v$ with respect to the other variables in $C$. The **variable identification code**, denoted by $Vid(v, C, d)$, of a distinct variable $v$ in a clause $C$ introduced at depth $d$ within a derivation is calculated by the formula

$$Vid(v,C,d) = RCid_C(d) + VOffset(v,C) \qquad \textbf{(E6.2)}$$

For example, suppose the clause in **Figure 6-4** is selected twice in a derivation; once at depth $d = 0$ as a side premise and another at depth $d = 4$. Then with $MVC = 32$, the variable identification codes for the variables $x, y, z$ of $C$ are:

$d = 0$:

$$Vid(x,C,0) = 32 \times (0+1) + 1 = 33,$$

$$Vid(y,C,0) = 32 \times (0+1) + 2 = 34,$$

$$Vid(z,C,0) = 32 \times (0+1) + 3 = 35,$$

$d = 4$:

$$Vid(x,C,4) = 32 \times (4+1) + 1 = 161,$$

$$Vid(y,C,4) = 32 \times (4+1) + 2 = 162,$$

$$Vid(z,C,4) = 32 \times (4+1) + 3 = 163.$$

With the above identification method, any clause can be used in the same derivation several times without performing any actual copies in memory of the clause. Renaming its variables is done in almost constant time, because once the clause identification code changes, the identification codes of the distinct variables within the clause are changed automatically.

A substitution set is abstractly represented as a directed graph. It is implemented as a one dimensional array with two fields per element as shown in **Figure 6-5**. In this example, the substitution set is $\{x \to f(y), y \to g(z,w), z \to a\}$. The substitution set represented in the bottom half of the **Figure 6-5** is read as follows. The variable $x$ belongs to some clause, which in this example is not important. What is important is the *RCid* of the clause that contains the substitution term for $x$. The *RCid* of this clause is 32. The variable $y$ is substituted by the term $g(z,w)$. This term belongs to the clause whose *RCid* is 64. The variable $w$ has no

substitution term. Since the field *RCid* refers to the clause that contains the substitution term, then for *w* there is no *RCid*. A variable that does not have a substitution term contains –1 in its *RCid* field and a pointer to NULL in its substitution term field. Notice that the variable identification codes in **Figure 6-5** are the same as the array indices. This makes the access to a variable's substitution term a constant time operation.

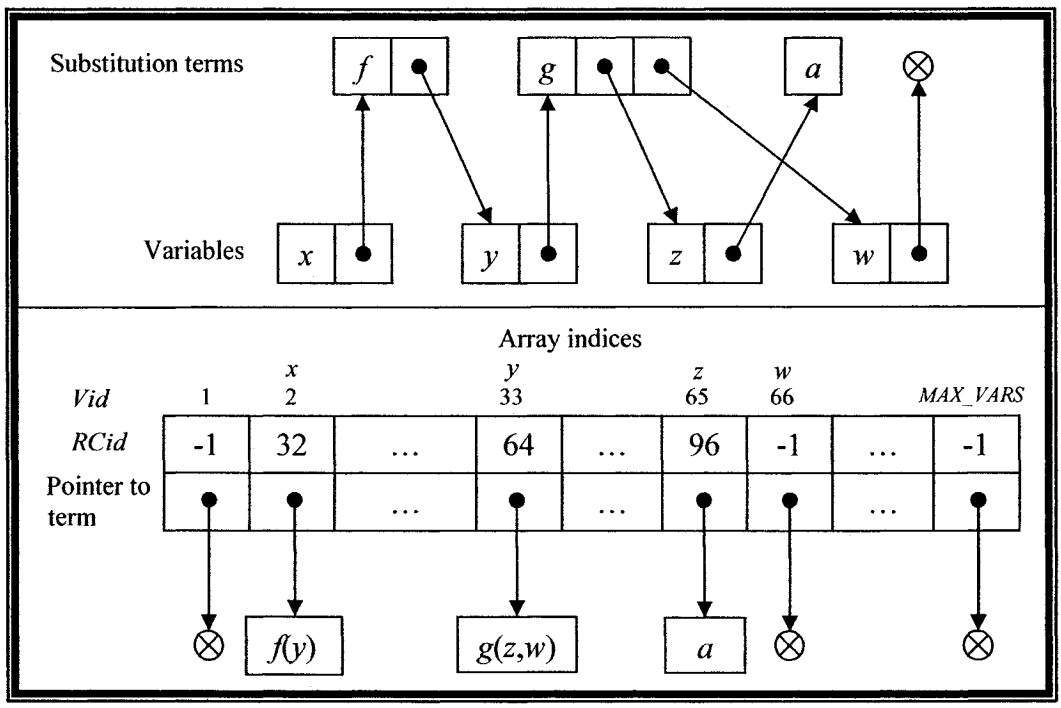



**Figure 6-5: Representation of the substitution set** $\{x \to f(y), y \to g(z,w), z \to a\}$ **as a directed graph (top) and as an array (bottom) in CARINE.**

**Example 6.2:**

$B_1 = G(x) \vee P(x)$, $B_2 = \neg P(f(x)) \vee Q(x)$,

$B_3 = \neg Q(g(x,y)) \vee R(x)$, $B_4 = \neg R(a)$

$VOffset(x, B_1) = 1$, $VOffset(x, B_2) = 1$,

$VOffset(x, B_3) = 1$, $VOffset(y, B_3) = 2$.

The derivation in **Figure 6-6** shows the role of *RCid* and *Vid*. The *RCid* is calculated for the new clause when the clause is introduced into the derivation. However, to save time, the *Vid* is calculated for a variable only when the variable is needed. There is no need to calculate the *Vid*s for all the variables in a clause if, for instance, only one is needed for the unification of two literals.

Although the variable $x$ is used in the clauses $B_1, B_2, B_3$, when those clauses are introduced into the derivation the symbol $x$ is no longer important. CARINE identifies the variables by their *Vid*s not by their symbols. However, we wrote each variable in a substitution set with its symbol first followed by its *Vid* for convenience. For instance, $\ddot{\sigma}_1 = \{x1 \rightarrow f(x33)\}$ is viewed by CARINE as $\ddot{\sigma}_1 = \{1 \rightarrow f(33)\}$. The *Vid* renaming guarantees that no two clauses from the side premises in a linear derivation share the same variable.

The resolvents $\dddot{R}_1$ and $\dddot{R}_2$ are not constructed, so we did not substitute the variables with their substitution terms. $\dddot{R}_3$ is first generated and then constructed (with variables normalized) by applying the p-idempotent substitution set $\ddot{\sigma}_{1..3} = \ddot{\sigma}_1 \cup \ddot{\sigma}_2 \cup \ddot{\sigma}_3$.

The identification codes given to the participating clauses in the derivation are calculated as follows.

$RCid_{B_1}(0) = 0$, since it is the initial clause.

$RCid_{B_2}(0) = 32 \times (0+1) = 32$, this is a side premise at depth 0.

$RCid_{B_3}(1) = 32 \times (1+1) = 64$, and $RCid_{B_4}(2) = 32 \times (2+1) = 96$.

The identification codes for the variables are

$Vid(x, B_1, 0) = 0 + 1 = 1$, $Vid(x, B_2, 0) = 32 + 1 = 33$,

$Vid(x, B_3) = 64 + 1 = 65$, $Vid(y, B_3) = 64 + 2 = 66$.

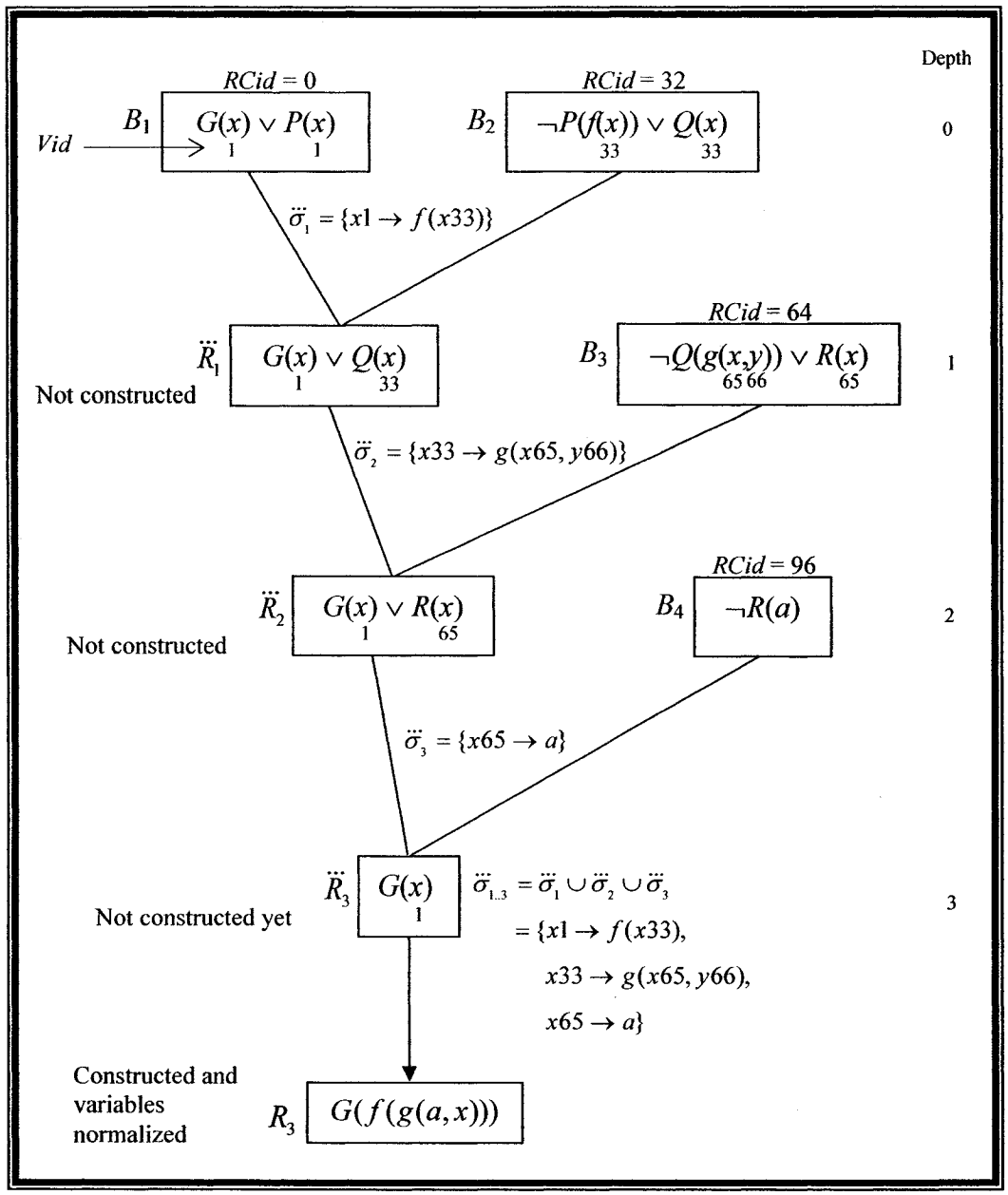



**Figure 6-6: An example of a derivation showing the role of *RCid* and *Vid* of the distinct variables of the participating clauses.**

### 6.3.2 The path table

The path table keeps track of the search state and retains all the necessary information to backtrack, to construct a clause that has been delayed, or to determine whether a refutation has been reached. The path table is almost a straight forward implementation of the delayed clause construction as presented in Chapter 3. Recall the *general formula* for a derived clause at depth *i*

$$\ddot{C}(I_i) = ((B^1_{r_1} \cup \cdots \cup B^m_{r_m}) \setminus (\alpha_{1..i} \cup \beta_{1..i}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i}),$$

where $B^1_{r_1}, \cdots, B^m_{r_m}$ are constructed clauses, and $\alpha_{1..i}$ and $\beta_{1..i}$ are the multisets of deleted literals.

We can readily envision this formula implemented as a table with *MAX_DEPTH* columns where the combined information from the first column up to the $i^{th}$ column determine the conclusion $\ddot{C}(I_i)$ as shown in **Figure 6-7**.

The "*Inference rule*" row indicates the inference rule applied at each step. The "*Newly introduced clause*" row contains pointers to the clauses that are introduced at each depth of a derivation. The "*Literal deleted from the newly introduced clause*" row contains a pointer to the literal that is deleted from the newly introduced clause. Notice that in binary factoring no new clause is introduced and hence, we leave the corresponding entries of the "*Newly introduced clause*" and the "*Literal deleted from the newly introduced clause*" fields empty (or pointing to NULL).

The "*Previous clause*" row contains pointers to the clauses that have been introduced earlier in the table. When an inference rule is applied at depth *i* one of the literals from the clauses introduced at depth $j < i$ is either modified or deleted. This literal is not actually deleted but marked as deleted. The row "*Literal deleted or modified from previous clause*" is used for this purpose. A pointer to the deleted literal is entered in this row at column *i* to indicate that this literal is marked as "deleted".

| Depth | 1 | 2 | ... | *i* | ... | *MAX_DEPTH* |
|---|---|---|---|---|---|---|
| Inference rule | | | | | | |
| Newly introduced clause | | | | | | |
| *RCid* of the newly introduced clause | | | | | | |
| Literal deleted from newly introduced clause | | | | | | |
| Previous clause [introduced at depth] | | | | | | |
| Literal deleted or modified from previous clause | | | | | | |
| Modified variables | | | | | | |
| Length of resolvent/factor | | | | | | |
| Merge clause | | | | | | |
| Delayed | | | | | | |

**Figure 6-7: The path table.**

The "*Modified variables*" row lists only the variables that are bound to terms at each depth. To determine the set of all the variables within a derivation that are bound to terms, we have to perform the union of the entries of this row.

The "*Length of resolvent/factor*" row contains the length of the resolvent or factor at the current depth. The "*Merge clause*" row indicates whether the factor is a merge clause or not. The "*Delayed*" row indicates whether the clause is marked for construction at a later time based upon the criteria discussed in Chapters 3 and 4.

### 6.3.3 Lookup tables

There are several operations that require a time which is linear, quadratic or even exponential in the number of elements existing in the set on which the operations are applied. Some of these operations can be reduced to a constant time through the use of lookup tables. For example, we can build lookup tables that maintain

information about clauses that produce resolvents, clauses that have factors and literals that unify with each other and then every time we need to select a clause $C$ to participate in the derivation, we first consult the lookup table to determine whether $C$ produces any factors or resolves with the clauses already in the path table.

In CARINE, there are three static lookup tables that are constructed after the input clauses are compiled and remain unchanged during the whole search, and one dynamic table that changes during the search process. The static tables are:

**clause-to-clause resolution table**. This table contains information on whether a pair of input clauses produces resolvents or not.
**clause-factors table**. This table contains information on whether an input clause has any factors.
**literal-to-literal unification table**. This table contains information on whether any pair of literals unify or not.

The dynamic table is similar to the literal-to-literal unification table except that it maintains information about the literal unifications of the literals of derived clauses (i.e., not input clauses) within the set *Goals* (see Chapter 4) and the literals of the input clauses.

### 6.3.4 Clause partitioning and clause grouping lists

Partition and group lists are sets of clauses that share a common characteristic. The difference between partitions and groups is that the intersection of any two partitions is empty, which is not necessarily the case in groups. In CARINE, input clauses are partitioned according to their sizes. Unit clauses, whether they are input clauses or derived clauses, are partitioned into lists, called **unit predicate lists**, according to their predicate symbols. Each predicate list is either a list of the

negative unit clauses or the positive unit clauses of the predicate in question, as shown in the example of **Figure 6-8**.

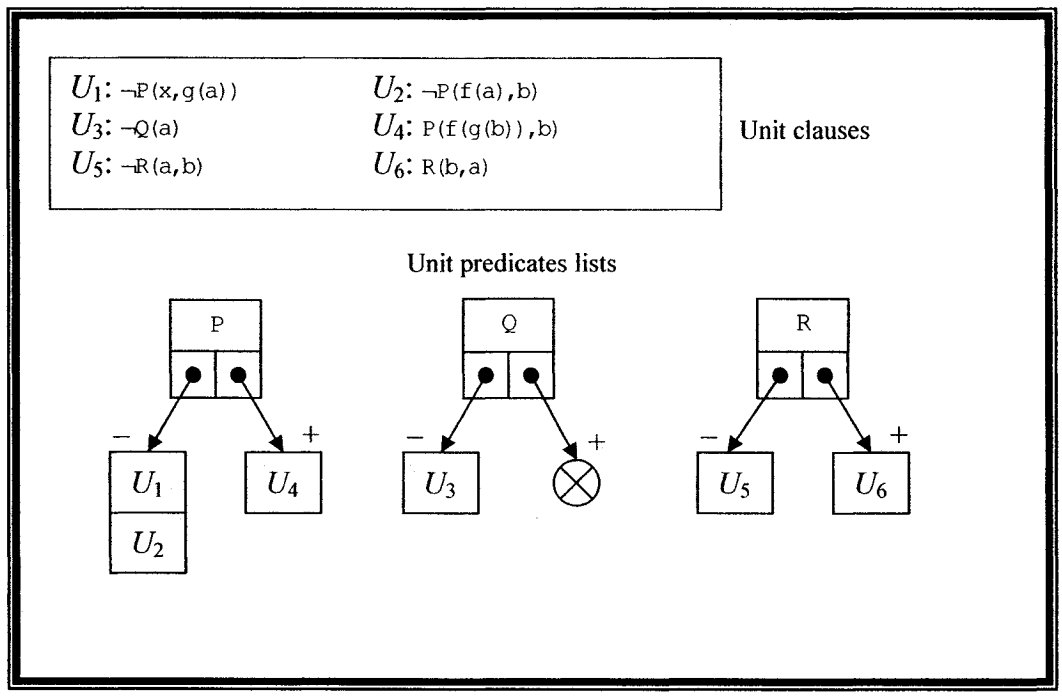



**Figure 6-8: An example of the partitioning of unit clauses (unit predicate lists) in CARINE.**

For instance, the predicate Q has a pointer to the list of unit clauses that contain a negative literal of Q. In our example, only one unit clause, namely $U_3$, belongs to this list. On the other hand, there are no positive unit clauses of Q and thus the list is empty.

Similar to the unit predicate lists, the input clauses are grouped by predicates forming predicate lists as shown in the example of **Figure 6-9**.

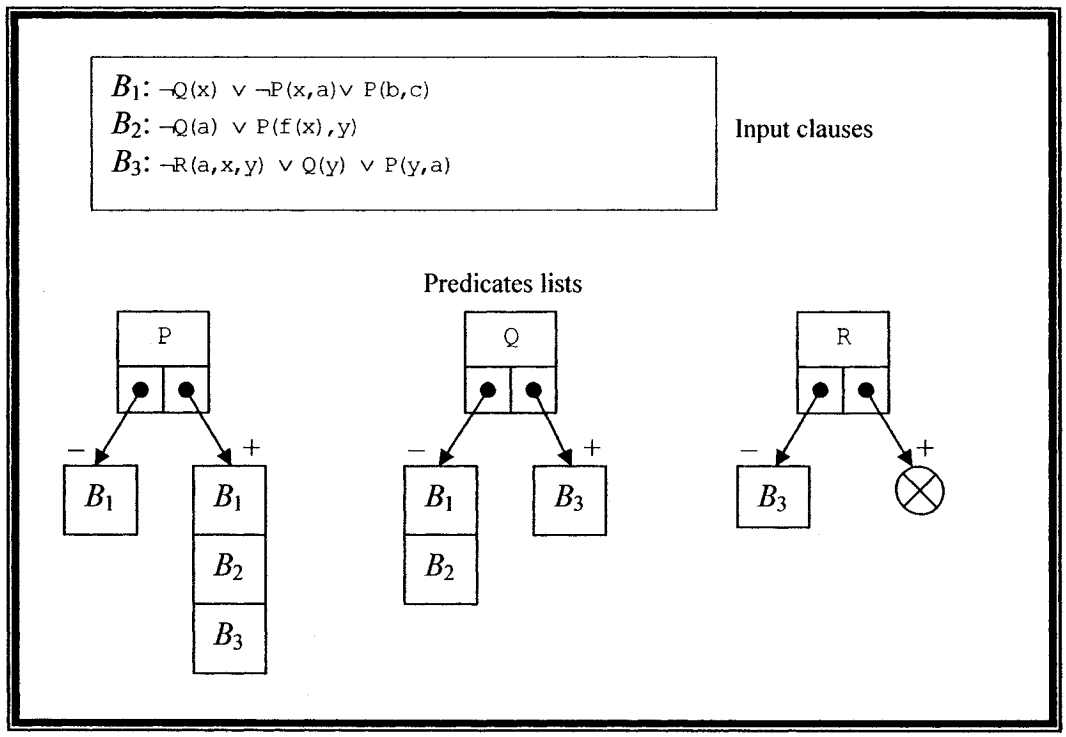



**Figure 6-9: An example of the grouping of input clauses in CARINE.**

The structures holding information about the predicates P, Q, and R have pointers to two lists of clauses; the negative and positive lists. All the clauses having at least one negative literal of the predicate P, Q or R belong to the negative list of the corresponding predicate. Similarly, all the clauses having at least one positive literal of the predicate P, Q or R belong to the positive list of the corresponding predicate.

Since SLR is refutationally complete, as demonstrated in Chapter 4, any clause that contains at least one literal that does not resolve with any other clause can be removed from the set of retained clauses. This procedure is called **pure literal clause deletion** [Plaisted & Zhu 1999]. Using the grouping of the clauses as described above it is easy to determine which clauses may be removed without affecting the completeness of SLR. In the example of **Figure 6-9**, $B_3$ has the literal ¬R(a,x,y) that does not resolve with any other literal from any other clause.

This can be quickly noticed by simply checking the positive list of the predicate R. Since it is empty then there exist no clause that can be resolved with $B_3$ over the literal ¬R(a,x,y) and therefore, $B_3$ can be eliminated from the set of input clauses. By eliminating $B_3$, the positive list of the predicate Q becomes empty leading to the predicates lists state shown in **Figure 6-10**.

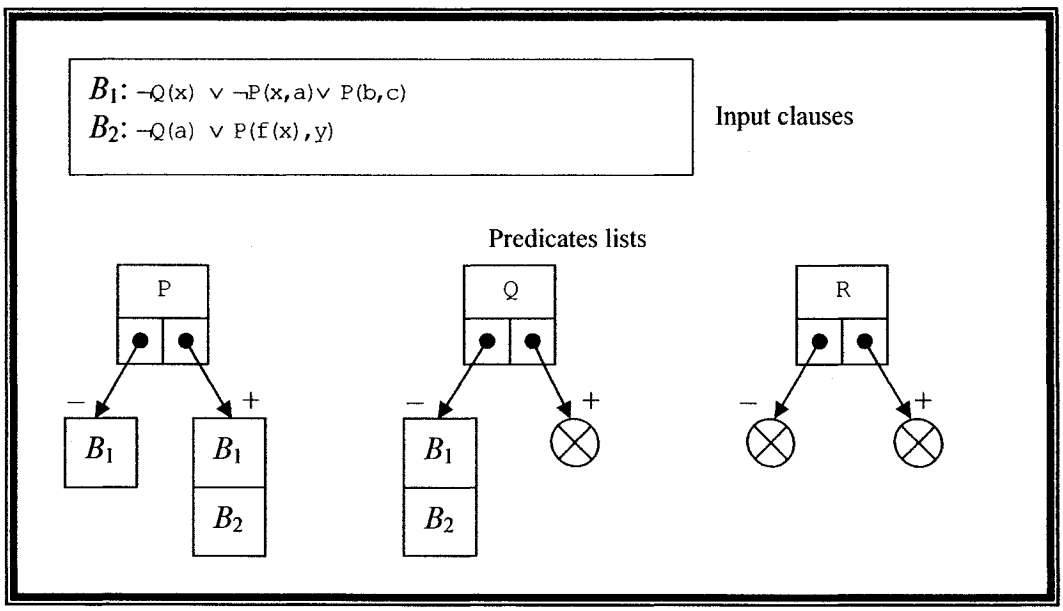



**Figure 6-10: Predicate lists after $B_3$ is deleted from the input clauses.**

Similarly, we can remove $B_1$ and $B_2$ from the set of input clauses because the positive list of the predicate Q is empty and thus no clause can resolve with either $B_1$ or $B_2$ over the literals ¬Q(x) or ¬Q(a()) respectively. By removing $B_1$ and $B_2$, there will be no more clauses in the set of input clauses and therefore, the set is satisfiable.

The process of pure literal deletion is implemented in CARINE and it is determined from the examination of the predicates lists as described in the above example.

### 6.3.5 Literal grouping

A literal grouping is a set of literals of retained clauses that share a common characteristic. In CARINE, a literal group, called **literal instance list**, constitute one literal that belongs to an input clause and forms the head of the list, and the rest of the literals, forming the tail of the list, are instances of this literal. Every distinct literal from the input clauses is the head of a literal instance list. The tail of the list is formed of literals that belong to derived constructed clauses.

The literal instance lists are useful for determining whether two literals may unify or not. This is achieved by checking the lookup tables in $O(1)$ time whether the head of the lists of the two literals unify or not. If they don't unify then their instances are definitely not going to unify.

The literal instance lists are helpful for determining potential unit resolutions. If the head of the list does not unify with a literal, $L$, of a clause then none of the literals from the tail is going to unify with the literal $L$.

### 6.3.6 Literal ordering

Sorting the literals according to some ordering relation is common among automated theorem provers because it facilitates the comparison of clauses and provides a faster way to prioritize the selection of clauses for the participation in particular inference rules.

In CARINE, the literals in each clause are partitioned into negative and positive literals with the negative literals listed first. Each partition is then sorted according to the arity of the predicate. If two predicates have the same arity then they are sorted lexicographically according to their predicate symbols and if they have the same predicate symbol then they are ordered according to their occurrence within the input clause within the input file. Formally, let *PredTable*

be a lexicographically sorted list of all predicate symbols used in some given input file *InputFile*. Suppose $L_1$ and $L_2$ are two literals from an input clause $C$ belonging to *InputFile*, $L_1$ is less than a $L_2$, denoted by $L_1 \prec L_2$, if one of the following conditions applies:

1. $Sign(L_1) < Sign(L_2)$
2. $Sign(L_1) = Sign(L_2)$ and $NArgs(L_1) < NArgs(L_2)$
3. $Sign(L_1) = Sign(L_2)$ and $NArgs(L_1) = NArgs(L_2)$ and $Index(PredTable, Pred(L_1)) < Index(PredTable, Pred(L_2))$
4. $Sign(L_1) = Sign(L_2)$ and $NArgs(L_1) = NArgs(L_2)$ and $Index(PredTable, Pred(L_1)) = Index(PredTable, Pred(L_2))$ and $InpOcc(C, L_1) < InpOcc(C, L_2)$.

Predicate, function, constant and variable symbols are stored in a symbol table (array) along with information about the arity of each predicate and function.

## 6.4 An Example of SLR with ATS

We present in this section an example of SLR using DCC as performed in CARINE, with some detail demonstrating the role of the path table. We also demonstrate how ATS can reduce the search space explored by an SLR based ATP. Example 6.3 shows in detail iteration by iteration, all the derivations necessary to obtain a proof starting with iteration 1.

In the following example, the set *Goals* and its initial contents are indicated at the beginning of each derivation. We assume that all derived unit clauses that are not in *Goals* are added to *Goals*. The iteration number is held in the variable, *bound*, and the step number is held in the variable, *depth*. The overall substitution set of

the whole derivation is denoted by $\ddot{\sigma}$. The mgu of a local inference (i.e., one unification of literals) at depth *i* is denoted by $\ddot{\sigma}_i$. The inference rules are labeled BR for binary resolution and BF for binary factoring in the path table. The value YES is used in the "*Merge clause*" row to indicate that the generated clause is a merge clause. The value YES is used in the "*Delayed*" row of the path table to indicate that the clause is marked for construction at later time. A value of NO indicates that the generated clause is not marked for construction at a later time either because it has been constructed or it is not necessary to construct it. The decision on whether a clause should be marked for construction at later time or not is determined based on the criteria discussed in Chapters 3 and 4.

**Example 6.3:**

This example is problem SYN035-1 from the TPTP library. The theorem contains three input clauses and all clauses are used in the proof.

$S = \{B_1, B_2, B_3\}$ is the set of input clauses. $B_1$ is the negated conclusion.

| |
|---|
| $B_1$: ¬P(x1,f(x2,x1)) ∨ ¬P(f(x2,x1),f(x2,x1)) ∨ ¬Q(x2,f(x2,x1)) ∨ ¬Q(f(x2,x1),f(x2,x1))<br>$L_{11}$ $L_{21}$ $L_{31}$ $L_{41}$ |
| $B_2$: P(x1,x2)<br>$L_{12}$ |
| $B_3$: ¬P(x1,f(x2,x1)) ∨ ¬P(f(x2,x1),f(x2,x1)) ∨ Q(x2,x1)<br>$L_{13}$ $L_{23}$ $L_{33}$ |

## Iteration 1:

There is nothing to do in iteration 1 because there is only one unit clause. At iteration 1 the bound is set to 1 and only resolution over unit clauses are attempted. Since there is only one unit clause then no resolutions are performed.

## Iteration 2:

If we look back at **Table 5-5** of the minimized attribute sequences, we notice that the only sequence for iteration 2 is ⟨1,2,1⟩. The input clauses in $S$ do not contain a clause of length 2 and therefore, no resolutions are performed at iteration 2.

## Iteration 3:

Derivation 1: (*bound* = 3, *depth* = 1, *Goals* = {}, $\ddot{\sigma}$ = {})

$B_2$: P(x1,x2)

$B_3$: ¬P(x33,f(x34,x33)) ∨ ¬P(f(x34,x33),f(x34,x33)) ∨ Q(x34,x33)

$\ddot{\sigma}_1$ = {x1 → x33, x2 → f(x34,x33)}

$R_1$: ¬P(f(x34,x33),f(x34,x33)) ∨ Q(x34,x33)

$B_2$: P(x65,x66)

Variables renamed

$\ddot{\sigma}_2$ = {x65 → f(x34,x33), x66 → f(x34,x33)}

$R_2$: Q(x34,x33)

$U_1$ Unit clause constructed

Since $R_2$ is a unit clause and it is not in *Goals*, it is constructed, labeled $U_1$ and added to *Goals* giving *Goals* = {$U_1$}. $U_1$ becomes $U_1$: $u_{11}$=Q(x1,x2) after the clause is normalized.

Notice that the variables in $B_3$ and the second occurrence of $B_2$ have been renamed based on the formula given in **E6.2**. The mgu of the unification of the literals $L_{12}$ from the first occurrence of $B_2$ and $L_{13}$ from $B_3$ is $\ddot{\sigma}_1$. The overall substitution set, $\ddot{\sigma}$, is the union of all the most general unifiers. The contents of the path table are shown in **Table 6-1**. The set of modified variables in column 1 are the variables within $\ddot{\sigma}_1$ that have been bounded to terms. The value of the field, "[introduced at depth]", is 0 for $B_2$ because the instance of $B_2$ that is used in this column was introduced at depth 0 (column 0 is not explicitly entered in the table).

**Table 6-1: Path table for Derivation 1 of Example 6.3**

| Depth | 1 | 2 | ... | *MAX_DEPTH* |
|---|---|---|---|---|
| Inference rule | BR | BR | | |
| Newly introduced clause | $B_3$ | $B_2$ | | |
| *RCid* of the newly introduced clause | 32 | 64 | | |
| Literal deleted from newly introduced clause | $L_{13}$ | $L_{12}$ | | |
| Previous clause [introduced at depth] | $B_2$ [0] | $B_3$ [0] | | |
| Literal deleted or modified from previous clause | $L_{12}$ | $L_{23}$ | | |
| Modified variables | {x1, x2} | {x65, x66} | | |
| Length of resolvent/factor | 2 | 1 | | |
| Merge clause | NO | NO | | |
| Delayed | YES | YES | | |

When a resolvent is a unit clause, an attempt to find a unit conflict is made. In this case, there are no unit conflicts. Derivation 1 ends at this point because the bound is 3, the depth is 2, and, following the attribute sequences discussed in Chapter 5, the length of the clause chosen from either *Goals* or $S$ must be 1. Since there are no clauses of length 1 that can resolve with $R_2$, there is no point in proceeding with the derivation any further because the empty clause cannot be obtained at depth 3. The possible attribute sequences for iteration 3 are ⟨1,2,2,1⟩, ⟨1,3,1,1⟩, and ⟨2,2,1,1⟩. However, there are no more clauses whose sizes satisfy any of those attribute sequences and therefore, iteration 3 ends. The substitution set is reinitialized, the *depth* is set to 1, and the *bound* is incremented to 4. The attribute sequences for iteration 4 are ⟨1,2,2,2,1⟩, ⟨1,2,3,1,1⟩, ⟨1,3,1,2,1⟩, ⟨1,3,2,1,1⟩, ⟨1,4,1,1,1⟩, ⟨2,2,1,2,1⟩, ⟨2,2,2,1,1⟩, and ⟨2,3,1,1,1⟩ (see **Table 5-5**). None of the sequences can be followed except ⟨1,4,1,1,1⟩, because first, there are no clauses of length 2 and $B_3$ cannot be factored in order to reduce its length to 2, so all the

sequences that begin with the prefixes ⟨1,2⟩ and ⟨2,2⟩ are eliminated. Second, the sequences that begin with ⟨1,3,1,2,1⟩ and ⟨1,3,2,1,1⟩ can be eliminated because they require clauses of length 2 at depth 2 and depth 3 respectively, but there are no clauses of length 2 in $S$ and from Derivation 1 we realize that the only clause of length 2 that can be generated from the initial clauses $B_2$ and $B_3$ is not a merge clause, and thus, it is not added to $T$, the temporary set of constructed clauses, (see SLR in Chapter 4) and cannot be used as a far parent at deeper levels of a the derivation. The sequence ⟨1,4,1,1,1⟩ leads to the empty clause as demonstrated in Derivation 2.

Derivation 2: (*bound* = 4, *depth* = 1, *Goals* = $\{U_1\}$, $\ddot{\sigma} = \{\}$)

$B_2$: P(x1,x2)

$B_1$: ¬P(x33,f(x34,x33)) ∨ ¬P(f(x34,x33),f(x34,x33)) ∨ ¬Q(x34,f(x34,x33)) ∨ ¬Q(f(x34,x33),f(x34,x33))

$\ddot{\sigma}_1 = \{x1 \rightarrow x33,\ x2 \rightarrow f(x34,x33)\}$

$R_1$: ¬P(f(x34,x33),f(x34,x33)) ∨ ¬Q(x34,f(x34,x33)) ∨ ¬Q(f(x34,x33),f(x34,x33))

$B_2$: P(x65,x66)

$\ddot{\sigma}_2 = \{x65 \rightarrow f(x34,x33),\ x66 \rightarrow f(x34,x33)\}$

$R_2$: ¬Q(x34,f(x34,x33)) ∨ ¬Q(f(x34,x33),f(x34,x33))

$U_1$: Q(x97,x98)

$\ddot{\sigma}_3 = \{x34 \rightarrow x97,\ x98 \rightarrow f(x34,x33)\}$

$R_3$: ¬Q(f(x97,x33),f(x97,x33))

$U_1$: Q(x129,x130)

$\ddot{\sigma}_4 = \{x129 \rightarrow f(x97,x33),\ x130 \rightarrow f(x97,x33)\}$

$R_4$: $\phi$

The contents of the substitution set $\ddot{\sigma}$ at the end of the derivation is the union of all the unifiers.

$$\ddot{\sigma} = \ddot{\sigma}_1 \cup \ddot{\sigma}_2 \cup \ddot{\sigma}_3 \cup \ddot{\sigma}_4$$
$$= \{x1 \rightarrow x33,\ x2 \rightarrow f(x34,x33),\ x34 \rightarrow x97,\ x65 \rightarrow f(x34,x33),\ x66 \rightarrow f(x34,x33),$$
$$x98 \rightarrow f(x34,x33),\ x129 \rightarrow f(x97,x33),\ x130 \rightarrow f(x97,x33)\}$$

In Derivation 2, the intermediate clauses $R_1$, $R_2$, and $R_3$ are not constructed due to DCC. Even though $R_3$ is a unit clause, there is no need to construct it and add it to *Goals* since a unit conflict between $R_3$ and $U_1$ exists. $U_1$, which belongs to *Goals*, behaves just like an input clause within the derivation, i.e., it is assigned an *RCid* and its variables renamed. This is because $U_1$ has been constructed and retained in memory. The contents of the path table are shown in **Table 6-2**.

**Table 6-2: Path table for Derivation 2 of Example 6.3**

| Depth | 1 | 2 | 3 | 4 | ... | *MAX_DEPTH* |
|---|---|---|---|---|---|---|
| Inference rule | BR | BR | BR | BR | | |
| Newly introduced clause | $B_1$ | $B_2$ | $U_1$ | $U_1$ | | |
| *RCid* of the newly introduced clause | 32 | 64 | 96 | 128 | | |
| Literal deleted from newly introduced clause | $L_{11}$ | $L_{12}$ | $u_{11}$ | $u_{11}$ | | |
| Previous clause [introduced at depth] | $B_2$ [0] | $B_1$ [0] | $B_1$ [0] | $B_1$ [0] | | |
| Literal deleted or modified from previous clause | $L_{12}$ | $L_{21}$ | $L_{31}$ | $L_{41}$ | | |
| Modified variables | {x1, x2} | {x65, x66} | {x34, x98} | {x129,x130} | | |
| Length of resolvent/factor | 3 | 2 | 1 | 0 | | |
| Merge clause | NO | NO | NO | NO | | |
| Delayed | YES | YES | YES | YES | | |

We notice in Derivation 2 that each of the clauses $B_2$ and $U_1$ is used twice. In order to differentiate between the two copies. CARINE uses the *RCid* as reference

and consequently the variables in each copy of the same clause have different names based on **E6.2**.

It is clear from the above example, that the use of attribute sequences reduced the number of unfruitful attempts considerably. Indeed, we reached a refutation in just two derivations.

## 6.5 Backtracking in SLR

Backtracking requires $O(n)$ operations, where $n$ is the number of modified variables. When the parents of a resolvent (or the parent of a factor) are propositional clauses the number of modified variables is, of course, zero, i.e. $n = 0$. In general, we have found that most of the time $n$ is much less than the weight $w$ of the obtained resolvent or factor as long as the resolvent is not the empty clause. This implies that constructing the resolvent or factor takes longer than the time to set the modified variables to NULL. Therefore, even though it takes $O(n)$ to backtrack one step, it is still less than $O(w)$ (see Appendix C) which is the time to construct the resolvent/factor and delete it later on after the backtracking is performed. We tested 4681 theorems from the TPTP set to determine an overall average value for the number of modified variables per unification, $n^*_{ave}$, and an overall average value for the weight of a generated clause, $w^*_{ave}$. We found $n^*_{ave}$ to be 2 whereas, $w^*_{ave}$ is 24.

**Figure 6-11** is a graph of $n_{ave}$, the average number of modified variables over all the successful unifications in a theorem, and the corresponding $w_{ave}$, the average weight of a generated clause in each theorem for the 4681 theorems. The graph is drawn on a logarithmic scale due to the relatively high values of $w_{ave}$ with respect to $n_{ave}$. Each cross (×) indicates the average number of modified variables, $n_{ave}$, in one theorem. Similarly, each dash (-) indicates the average

length, $w_{ave}$, of a generated clause. It is readily noticeable that $w_{ave}$ is much greater than $n_{ave}$ most of the time.

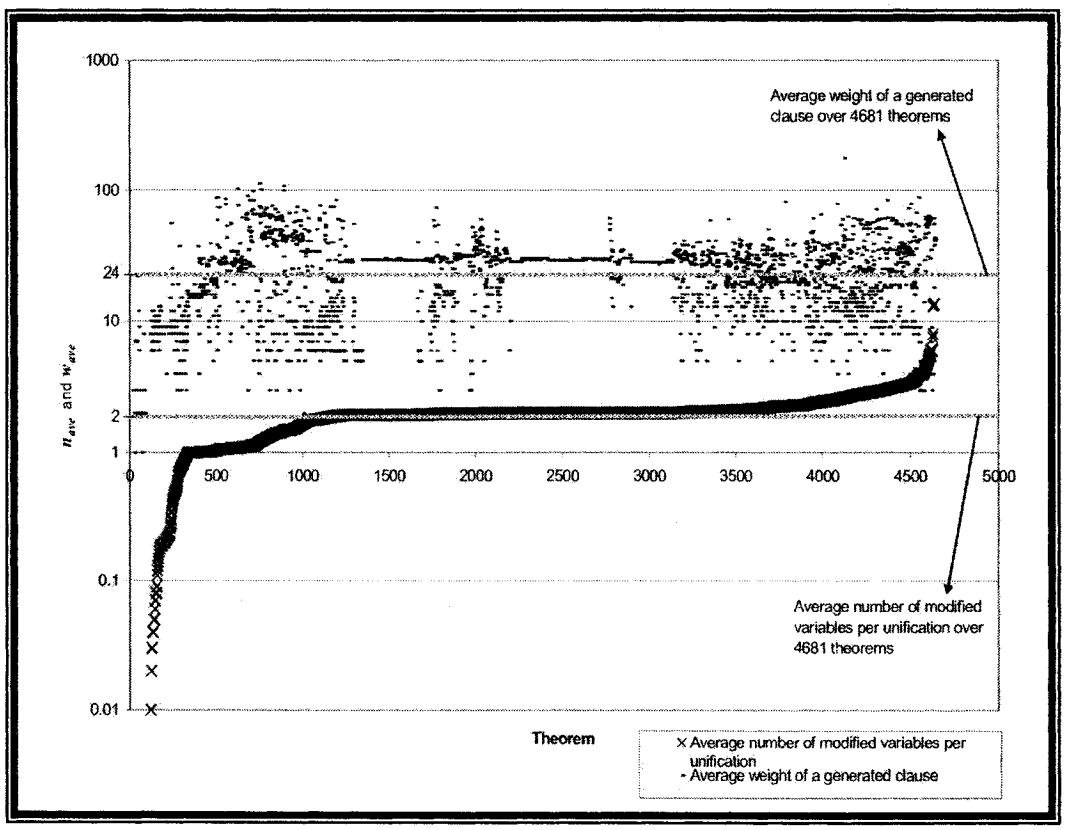



**Figure 6-11: Chart of the average number of modified variables per unification versus average length of generated clause.**

**Example 6.4:**

In this example, we show how backtracking is performed. We skip the first three iterations and start with iteration 4. The following derivations, although numbered as 1 and 2, are not necessarily the first two derivations that CARINE caries out at iteration 4. We chose such derivations to depict an instance of backtracking. Suppose $S = \{B_1, B_2, B_3, B_4\}$ is a set of input clauses, where $B_1$ is the negated conclusion, and

| |
|---|
| $B_1$: ¬P(a,a)<br>$L_{11}$ |
| $B_2$: ¬Q(b) ∨ P(b,a)<br>$L_{21}$ $L_{22}$ |
| $B_3$: ¬Q(x1) ∨ ¬P(x2,x1)<br>$L_{13}$ $L_{23}$ |
| $B_4$: Q(x1) ∨ P(x1,a)<br>$L_{14}$ $L_{24}$ |

## Iteration 4:

Derivation 1a: (*bound* = 4, *depth* = 1, *Goals* = {}, $\ddot{\sigma}$ = {})

$B_1$: ¬P(a,a)

$B_4$: Q(x33) ∨ P(x33,a)

$\ddot{\sigma}_1$ = {x33→a}

$R_1$: Q(a)

$B_3$: ¬Q(x65) ∨ ¬P(x66,x65)

$\ddot{\sigma}_2$ = {x65→a}

$R_2$: ¬P(x66,a)

$B_4$: Q(x97) ∨ P(x97,a)

$\ddot{\sigma}_3$ = {x66→x97}

$R_3$: Q(x97) ← $U_1$

$\ddot{\sigma} = \ddot{\sigma}_1 \cup \ddot{\sigma}_2 \cup \ddot{\sigma}_3$

= {x33→a, x65→a, x66→x97}

Even though $B_1$ has no variables, still, an *RCid* is assigned to it. Derivation 1a terminates at depth 3 because at this point, there are no unit clauses that resolve with $R_3$. Since $R_3$ is a unit clause that does not exist in *Goals*, it is constructed,

labeled $U_1$, and added to *Goals*. $R_1$ and $R_2$ are also unit clauses but they are not added to *Goals* in this derivation because it is assumed that they have been added within previous iterations. We do not show them as part of the *Goals* for the sake of demonstrating the backtracking process. *Goals* becomes *Goals* = $\{U_1\}$, where $U_1 : u_{11} = \texttt{Q(x1)}$ after $U_1$ is constructed and normalized. The contents of the path table for Derivation 1a is shown in **Table 6-3**.

**Table 6-3: Path table for Derivation 1a of Example 6.4**

| Depth | 1 | 2 | 3 | ... | *MAX_DEPTH* |
|---|---|---|---|---|---|
| Inference rule | BR | BR | BR | | |
| Newly introduced clause | $B_4$ | $B_3$ | $B_4$ | | |
| *RCid* of the newly introduced clause | 32 | 64 | 96 | | |
| Literal deleted from newly introduced clause | $L_{24}$ | $L_{13}$ | $L_{24}$ | | |
| Previous clause [introduced at depth] | $B_1$ [0] | $B_4$ [0] | $B_3$ [1] | | |
| Literal deleted or modified from previous clause | $L_{11}$ | $L_{14}$ | $L_{23}$ | | |
| Modified variables | {x33} | {x65} | {x66} | | |
| Length of resolvent/factor | 1 | 1 | 1 | | |
| Merge clause | NO | NO | NO | | |
| Delayed | YES | YES | YES | | |

Once depth 3 is reached and no further resolutions with unit clause are possible, CARINE backtracks to depth 2 and tries to resolve $R_2$ with $B_2$. The two step backtracking process is quite simple when using the path table:

(1) The variables that have been bound at depth 3 are freed. In our case, x66 is removed from $\ddot{\sigma}$, i.e., the "*Pointer to term*" field (see **Figure 6-5**) of location 66 in the substitution set array is set to NULL.

(2) The pointer to column 3 is moved back to the previous column.

The derivation, after backtracking is performed, is listed below as Derivation 1b. The *bound* is still 4 but the *depth* is decremented to 2. The set *Goals* maintains the additional unit clause $U_1$ generated at depth 3 even though backtracking has been carried out. The substitution set $\ddot{\sigma}$ contains all the variables that have been bounded to terms up to and including depth 2, i.e., $\ddot{\sigma} = \ddot{\sigma}_1 \cup \ddot{\sigma}_2$.

Derivation 1b: (*bound* = 4, *depth* = 2, *Goals* = $\{U_1\}$, $\ddot{\sigma} = \{x33 \rightarrow a, x65 \rightarrow a\}$)

$B_1$: ¬P(a,a)

$B_4$: Q(x33) ∨ P(x33,a)

$\ddot{\sigma}_1 = \{x33 \rightarrow a\}$

$R_1$: Q(a)

$B_3$: ¬Q(x65) ∨ ¬P(x66,x65)

$\ddot{\sigma}_2 = \{x65 \rightarrow a\}$

Backtrack

$R_2$: ¬P(x66,a)

$B_4$: Q(x97) ∨ P(x97,a)

$\ddot{\sigma}_3 = \{x66 \rightarrow x97\}$

$R_3$: Q(x97)

$B_4$ is removed from the path table, the modified variables in $\ddot{\sigma}_3$ are freed and the depth is decremented back to 2.

The contents of the path table after the backtracking is performed are shown below in **Table 6-4**. After the backtracking is completed, the resolution proceeds with $B_2$ as the input clause. Derivation 1c lists the steps up to depth 4. In Derivation 1c the empty clause is obtained by resolution between $R_3$ and the unit clause $U_1$.

**Table 6-4: Path table for Derivation 1b of Example 6.4**

| Depth | 1 | 2 | 3 | ... | *MAX_DEPTH* |
|---|---|---|---|---|---|
| Inference rule | BR | BR | | | |
| Newly introduced clause | $B_4$ | $B_3$ | | | |
| *RCid* of the newly introduced clause | 32 | 64 | | | |
| Literal deleted from newly introduced clause | $L_{24}$ | $L_{13}$ | | | |
| Previous clause [introduced at depth] | $B_1$ [0] | $B_4$ [0] | | | |
| Literal deleted or modified from previous clause | $L_{11}$ | $L_{14}$ | | | |
| Modified variables | {x33} | {x65} | | | |
| Length of resolvent/factor | 1 | 1 | | | |
| Merge clause | NO | NO | | | |
| Delayed | YES | YES | | | |

Derivation 1c: ($bound = 4$, $depth = 2$, $Goals = \{U_1\}$, $\ddot{\sigma} = \{x33 \rightarrow a, x65 \rightarrow a\}$)

$B_1$: ¬P(a,a)

$B_4$: Q(x33) ∨ P(x33,a)

$\ddot{\sigma}_1 = \{x33 \rightarrow a\}$

$R_1$: Q(a)

$B_3$: ¬Q(x65) ∨ ¬P(x66,x65)

$\ddot{\sigma}_2 = \{x65 \rightarrow a\}$

$R_2$: ¬P(x66,a)

$B_2$: ¬Q(b) ∨ P(b,a)

$\ddot{\sigma}_3 = \{x66 \rightarrow b\}$

$R_3$: ¬Q(b)

$U_1$: Q(x129)

$\ddot{\sigma}_4 = \{x129 \rightarrow b\}$

$R_4$: $\phi$

$\ddot{\sigma} = \ddot{\sigma}_1 \cup \ddot{\sigma}_2 \cup \ddot{\sigma}_3 \cup \ddot{\sigma}_4$

$= \{x33 \rightarrow a, x65 \rightarrow a, x66 \rightarrow b, x129 \rightarrow b\}$

Derivation 1c terminates at depth 4 when the empty clause is obtained. The contents of the path table for Derivation 1c are shown in **Table 6-5**.

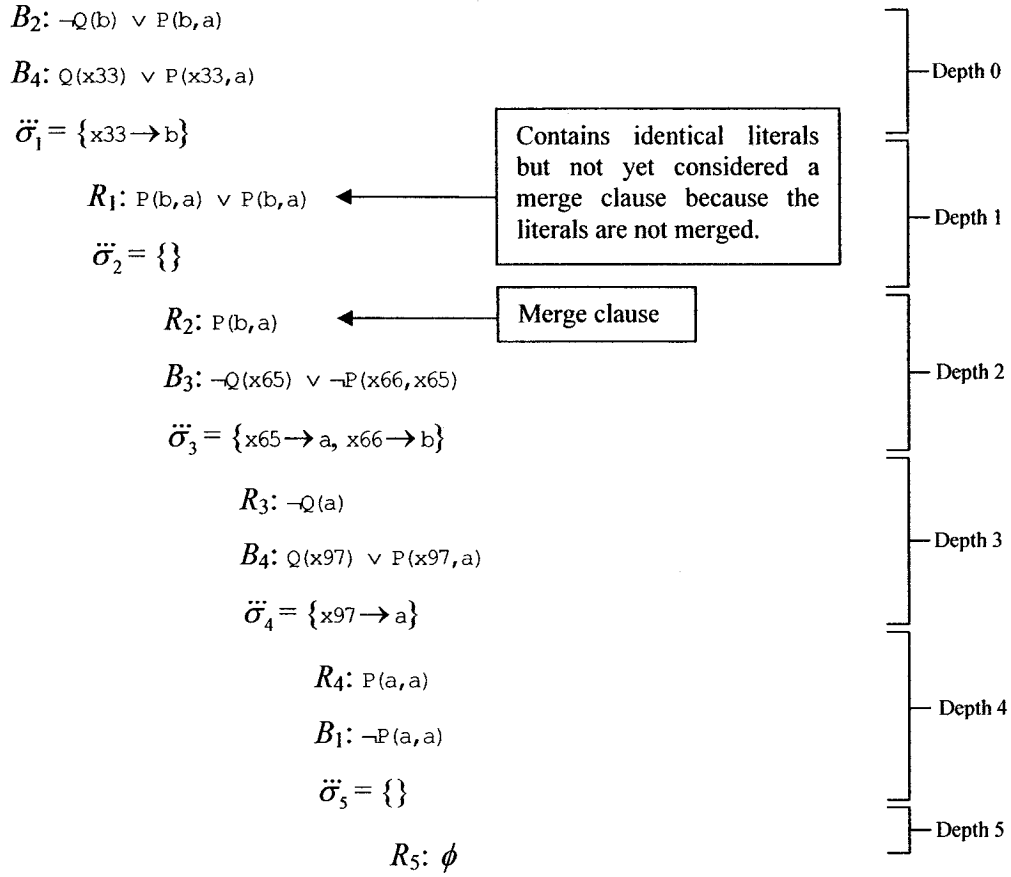
**Table 6-5: Path table for Derivation 1c of Example 6.4**

| Depth | 1 | 2 | 3 | 4 | ... | *MAX_DEPTH* |
|---|---|---|---|---|---|---|
| Inference rule | BR | BR | BR | BR | | |
| Newly introduced clause | $B_4$ | $B_3$ | $B_2$ | $U_1$ | | |
| *RCid* of the newly introduced clause | 32 | 64 | 96 | 128 | | |
| Literal deleted from newly introduced clause | $L_{24}$ | $L_{13}$ | $L_{22}$ | $u_{11}$ | | |
| Previous clause [introduced at depth] | $B_1$ [0] | $B_4$ [0] | $B_3$ [1] | B2 [2] | | |
| Literal deleted or modified from previous clause | $L_{11}$ | $L_{14}$ | $L_{23}$ | $L_{12}$ | | |
| Modified variables | {x33} | {x65} | {x66} | {x129} | | |
| Length of resolvent/factor | 1 | 1 | 1 | 0 | | |
| Merge clause | NO | NO | NO | NO | | |
| Delayed | YES | YES | YES | NO | | |

**Example 6.5**

This example demonstrates how binary factoring is managed by the path table and how the temporary set $T$ of constructed clauses is used (we do no show the first 4 iterations, we jump directly to iteration 5). The following derivation is a continuation of the previous example so the input clauses are the same. We assume that the set *Goals* is empty. We show the contents of the temporary set $T$ of constructed clauses. At the beginning of each derivation the set $T$ is empty (see SLR in Chapter 4).

Derivation 2: (*bound* = 5, *depth* = 1, *Goals* = {}, $\ddot{\sigma}$ = {}, *T*={})




$$\ddot{\sigma} = \ddot{\sigma}_1 \cup \ddot{\sigma}_2 \cup \ddot{\sigma}_3 \cup \ddot{\sigma}_4 \cup \ddot{\sigma}_5$$

$$= \{x33 \rightarrow b, x65 \rightarrow a, x66 \rightarrow b, x97 \rightarrow a\}$$

In Derivation 2, the merge clause $R_2$ is constructed, labeled $C_1$, and added to the set *T*. *T* becomes $T=\{C_1\}$ where $C_1$:P(b,a). The empty clause is derived at *depth* = *bound* and hence, there was no need to use the merge clause. Notice that in this case the merge clause was not needed to obtain a refutation. The contents of the path table for Derivation 2 are shown in **Table 6-6**.

**Table 6-6: Path table for Derivation 2 of Example 6.4**

| Depth | 1 | 2 | 3 | 4 | 5 | ... | *MAX_DEPTH* |
|---|---|---|---|---|---|---|---|
| Inference rule | BR | BF | BR | BR | BR | | |
| Newly introduced clause | $B_4$ | - | $B_3$ | $B_4$ | $B_1$ | | |
| *RCid* of the newly introduced clause | 32 | - | 64 | 96 | 128 | | |
| Literal deleted from newly introduced clause | $L_{14}$ | - | $L_{23}$ | $L_{14}$ | $L_{11}$ | | |
| Previous clause [introduced at depth] | $B_2$ [0] | $B_2$ [0] | $B_4$ [0] | $B_3$ [2] | $B_4$ [3] | | |
| Literal deleted or modified from previous clause | $L_{12}$ | $L_{22}$ | $L_{24}$ | $L_{13}$ | $L_{24}$ | | |
| Modified variables | {x33} | {} | {x65, x66} | {x97} | {} | | |
| Length of resolvent/factor | 2 | 1 | 1 | 1 | 0 | | |
| Merge clause | NO | YES | NO | NO | NO | | |
| Delayed | YES | NO | YES | YES | NO | | |

To construct the merge clause using the above path table, we list the literals of the input clauses $B_2$ and $B_4$, delete all the literals mentioned in the row "*Literal deleted or modified from previous clause*" up to column 2, and finally apply the substitution sets from columns 1 and 2 over the remaining non-deleted literals. Since only one substitution set, namely $\ddot{\sigma}$, is maintained, it may appear difficult to extract the substitution terms of only the variables that are involved in the first two columns when $\ddot{\sigma}$ contains all the bound variables up to depth 5. However, this is quite simple because we rely on the information in the modified variables row. Formally, the problem can be stated as follows. Given $\ddot{\sigma} = \bigcup_{i=1}^{depth} \ddot{\sigma}_i$, we want to extract $\ddot{\sigma}_{1..k} = \bigcup_{i=1}^{k} \ddot{\sigma}_i$ where $k \leq depth$ from $\ddot{\sigma}$, and then apply $\ddot{\sigma}_{1..k}$ to the delayed clause at depth $k$. In our case, $k = 2$ and $depth = 5$. We want to rebuild $\ddot{\sigma}_1 \cup \ddot{\sigma}_2$ from $\ddot{\sigma}$. First, we list all the variables in the "*Modified variables*" row

up to column $k$. Next, we obtain all the substitution terms for those variables from $\ddot{\sigma}$. Since we store $\ddot{\sigma}$ as a non-idempotent substitution set, this makes the extraction of $\ddot{\sigma}_{1..k}$ using the above two simple steps easy and the result is correct. In our example, the modified variables up to column 2 are {x33}. The substitution term for x33 is b. We can now construct the merge clause. First, we list all the literals of $B_2$ and $B_4$

| ¬Q(b) ∨ P(b,a) ∨ Q(x33) ∨ P(x33,a) | | | |
|---|---|---|---|
| $L_{12}$ | $L_{22}$ | $L_{14}$ | $L_{24}$ |

Next, we delete all the literals $L_{14}$, $L_{12}$, and $L_{22}$ as entered in the path table in the rows of "*Literal deleted from newly introduced clause*" and "*Literal deleted or modified from previous clause*" up to column 2. We are left with P(x33,a). Finally, we apply the substitution set $\ddot{\sigma}_{1..2} = \ddot{\sigma}_1 \cup \ddot{\sigma}_2$ to the remaining literals, and we get $[\text{P(x33,a)}]\ddot{\sigma}_{1..2} = \text{P(b,a)}$ which is $R_2$ in Derivation 2.

## 6.6 Experimental Results

In this section, we present some of the experimental results gathered from running CARINE over a selected set of theorems from the TPTP library version 2.6.0. We selected a sample of 100 theorems from all the domains in the TPTP library whose characteristics (number of input clauses, rating, maximum term depth, length of clauses, weight of clauses, etc.) cover most of the characteristics found in the rest of the theorems in the TPTP library (see Appendix D for the list of the 100 theorems). We set the time limit in CARINE to 180 seconds so that CARINE had 180 seconds to prove each theorem. None of the theorems was proved by CARINE, so CARINE used up all of the given time for each theorem.

We begin by presenting and discussing the results of the percentage of time spent in constructing clauses. In order to acquire comparable results, we built a version of CARINE that constructs and discards every generated clause (i.e., no

DCC is employed). We use the symbol 𝔸 in the following sections to refer to this version of CARINE and the symbol 𝔹 to refer to the version of CARINE where DCC is employed.

The total time, $Ct(t)$, spent in the construction process in each theorem, $t$, is divided by the total time, $Gt(t)$, given for 𝔸 to prove the theorem $t$, and then multiplied by 100 to obtain the percent of time spent in constructing clauses; $PTCC(t)$. The formula of the percentage of time spent constructing clauses is

$$PTCC(t) = \frac{Ct(t)}{Gt(t)} \times 100 \qquad \textbf{(E6.3)}$$

The chart in **Figure 6-12** shows the $PTCC(t)$ for each of the 100 theorems.

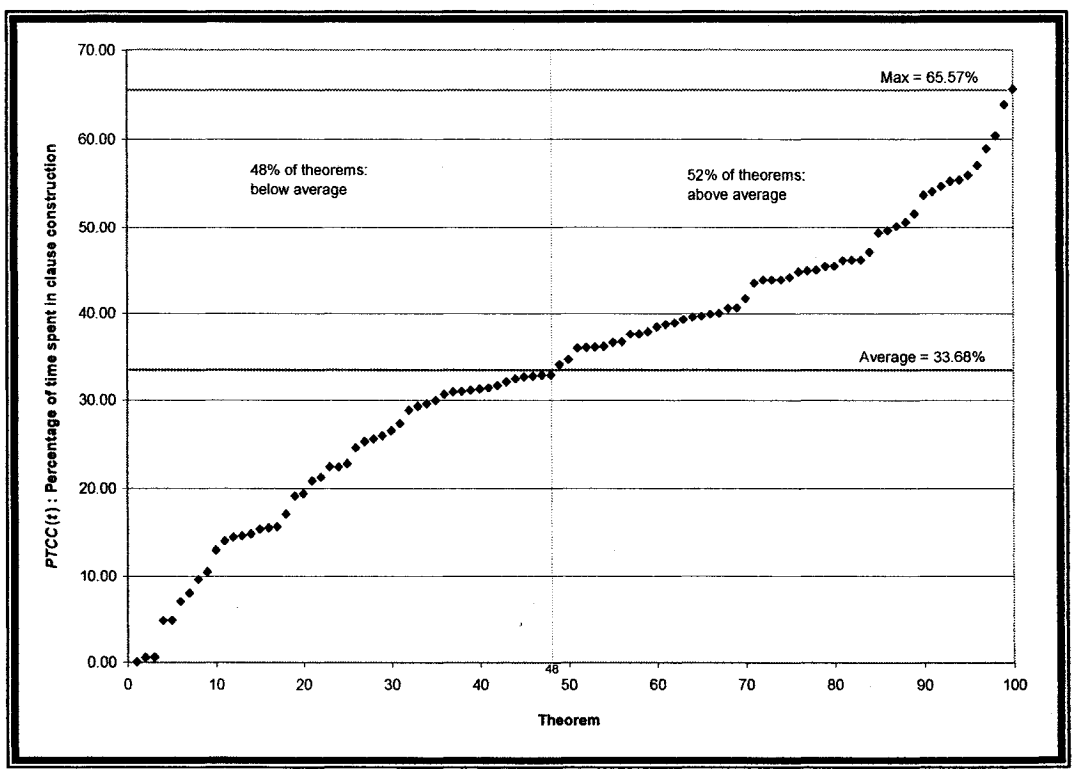



**Figure 6-12: Chart of the percentage of time spent constructing clauses.**

The time spent constructing clauses affects the inference rate. The inference rate usually degrades as the percentage of time spent constructing clauses goes up. The inference rate is the number of generated clauses *NGen(t)* over a period of time *Rt* divided by *Rt*, where *Rt* is the running time of the ATP over a given theorem *t*. The running time is either the time it takes the ATP to find a proof, or the time that the ATP runs for before it gives up without finding a proof. Since we are using the same platform for all the theorems and the running time is *Rt* =180 seconds (since CARINE did not prove any of the 100 theorems), we calculate the inference rate by the formula

$$IR(t) = \frac{NGen(t)}{180} \qquad \textbf{(E6.4)}$$

The chart in **Figure 6-13** shows the inference rates, calculated by **E6.4**, where *NGen(t)* was acquired through the running of 𝔸 and 𝔹 over the selected 100 theorems. The theorems in the chart in **Figure 6-13** are sorted by the inference rate obtained from 𝔸. This makes it easier to notice that the inference rate of the version where the clauses were not constructed is generally substantially higher than the inference rate of the version where the clauses were constructed.

The increase in inference rate or **inference rate speedup** (*IRS*) is calculated as

$$IRS(t) = \frac{IR_{\mathbb{B}}(t)}{IR_{\mathbb{A}}(t)} \qquad \textbf{(E6.5)}$$

From **Figure 6-14** we notice that the inference rate speedup increases as the time spent constructing clauses increases. This implies that the inference rate speedup is not constant; it depends on the overall time spent in clause construction. The more time 𝔸 spends in clause construction, the slower it is with respect to 𝔹 and hence, the more is the inference rate speedup.

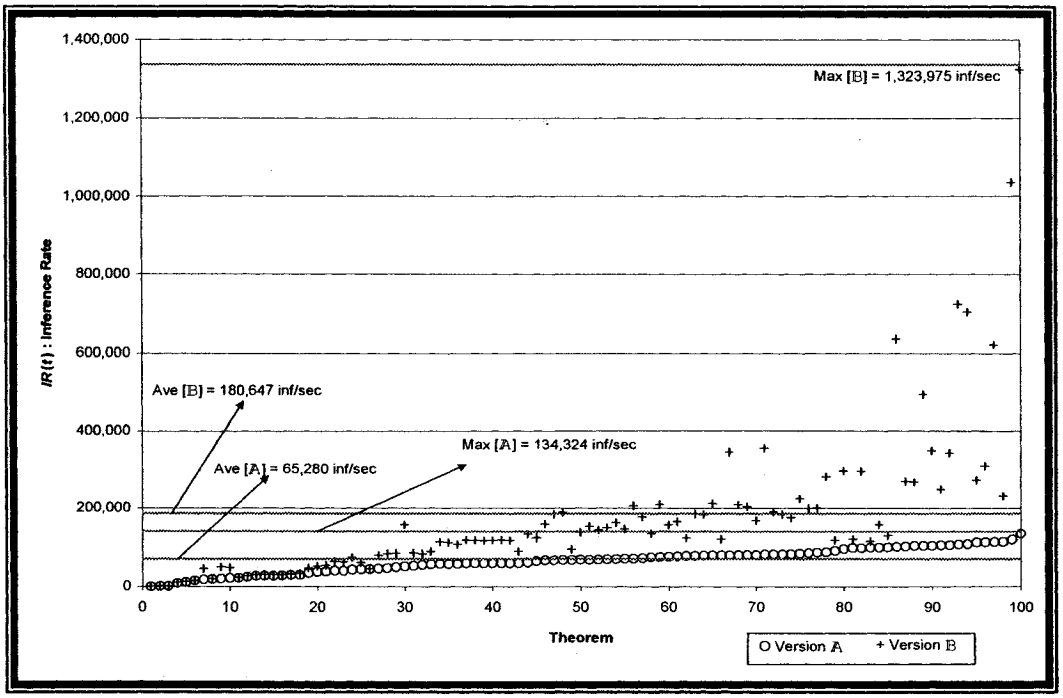



**Figure 6-13: Chart comparing the inference rates of CARINE with and without DCC.**

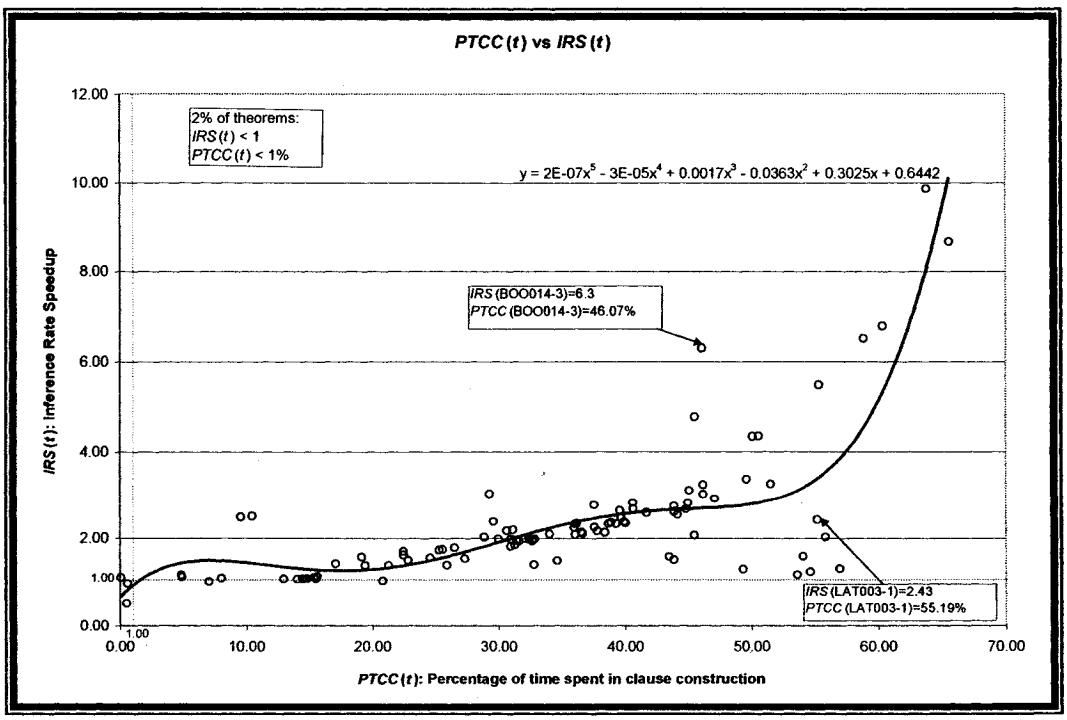



**Figure 6-14: Chart of the relation between *PTCC(t)* and *IRS(t)*.**

We conclude that, in general, the higher the value of $PTCC(t)$, the higher is $IRS(t)$. Even though this statement is true most of the time, it is not true all the time as can be seen from the chart of **Figure 6-14**. For instance, if we inspect the results over the theorems LAT003-1 and BOO014-3 that are listed in **Table 6-7**, we notice that the percentage of time spent in clause construction when A was attempting to find a proof for theorem LAT003-1 is 9.12% higher than that of BOO014-3, however, the inference rate speedup is 6.3/2.43=2.59 times lower. We can obviously spot other instances from **Figure 6-14** where this case occurs. We also notice that 2% of the theorems resulted in an inference rate speedup that is less than 1 and both theorems had a $PTCC(t)$ that is less than 1%. We discuss why the inference rate speedup from those two theorems was less than 1 later when we define the unit conflict tests.

**Table 6-7: Comparison of *PTCC* and *IR* of some theorems**

| Theorem | *PTCC* [%] | $IR_B$ [inf/sec] | $IR_A$ [inf/sec] | *IRS* |
|---|---|---|---|---|
| BOO014-3 | 46.07 | 635,875 | 100,854 | 6.30 |
| LAT003-1 | 55.19 | 271,284 | 111,651 | 2.43 |

The curve-fit[1] drawn in **Figure 6-14** has a margin of error of 23.01% on average. Nevertheless, it can be used to obtain a rough estimate of the inference rate speedup once the percentage of time spent in clause construction is determined. This might be helpful for example, in finding out ahead of time if implementing or turning on the option of using delayed clause construction is going to result in the desired inference rate speedup over a range of theorems. For instance, if the clauses are all propositional clauses or the maximal term depth is 1 (i.e., clauses contain no functions but only variables and constants), then DCC might not provide a significant speedup in the inference rate.

[1] We experimented with several dozens of different curve-fits using polynomial (of different degrees), logarithmic and trigonometric functions and we found that the curve-fit presented here provided the least margin of error on average and the expression is relatively simple with respect to the other tested curve-fits.

The reason that a higher *PTCC* does not always translate into a higher *IRS* when clauses are no longer constructed is that it is not the only factor that affects the inference rate speedup. Another factor that affects the inference rate speedup is the difference in the percentage of successful unifications between version $\mathbb{A}$ and version $\mathbb{B}$. In CARINE, a successful unification due to binary resolution or binary factoring leads to a newly generated clause. **Figure 6-15** shows a chart of the percentage of successful unifications obtained from running $\mathbb{A}$ and $\mathbb{B}$ over the selected 100 theorems. Therefore, the number of successful unifications is the same as the number of derived clauses. The percentage of successful unifications, $PSU_{ATP}(t)$, obtained from running an ATP over a theorem $t$ is calculated by the formula

$$PSU_{ATP}(t) = \frac{SU_{ATP}(t)}{TU_{ATP}(t)} \times 100, \qquad \textbf{(E6.6)}$$

where $SU_{ATP}(t)$ is the number of successful unifications that occurred over the running time period and $TU_{ATP}(t)$ is the total number of attempted unifications over the running time period. From the chart in **Figure 6-15** we can see that the percentage of successful unifications obtained from $\mathbb{B}$ can be higher or lower than the percentage of successful unifications obtained from $\mathbb{A}$. Indeed, 37% of the theorems resulted in a higher percentage of successful unifications under $\mathbb{B}$, 8% of the theorems produced around the same percentage of successful unifications as $\mathbb{A}$, and 55% had a lower percentage than that with $\mathbb{A}$. The ratio of the percentage of successful unifications obtained from running the two versions of CARINE over a theorem $t$ is

$$RPSU(t) = \frac{PSU_{\mathbb{B}}(t)}{PSU_{\mathbb{A}}(t)} \times 100, \qquad \textbf{(E6.7)}$$

By plotting *RPSU*($t$) versus *IRS*($t$), as shown in **Figure 6-16**, we can investigate the effect of the percentage of successful unifications on the inference rate speedup.

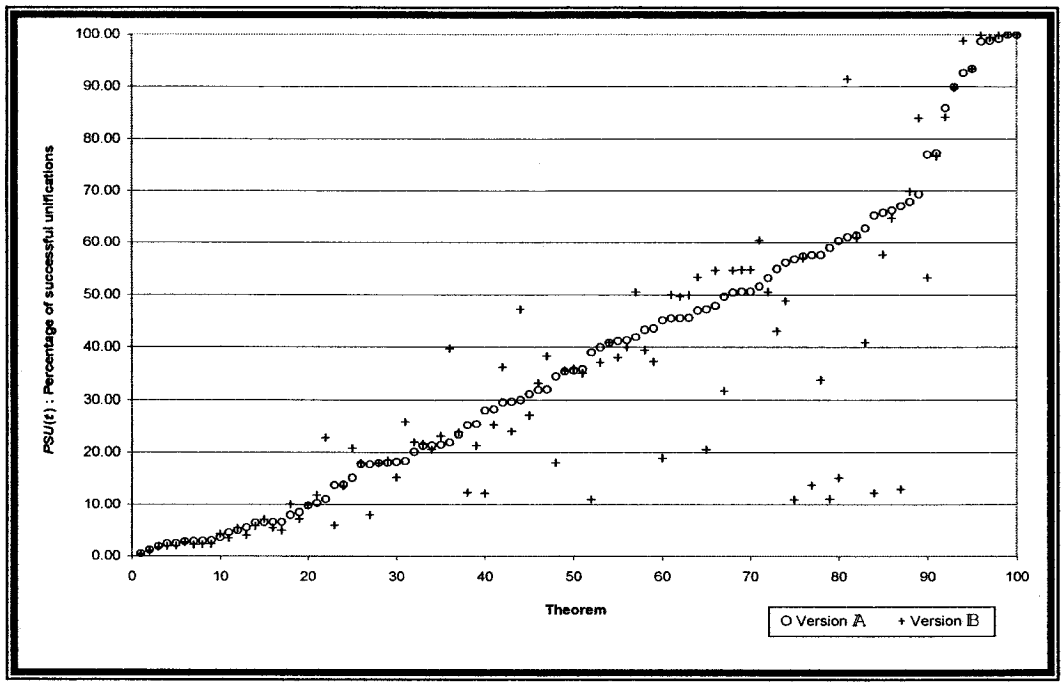



**Figure 6-15: Chart of the percentage of successful unifications obtained from running 𝔸 and 𝔹 over 100 selected theorems.**

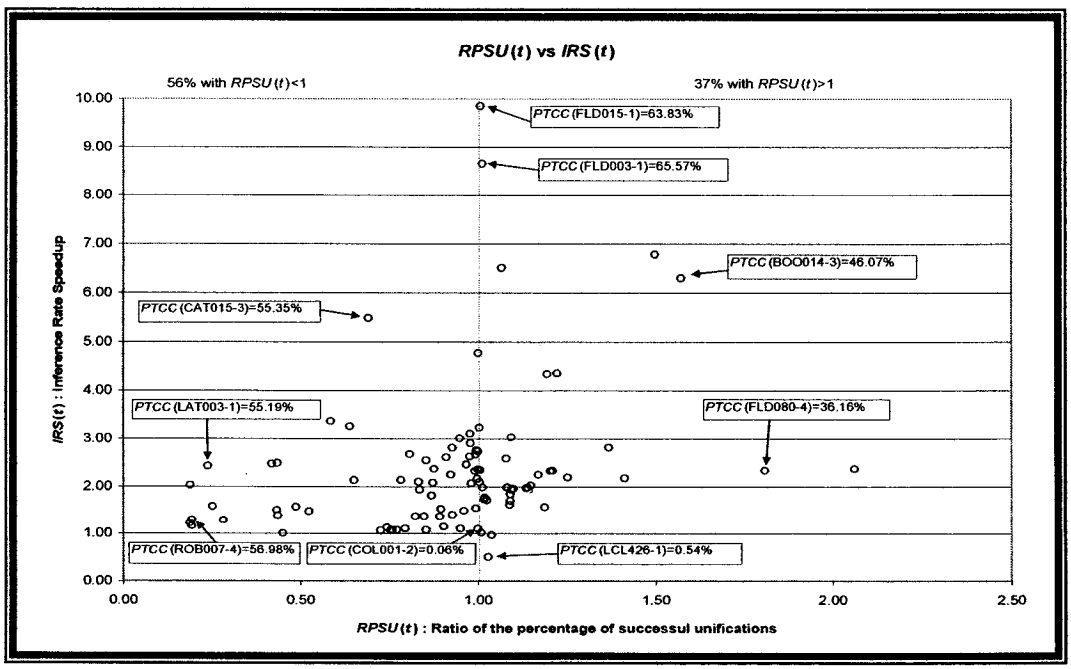



**Figure 6-16: Chart of *RPSU*(*t*) vs. *IRS*(*t*).**

We added the *PTCC* values in **Figure 6-16** for some theorems to give an idea of the relationship between the two main factors, *RPSU*($t$) and *PTCC*($t$), that affect the inference rate speedup. If the ratio of the percentages of successful unifications is less than 1, then $\mathbb{B}$ performed a lower percentage of successful unifications than $\mathbb{A}$. However, if $RPSU(t) < 1$, it does not necessarily mean that the *IRS*($t$) is low. In fact, even though *RPSU*(CAT015-3)=0.69, the value for *IRS*(CAT015-3)=5.49 which is relatively high by comparison with other *IRS* values. This is due to the high value of *PTCC*(CAT015-3)=55.35%. The reason why $RPSU(t) < 1$ is usually due to the fact that it is highly likely that when theorems contain constants, the number of successful resolutions in a derivation decreases as the derivation grows deeper because of the binding of the variables to constants and function.

**Table 6-8** lists some theorems and the corresponding values of *RPSU*($t$), *PTCC*($t$), and *IRS*($t$).

**Table 6-8: *RPSU*($t$), *PTCC*($t$) and *IRS*($t$) of some theorems**

| Theorem | *RPSU*($t$) [%] | *PTCC*($t$) [%] | *IRS*($t$) |
|---|---|---|---|
| BOO014-3 | 1.57 | 46.07 | 6.30 |
| CAT015-3 | 0.69 | 55.35 | 5.49 |
| COL001-2 | 1.00 | 0.06 | 1.10 |
| FLD003-1 | 1.01 | 65.57 | 8.66 |
| FLD015-1 | 1.01 | 63.83 | 9.86 |
| FLD080-4 | 1.81 | 36.16 | 2.34 |
| LAT003-1 | 0.24 | 55.19 | 2.43 |
| LCL426-1 | 1.03 | 0.54 | 0.50 |
| ROB007-4 | 0.19 | 56.98 | 1.30 |

An interesting observation from **Table 6-8** is the value of *IRS*(LCL426-1)=0.5 which is less than 1 indicating that 𝔹 actually performed worse than 𝔸 in this case. The main reason for the worse performance is attributed to two factors:

1. the low value of *PTCC*(LCL426-1)=0.54% which occurred at the same time when the percentage of successful unifications is almost unchanged, i.e., *RPSU*(LCL426-1)=1.03 (which is close to 1), and
2. the number of times the test for unit conflict is performed.

Even though a unit conflict test is an attempt to unify possible potentially complementary literals from two unit clauses, it is not counted in the number of attempted unifications. This is because the number of unit conflict can change dramatically with respect to the number of attempted unifications between clauses where at least one of them is not a unit clause. For example, the number of unit conflict tests can sometimes be a million times less (e.g., LDA011-1) than the number of attempted unifications where at least one of them is not a unit clause while in others it can be 75000 time more (e.g., COL001-2). Due to this dramatic change, we decided to study the effect of unit conflict test separately.

We denote the **number of unit conflicts tests** performed by an ATP over a theorem $t$ during a runtime $Rt$ by $UCT_{ATP}(t)$. For LCL426-1, 𝔹 performed 134,754,651 attempts to find a unit conflict (i.e, $UCT_{\mathbb{B}}(\text{LCL426-1}) = 134,745,651$), and a mere 151,424 successful unifications from a total of 216,956 attempted unifications. By comparison, for the same theorem, 𝔸 performed 43,338,280 attempts on finding a unit conflict (i.e., $UCT_{\mathbb{A}}(\text{LCL426-1}) = 43,338,280$), 130,443 successful unification and 192,206 attempted unifications. The ratio

$$\frac{UCT_{\mathbb{B}}(\text{LCL426-1})}{UCT_{\mathbb{A}}(\text{LCL426-1})} = \frac{134,754,651}{43,338,280} \sim 3.11$$

indicates that the number of unit conflict attempts tripled, whereas $RPSU(\text{LCL426-1}) = 1.03$ (see **Table 6-8**) remained practically the same. To maintain a relatively high inference rate speedup, the ratio of the number of unit conflict tests of $\mathbb{B}$ to $\mathbb{A}$ must be (ideally) less or equal to 1, if the percentage of successful unifications remain almost the same, i.e. $RPSU(t)$ is close to 1. The ratio of the number of unit conflict tests performed while searching for a proof of a theorem $t$ between the two versions of CARINE is expressed as

$$RUCT(t) = \frac{UCT_{\mathbb{B}}(t)}{UCT_{\mathbb{A}}(t)}, \qquad \textbf{(E6.8)}$$

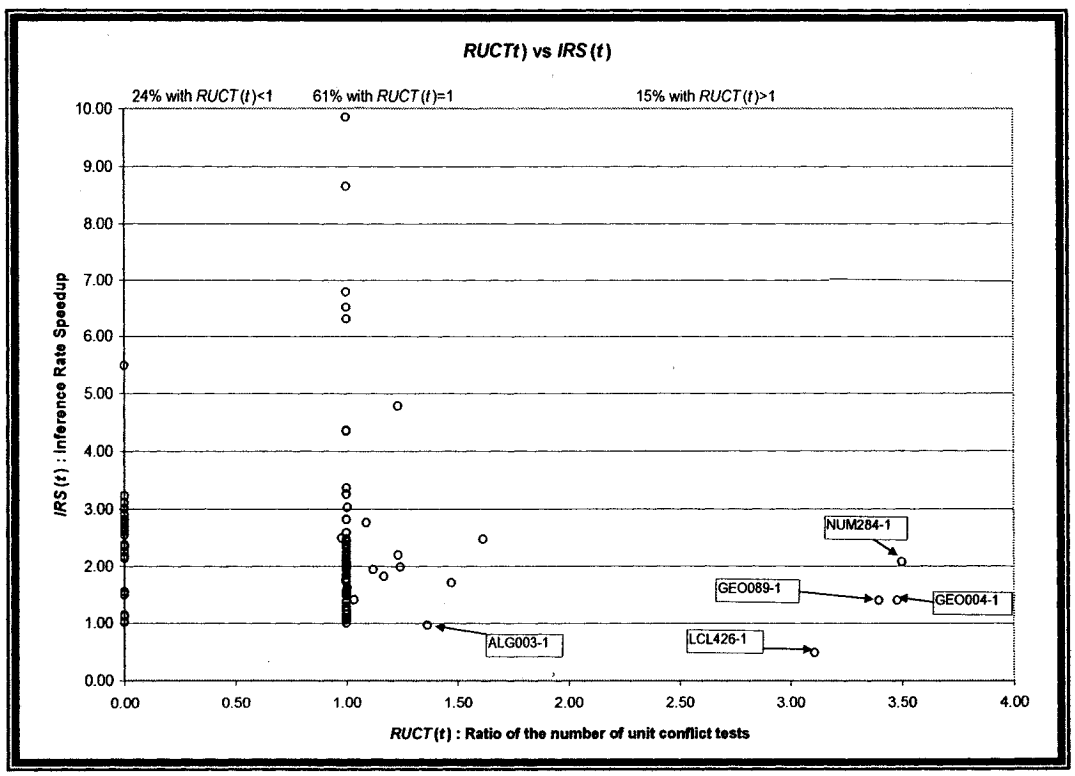



**Figure 6-17: Chart of *RUCT(t)* vs *IRS(t)*.**

**Figure 6-17** shows *RUCT*($t$) versus *IRS*($t$) for the selected 100 theorems. We notice that $RUCT(t) > 1$ for 15% of the theorems and $RUCT(t) > 3$ for only 4% of the theorems. This is an indication that, generally, the number of unit conflict tests does not increase substantially between $\mathbb{B}$ and $\mathbb{A}$. Even though the *RUCT*($t$) might occasionally increase substantially, this does not mean that the inference rate speedup is going to be less than 1 as can be seen in **Table 6-9** with the three theorems GEO004-1, GEO089-1, and NUM284-1.014. For instance, *RUCT*(NUM284-1.014)=3.5 is higher than *RUCT*(LCL426-1)=3.11 and even though the difference in the percentage of successful unifications between both theorems is very small, i.e., *RPSU*(LCL426-1)-*RPSU*(NUM284-1.014)=1.03-0.98=0.05, the inference rate of $\mathbb{B}$ is double that of $\mathbb{A}$ over NUM284-1.014, whereas the inference rate of $\mathbb{B}$ is half that of $\mathbb{A}$ over LCL426-1.

**Table 6-9: *RUCT*($t$), *PTCC*($t$), *RPSU*($t$), *IRS*($t$) of some theorems.**

| Theorem | *RUCT*($t$) | *PTCC*($t$) [%] | *RPSU*($t$) | *IRS*($t$) |
|---|---|---|---|---|
| ALG003-1 | 1.36 | 0.59 | 1.04 | 0.97 |
| GEO004-1 | 3.48 | 25.92 | 0.82 | 1.38 |
| GEO089-1 | 3.40 | 32.82 | 0.44 | 1.40 |
| LCL426-1 | 3.11 | 0.54 | 1.03 | 0.50 |
| NUM284-1.014 | 3.50 | 45.46 | 0.98 | 2.07 |

By consulting **Table 6-9**, we conclude that the reason why the inference rate speedup for NUM284-1.014 still went up as opposed to the inference rate speedup for LCL426-1 which went down is the much higher *PTCC*($t$) of NUM284-1.014 since both have an *RPSU*($t$) that is close to 1 and their *RUCT*($t$) values are close.

The question that arises at this point, after investigating the three factors *PTCC*($t$), *RPSU*($t$) and *RUCT*($t$) that affect the inference rate, is *which one of those factors is most tightly related to the inference rate speedup?* By inspecting the chart of all three factors along with the *IRS*($t$) over the selected 100 theorems in **Figure 6-18**,

we can conclude that *PTCC*(*t*) follows the trend of *IRS*(*t*) more closely than either *RPSU*(*t*) and *RUCT*(*t*).

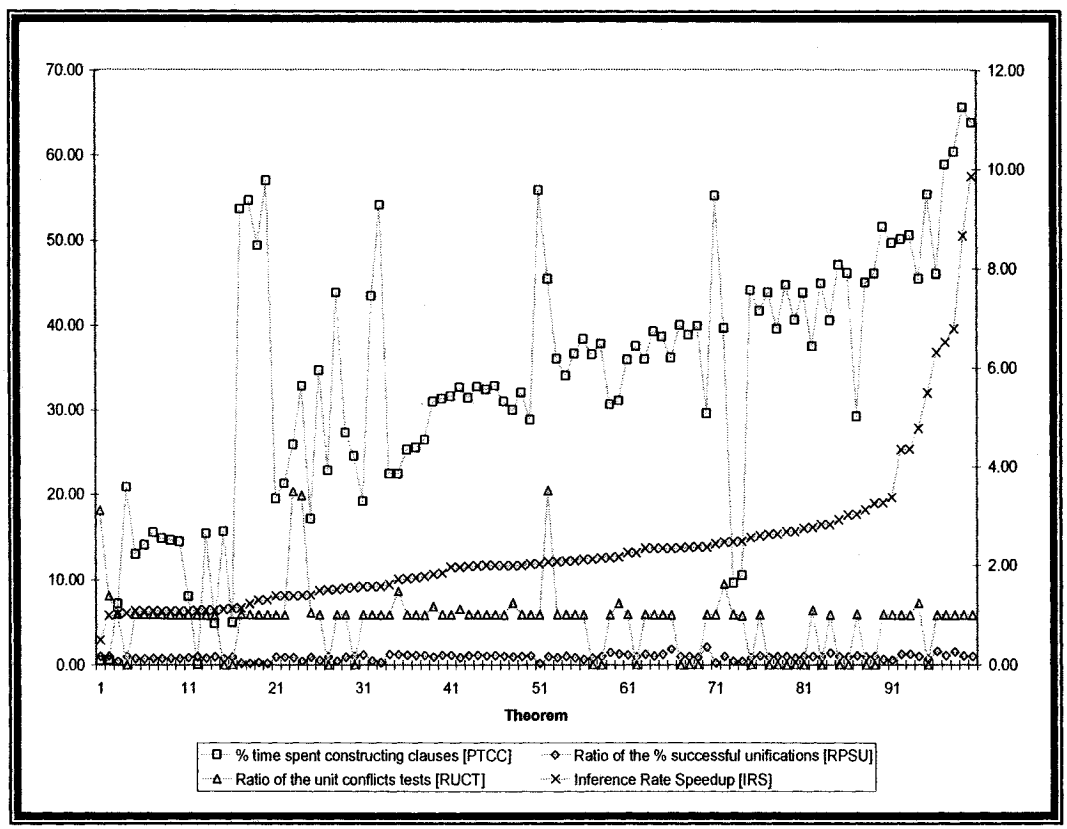



**Figure 6-18: Chart of *PTCC*(*t*), *RPSU*(*t*), *RUCT*(*t*), and *IRS*(*t*) over the selected 100 theorems from the TPTP library v2.6.0.**

The theorems in **Figure 6-18** are sorted by *IRS*(*t*) value. The left side y-axis is a measurement in percent units of the percentage of time spent constructing clauses *PTCC*(*t*), whereas the right y-axis indicates the values for all the other factors as well as the values for the *IRS*(*t*). Since *PTCC*(*t*) is the factor that is most tightly related to the *IRS*(*t*), it is interesting to determine at what point the *PTCC*(*t*) starts to affect the *IRS*(*t*) values significantly. If we look again at the chart in **Figure 6-14**, we notice that after a *PTCC*(*t*) of 1% the inference rate begins to increase, i.e., *IRS*(*t*) > 1. For *PTCC*(*t*) > 33%, the inference rates start to at least double for most theorems. For *PTCC*(*t*) > 50% the inference rates start to at least quadruple on

most theorems. By looking again at **Figure 6-12**, we notice that the average of $PTCC(t)$ over the selected 100 theorems is 33.68%. Since the inference rate on most theorems starts to double when $PTCC(t) > 33\%$, we can claim that on average the inference rate at least doubles when clauses are not constructed. This observation is confirmed by inspecting the chart in **Figure 6-19**.

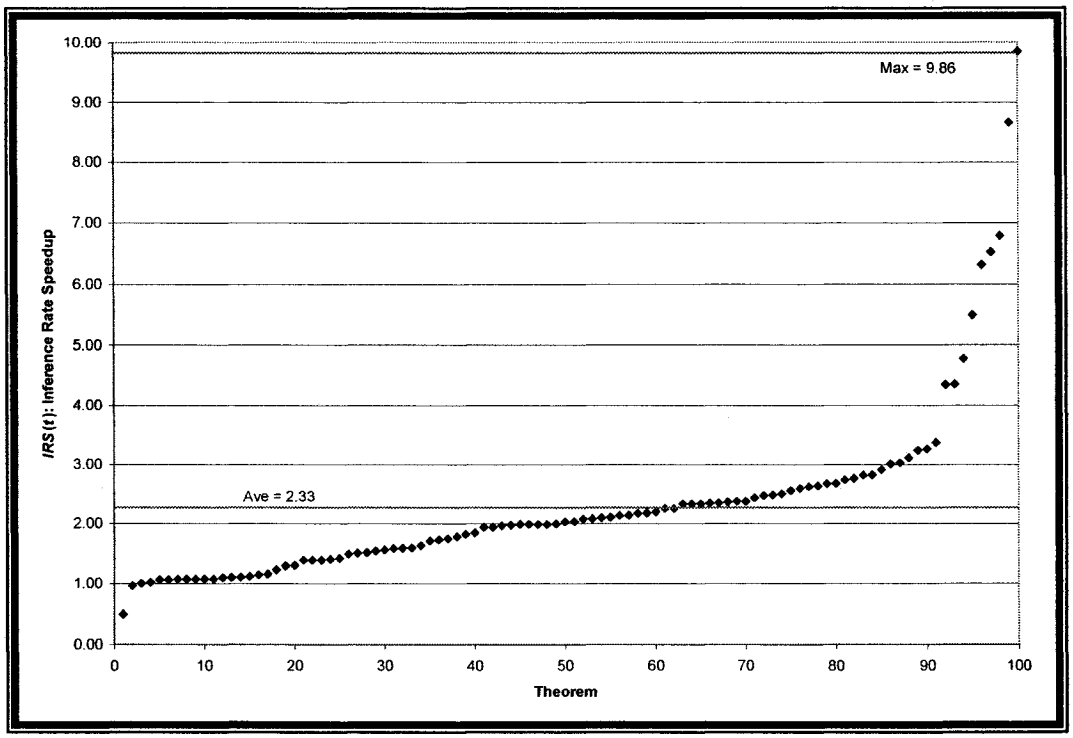



**Figure 6-19: Inference rate speedup of the 100 selected theorems.**

We conclude that even though all three factors, $PTCC(t)$, $RPSU(t)$, and $RUCT(t)$, affect the inference rate speedup, the percentage of time spent in clause construction, $PTCC(t)$, is the one most tightly related to the inference rate speedup and when the time spent constructing clauses is 33% or higher, i.e., $PTCC(t) \geq 33\%$, the use of delayed clause construction may improve the inference rate of an ATP significantly.

# 6.7 The Effect of DCC and ATS on SLR

In this section we present the results that show the effect of DCC and ATS on SLR. We selected all the theorems that CARINE was able to solve (see Appendix H) where the CPU time that CARINE spent to prove a theorem was at least 10 seconds and at most 180 seconds. The total number of those theorems is 106. We then ran CARINE with different configurations on each theorem. The configurations are the following.

**(A) ATS and DCC**: Both ATS and DCC used (default configuration of CARINE)

**(B) No ATS**: The option of using ATS was turned off.

**(C) No DCC**: The option of using DCC was turned off.

**(D) No ATS and No DCC**: The options of using ATS and DCC were turned off.

The number of theorems solved in each configuration is shown in **Table 6-10**.

**Table 6-10: Number of theorems solved by CARINE using different configurations**

| ATS and DCC | No ATS | No DCC | No ATS and No DCC |
|---|---|---|---|
| 106 | 58 | 90 | 39 |

The results show that ATS has more effect than DCC on SLR. When ATS was turned off, CARINE solved only 58 (about 55%) of the 106 theorems, whereas when DCC was turned off, CARINE solved 90 (about 85%) of the 106 theorems. Furthermore, when we compared the number of generated clauses using configuration (B) to the number of generated clause using configuration (A) over the 58 theorems that were solved by (B), we found that, on average, the number of generated clauses almost tripled, as shown in **Table 6-11**.

**Table 6-11: Average number of generated clauses by (A) and (B) over the 58 theorems solved by (B)**

| ATS and DCC | No ATS |
|---|---|
| 8,848,098 | 25,955,223 |

The results in **Table 6-10** and **Table 6-11** indicate that by using ATS, the explored search space was reduced significantly leading to a more efficient search.

## 6.8 Summary

In this chapter, we introduced CARINE, an implementation of semi-linear resolution. We described the data structures involved in the implementation and demonstrated that delayed clause-construction does not require complicated data structures. DCC can be implemented using a path table.

We provided examples that demonstrate how DCC works in practice. The examples also show how attribute sequences can reduce the explorable search space.

We provided experimental results and analyzed the relationship between the inference rate speedup and the factors that affect it. We found that when clauses are not constructed, the inference rate more than doubles on average but it can increase up to 10 times.

Finally, we compared different configurations of CARINE to determine the effect of DCC and ATS on SLR. We found that in terms of number of theorems solved, the impact of ATS on SLR was greater than DCC.

CHAPTER

# 7

# Conclusion

In this chapter we sum up and discuss our work by referring to the main contributions listed in Chapter 1. We then detail further research issues that could be addressed in future work.

## 7.1 Summary and Discussion

We developed two strategies, delayed clause-construction and attributes sequences, to improve the performance of an ATP. We integrated those strategies into a top-down bottom-up search procedure called semi-linear resolution. We built an experimental system, called CARINE, that implements semi-linear resolution. The results obtained from experiments conducted on theorems from the TPTP library using CARINE demonstrated that the methods presented in this thesis are promising and can improve the performance of a resolution-refutation ATP based on depth-first search substantially.

In this thesis we improved the inference rate of CARINE by an average of approximately 2.5 times and in certain cases by as much as 10 times due the use of delayed clause-construction. Consequently, CARINE proved 18% more theorems due to the use of DCC.

We improved the efficiency of CARINE by reducing the explorable search space using attribute sequences. Consequently, CARINE proved almost twice as

many theorems due to the use of attribute sequences than without the use of attribute sequences.

When both DCC and ATS were combined, CARINE was able to prove almost three times the number of theorems than when neither strategy was used. In addition our analysis indicated that the reduction in the size of the explorable search space due to the use of ATS is exponential in the depth bound.

We have shown that an ATP can perform large steps in a search, using a mega-inference rule, which is a consequence of DCC. Although the mega-inference rule allows an ATP to take large steps in a search, the information that can be learned from the small steps is not lost. We have shown that it is possible to obtain information from small steps in one iteration to reduce the explorable search space in the following iterations (see Example 6.3).

## 7.2 Future Work

There are several ways to extend and improve upon the concepts laid out in this thesis. Here we discuss two major issues that we will be working on in the future: expanding the delayed clause-construction to a calculus for substitutions and improving the efficiency of semi-linear resolution through the use of attribute subsequences.

### 7.2.1 A calculus for substitutions

The potential of delayed clause-construction can go beyond simply delaying or avoiding the construction of clauses. The framework established in this thesis is a first attempt to build a system that relies only on the input clauses and a calculus of substitution sets. With such a system, it would no longer be necessary to construct clauses and DCC can be extended to non-linear derivations. The generation of new clauses, obtained by the application of inference rules, will be done through substitution set operations, such as union, intersection and

difference. For instance, the union represent resolutions. We have shown in Chapter 3 that the final p-idempotent substitution set in a linear derivation is the union of all the mgu's obtained from the unification of terms or literals along the linear derivation. Substitution set difference represents backtracking. We have shown in Chapter 6 how backtracking is performed by deleting recent mgu's before backing up to shallower depths. Intersection of substitution sets represent several possibilities depending on the result. We use the intersection operation to discover ancestor resolution, perform non-linear deductions and control redundancy. We give only a simple example on how the intersection operation on substitution sets can help in building non-linear deduction using DCC. For instance, if the intersection of substitution sets is empty, then we either have a variant of an input clause and a linearly derived clause that share no variables with each other, or two linearly derived clauses that share no variables with each other. A resolution between two linearly derived non-constructed clauses $C$ and $D$ that share no variables lead to the generation of a new clause $N$, which is derived in a non-linear way. Thus, we have non-linear deduction using DCC

### 7.2.2 Improving semi-linear resolution

There are several ways to improve the efficiency of semi-linear resolution. We indicated in Chapter 4 that the inclusion of demodulation and paramodulation in semi-linear resolution is possible. We did not provide multiple ways to control the application of these rules. The only guide we suggested to control the application of these rules is through the use of attribute sequences. There are several studies performed to control paramodulation from generating too many clauses. The results of those studies are summarized in [Nieuwenhuis & Rubio 2001] and [Degtyarev & Voronkov 2001]. It is important that these constraints be implemented, otherwise the efficiency of SLR degrades tremendously.

The inclusion of subsumption to reduce redundancy is also an important factor in SLR. However, because clauses are not constructed, we have to rely on the

combined information from the substitution sets and term replacement lists to perform the subsumption. This is an issue related mainly to the extension of DCC to a calculus of substitution set. However, we can also make use of the method proposed in [Schulz 2004] to perform subsumption.

Another improvement to SLR can be achieved by memoization. Memoization is the process of caching (storing most frequently used or most recently used) results. This technique can be highly effective over attribute sequences and subsequences because it eliminates longer sequences based on the knowledge obtained from shorter sequences (see Example 6.3).

There are many other possibilities to enhance the efficiency of SLR through attribute sequences pruning. We have only used the length attribute but there are many other clause attributes (weight, maximum term depth, number of constants, number of variables, etc.) that can be used although not all of them allow the ATP to maintain its state of completeness. Nevertheless, attributes other than the length can be used as part of heuristic functions. The heuristic functions may be cost functions that estimate the cost of attribute sequences. Based on the values of the cost functions, the ATP may decide whether to follow an attribute sequence and consequently, proceed with its search over the corresponding search paths or discard the attribute sequence and all the search paths that correspond to it.

As discussed in Chapter 6, literal ordering is only done initially over the input clauses. We can order the literals in a path table that are not marked deleted. The inclusion of literal ordering may provide a more flexible way to select the potential literals for resolution. This may lead to the termination of an unfruitful path early in the derivation and increase the overall efficiency of an ATP.

# Appendix A

The *Thousands of Problems for Theorem Provers* (TPTP) set is a library of around 7000 theorems[1] that is currently used by at least 118 scientists[2] as test problems for their automated theorem provers. It is maintained and updated by Geoff Sutcliffe and Christian Suttner [TPTP site]. The TPTP set consists of over 30 domains containing *conjunctive normal form* (CNF) and *first-order formula* (FOF) problems.

The studies that we have conducted are over CNF problems, and hence, we have used only the theorems from the CNF problems for our experiments. Our results were obtained from tests performed on theorems from TPTP version 2.6.0. We chose TPTP for our experiments for the following reasons:

- The large variety of CNF theorems spread over many domains.
- The wide range of characteristics that the theorems have. For example, the number of clauses in some problems can be as small as 2 and in others as large as 3240. Also theorems may contain only propositional clauses or only first-order clauses or a combination of both.
- The rating of every theorem according to its difficulty. The difficulty of a theorem is measured by a real number between 0 and 1 and is based on the number of registered[3] theorem provers that were able to solve the problem. A rating of 0 for a problem implies that all registered ATPs

[1] Version 2.6.0 of the TPTP contains 6973 theorems spread over 31 domains.
[2] This is the number of registered users.
[3] The registered theorem provers are state-of-the-art ATPs submitted to the editors of the TPTP library.

solved the problem while a rating of 1 implies that no ATP solved the problem.

- The growing number of users using the TPTP set which is turning this set into a de facto standard for testing new or improved ATPs.
- The good maintenance and support by the editors.
- The comprehensive documentation on every theorem, which includes its rating, satisfiability status, author, reference, domain, brief description, and characteristics (number of clauses, literals, functors and so on), and the statistics and synopsis on the overall set.
- The availability of the library for free.

With such a large number of theorems spanning a wide range of characteristics, we can test our experimental ATP with confidence that the results obtained from the experiments provide an adequate projection of its speed and efficiency from an empirical point of view. Testing an ATP over a large number of theorems also reveals its stability (e.g., does not cause an error which halts the operation of the machine leading to a reboot of the system) and reliability (e.g., the proof is sound and the output is correct).

The table below shows some statistics on the TPTP version 2.6.0 problem library. It lists all the domain names and their abbreviations. The abbreviations are used as a prefix for naming theorems. For example, GEO006-1 is a theorem from the geometry domain. The average number of clauses in each domain for the CNF problems is indicated along with the minimum and maximum number of clauses in any theorem within a domain. The average rating for each domain provides a rough indication on the *difficulty* of the theorems within the domain. The higher the rating value in a domain, the more difficult are the problems in this domain. The NUM and SET domains have the highest ratings. This indicates that a lot of the problems in those domains are difficult.

| Domain | Abbrv. | Number of CNF problems | Number of FOF problems | Ave. number of clauses CNF (min-max) | Ave. rating of CNF |
|---|---|---|---|---|---|
| General Algebra [M] | ALG | 12 | 0 | 97 (9-24) | 0.500 |
| Analysis [M] | ANA | 21 | 0 | 26 (12-50) | 0.615 |
| Boolean Algebra [M] | BOO | 139 | 0 | 16 (7-49) | 0.423 |
| Category Theory [M] | CAT | 62 | 0 | 28 (12-37) | 0.092 |
| Combinatory Logic [L] | COL | 165 | 0 | 10 (7-22) | 0.246 |
| Computing Theory [CS] | COM | 6 | 3 | 27 (11-50) | 0.063 |
| Fields [M] | FLD | 281 | 0 | 33 (27-49) | 0.588 |
| Geometry [M] | GEO | 249 | 77 | 84 (6-169) | 0.554 |
| Graph Theory [M] | GRA | 1 | 0 | 12 | 0.000 |
| Groups [M] | GRP | 791 | 3 | 26 (4-328) | 0.193 |
| Homological Algebra [M] | HAL | 0 | 9 | - | - |
| Henkin Models [L] | HEN | 67 | 0 | 20 (10-36) | 0.008 |
| Hardware Creation [E] | HWC | 6 | 0 | 42 (9-79) | 0.167 |
| Hardware Verification [E] | HWV | 81 | 0 | 140 (21-205) | 0.242 |
| Knowledge Representation Schemes [CS] | KRS | 17 | 0 | 24 (4-54) | 0.000 |
| Lattices [M] | LAT | 104 | 0 | 17 (7-50) | 0.378 |
| Logic Calculi [L] | LCL | 527 | 4 | 12 (3-34) | 0.377 |
| Left Distributive Algebra [M] | LDA | 23 | 0 | 26 (10-36) | 0.732 |
| Management [SS] | MGT | 78 | 78 | 39 (8-85) | 0.141 |
| Miscellaneous | MSC | 13 | 1 | 33 (6-204) | 0.180 |
| Natural Language Processing [CS] | NLP | 258 | 258 | 140 (30-285) | 0.194 |
| Number Theory [M] | NUM | 315 | 0 | 248 (6-409) | 0.890 |
| Planning [CS] | PLA | 32 | 6 | 28 (10-31) | 0.239 |
| Puzzles | PUZ | 74 | 4 | 42 (5-504) | 0.127 |
| Rings [M] | RNG | 104 | 0 | 32 (8-74) | 0.394 |
| Robbins Algebra [M] | ROB | 38 | 0 | 14 (9-24) | 0.487 |
| Set Theory [M] | SET | 704 | 326 | 185 (2-295) | 0.746 |
| Software Creation [CS] | SWC | 423 | 423 | 236 (222-333) | 0.650 |
| Software Verification [CS] | SWV | 20 | 9 | 25 (3-41) | 0.151 |
| Syntactic | SYN | 838 | 299 | 261 (2-3240) | 0.139 |
| Topology [M] | TOP | 24 | 0 | 91 (3-119) | 0.535 |
| | **Total:** | 5473 | 1500 | | |

[CS] = Computer Science [E] = Engineering [L] = Logic

[M] = Mathematics [SS] = Social Sciences

# Appendix B

This appendix contains the proofs of theorems from Chapter 3.

## Proof of Theorem 3.2

**Theorem 3.2**

*Given two idempotent substitution sets* $\sigma_1$ *and* $\sigma_2$ *that are consistent, if none of the variables in* $Dom(\sigma_1)$ *occurs in any of the terms in* $Ran(\sigma_2)$, *then* $\sigma_1 \cup \sigma_2 \rightarrow \sigma_1\sigma_2$.

**Proof:**

To prove that $\sigma_1 \cup \sigma_2 \rightarrow \sigma_1\sigma_2$ means to show that the application of $\sigma_1 \cup \sigma_2$ a finite number of times over itself should lead to the set $\sigma_1\sigma_2$. This means that $\sigma_1\sigma_2$ should be equal to $(\sigma_1 \cup \sigma_2)\ldots(\sigma_1 \cup \sigma_2)$.

Suppose that $\sigma_1 = \{v_1 \rightarrow t_1, \ldots, v_n \rightarrow t_n\}$ and $\sigma_2 = \{u_1 \rightarrow s_1, \ldots, u_m \rightarrow s_m\}$ are idempotent substitution sets that are consistent. Since $\sigma_1$ and $\sigma_2$ are consistent, then by Definition 3.6 $Dom(\sigma_1) \cap Dom(\sigma_2) = \{\}$ and $\sigma_1 \cup \sigma_2$ is p-idempotent. Since $\sigma_1 \cup \sigma_2$ is p-idempotent then let

$$\begin{aligned}\ddot{\theta} &= \sigma_1 \cup \sigma_2 \\ &= \{v_1 \rightarrow t_1, \ldots, v_n \rightarrow t_n, u_1 \rightarrow s_1, \ldots, u_m \rightarrow s_m\}.\end{aligned}$$

The application of $\ddot{\theta}$ on $\ddot{\theta}$ gives

$$\ddot{\theta}\ddot{\theta} = (\sigma_1 \cup \sigma_2)\ddot{\theta} = \{v_1 \rightarrow t_1\ddot{\theta}, \ldots, v_n \rightarrow t_n\ddot{\theta}, u_1 \rightarrow s_1\ddot{\theta}, \ldots, u_m \rightarrow s_m\ddot{\theta}\}.$$

Since $\sigma_1$ is idempotent then the application of the subset of $\ddot{\theta}$ that is equal to $\sigma_1$ has no effect on the substitution terms $t_1,\ldots,t_n$. Similarly, the application of the subset of $\ddot{\theta}$ that is equal to $\sigma_2$ has no effect on the substitution terms $s_1,\ldots,s_n$. Therefore,

$$\ddot{\theta}\ddot{\theta} = \{v_1 \to t_1\sigma_2,\ldots,v_n \to t_n\sigma_2, u_1 \to s_1\sigma_1,\ldots,u_m \to s_m\sigma_1\}.$$

If none of the variables in $Dom(\sigma_1)$ occurs in $Ran(\sigma_2)$, then $\sigma_1$ has no effect on the terms $s_1,\ldots,s_n$, i.e., for all $1 \le i \le n$, for all $1 \le j \le m$, $v_i \notin^o s_j$, then $s_j\sigma_1 = s_j$. This implies that $\ddot{\theta}\ddot{\theta}$ can be written as

$$\ddot{\theta}\ddot{\theta} = \{v_1 \to t_1\sigma_2,\ldots,v_n \to t_n\sigma_2, u_1 \to s_1,\ldots,u_m \to s_m\}.$$

The application of $\ddot{\theta}\ddot{\theta}$ over $\ddot{\theta}$ gives

$$\begin{aligned}\ddot{\theta}(\ddot{\theta}\ddot{\theta}) &= (\sigma_1 \cup \sigma_2)(\ddot{\theta}\ddot{\theta}) \\ &= \{v_1 \to t_1\ddot{\theta}\ddot{\theta},\ldots,v_n \to t_n\ddot{\theta}\ddot{\theta}, u_1 \to s_1\ddot{\theta}\ddot{\theta},\ldots,u_m \to s_m\ddot{\theta}\ddot{\theta}\}.\end{aligned}$$

The subset $\{v_1 \to t_1\sigma_2,\ldots,v_n \to t_n\sigma_2\}$ of $\ddot{\theta}\ddot{\theta}$ has no effect on the substitution terms $t_1,\ldots,t_n$ because $\sigma_1$ is idempotent, so none of the variables $v_1,\ldots,v_n$ occurs in any of the terms $t_1,\ldots,t_n$. Furthermore, none of the variables in $Dom(\sigma_1)$ occurs in $Ran(\sigma_2)$ (given). So when $\ddot{\theta}\ddot{\theta}$ was formed, the application of $\sigma_2$ over $t_1,\ldots,t_n$ did not introduce any of the variables in $Dom(\sigma_1)$ into any of the resulting terms $t_1\sigma_2,\ldots,t_n\sigma_2$. In other words, for all $1 \le i \le n$, for all $1 \le j \le n$, $v_i \notin^o t_j\sigma_2$. Moreover, the subset $\{v_1 \to t_1\sigma_2,\ldots,v_n \to t_n\sigma_2\}$ of $\ddot{\theta}\ddot{\theta}$ has no effect on the substitution terms $s_1,\ldots,s_m$ because none of the variables in $Dom(\sigma_1)$ occurs in $Ran(\sigma_2)$. The only subset of $\ddot{\theta}\ddot{\theta}$ that may affect $\ddot{\theta}$ is $\{u_1 \to s_1,\ldots,u_m \to s_m\}$,

but this is the same as set $\sigma_2$. So the application of $\ddot{\theta}\ddot{\theta}$ to $\ddot{\theta}$ is reduced to the application of $\sigma_2$ to $\ddot{\theta}$. Therefore

$$\begin{aligned}\ddot{\theta}(\ddot{\theta}\ddot{\theta}) &= \ddot{\theta}\sigma_2 \\ &= \{v_1 \to t\sigma_2, \ldots, v_n \to t_n\sigma_2, u_1 \to s_1\sigma_2, \ldots, u_m \to s_m\sigma_2\}.\end{aligned}$$

Since $\sigma_2$ is idempotent then it has no effect on the terms $s_1, \ldots, s_m$. Therefore,

$$\begin{aligned}\ddot{\theta}(\ddot{\theta}\ddot{\theta}) &= (\sigma_1 \cup \sigma_2)\sigma_2 \\ &= \{v_1 \to t\sigma_2, \ldots, v_n \to t_n\sigma_2, u_1 \to s_1, \ldots, u_m \to s_m\}.\end{aligned}$$

Notice that the set obtained from applying $\ddot{\theta}\ddot{\theta}$ to $\ddot{\theta}$ is same as $\ddot{\theta}\ddot{\theta}$, i.e., $\ddot{\theta}(\ddot{\theta}\ddot{\theta}) = \ddot{\theta}\ddot{\theta}$. This implies that any further application to $\ddot{\theta}$ is not going to produce a set different from $\ddot{\theta}\ddot{\theta}$. Let $\theta = \ddot{\theta}\ddot{\theta}$ then $\ddot{\theta} \to \theta$.

The set $\theta = \{v_1 \to t\sigma_2, \ldots, v_n \to t_n\sigma_2, u_1 \to s_1, \ldots, u_m \to s_m\}$. The set $\sigma_1\sigma_2$ as defined in Chapter 2 is $\sigma_1\sigma_2 = \{v_1 \to t\sigma_2, \ldots, v_n \to t_n\sigma_2, u_1 \to s_1, \ldots, u_m \to s_m\}$. Therefore $\theta$ is exactly $\sigma_1\sigma_2$. So $\theta = \sigma_1\sigma_2$ and hence $\ddot{\theta}\ddot{\theta} = \sigma_1\sigma_2$. This implies that $(\sigma_1 \cup \sigma_2)(\sigma_1 \cup \sigma_2) = \sigma_1\sigma_2$. Therefore $\sigma_1 \cup \sigma_2 \to \sigma_1\sigma_2$. □

## Proof of Theorem 3.3

**Theorem 3.3**

*Given* $k \geq 2$ *idempotent substitution sets* $\sigma_1,\ldots,\sigma_k$ *that are pair-wise consistent, if for each* $1 \leq i \leq k-1$ *none of the variables in* $Dom(\sigma_i)$ *occurs in any of the terms in any* $Ran(\sigma_j)$, *where* $i+1 \leq j \leq k$, *then* $\sigma_1 \cup \cdots \cup \sigma_k \rightarrow \sigma_1 \cdots \sigma_k$.

**Proof:**

This theorem is a generalization of Theorem 3.2. It can be proved by induction.

Base case: $k = 2$.

Given two idempotent substitution sets $\sigma_1$ and $\sigma_2$ that are consistent, if for all $v \in Dom(\sigma_1) \Rightarrow v \not\in^o Ran(\sigma_2)$, then $\sigma_1 \cup \sigma_2 \rightarrow \sigma_1\sigma_2$ by Theorem 3.2.

General case: $k > 2$.

Suppose that for $n < k$, $\sigma_1,\ldots,\sigma_n$ are idempotent, pair-wise consistent, and for every $1 \leq i \leq n$, for all $v \in Dom(\sigma_i) \Rightarrow v \not\in^o \bigcup_{j=i+1}^{n} Ran(\sigma_j)$. Assume that for all $n < k$, $\sigma_1 \cup \cdots \cup \sigma_n \rightarrow \sigma_1 \cdots \sigma_n$ is true. Since $\sigma_1,\ldots,\sigma_n$ are consistent (given), then their union is p-idempotent by Definition 3.6. Let $\ddot{\theta} = \sigma_1 \cup \cdots \cup \sigma_n$, then $\ddot{\theta} \rightarrow \sigma_1 \cdots \sigma_n$ (assumption). The composition of idempotent sets is idempotent (see Chapter 2); hence $\sigma_1 \cdots \sigma_n$ is idempotent. Let $\theta = \sigma_1 \cdots \sigma_n$, then $\ddot{\theta} \rightarrow \theta$ and $\theta$ is idempotent.

Suppose that $\sigma_{n+1}$ is idempotent, consistent with every $\sigma_i$, $1 \leq i \leq n$, and for all $v \in Dom(\sigma_i) \Rightarrow v \not\in^o Ran(\sigma_{n+1})$, then $\sigma_1 \cup \cdots \cup \sigma_{n+1} = \ddot{\theta} \cup \sigma_{n+1}$. Since $\ddot{\theta} \rightarrow \theta$,

then by Theorem 3.1 and the second property in Definition 3.7, $\ddot{\theta} \cup \sigma_{n+1}$ is confluent to $\theta \cup \sigma_{n+1}$, i.e., $\theta \cup \sigma_{n+1} \Downarrow \ddot{\theta} \cup \sigma_{n+1}$.

$\theta$ is idempotent and $\sigma_{n+1}$ is idempotent (given). Furthermore, none of the variables in $\theta$ occurs in $Ran(\sigma_{n+1})$ because all the variables in $\theta$ belong to $\bigcup_{i=1}^{n} Dom(\sigma_i)$ and they do not occur in $Ran(\sigma_{n+1})$ (given). Therefore, $\theta \cup \sigma_{n+1} \to \theta\sigma_{n+1}$ by Theorem 3.2. Since $\theta \cup \sigma_{n+1} \Downarrow \ddot{\theta} \cup \sigma_{n+1}$ and $\theta = \sigma_1 \cdots \sigma_n$, then $\theta \cup \sigma_{n+1} \to \theta\sigma_{n+1} \Rightarrow \ddot{\theta} \cup \sigma_{n+1} \to \sigma_1 \cdots \sigma_n \sigma_{n+1}$, but $\ddot{\theta} = \sigma_1 \cup \cdots \cup \sigma_n$ therefore $\sigma_1 \cup \cdots \cup \sigma_n \cup \sigma_{n+1} \to \sigma_1 \cdots \sigma_n \sigma_{n+1}$. Since by assumption $\sigma_1 \cup \cdots \cup \sigma_n \to \sigma_1 \cdots \sigma_n$ is true for all $n < k$, then $n+1 = k$ and so

$$\sigma_1 \cup \cdots \cup \sigma_n \cup \sigma_{n+1} \to \sigma_1 \cdots \sigma_n \sigma_{n+1} \Leftrightarrow \sigma_1 \cup \cdots \cup \sigma_n \cup \sigma_k \to \sigma_1 \cdots \sigma_n \sigma_k .$$

Therefore, for any $k \geq 2$, $\sigma_1 \cup \cdots \cup \sigma_k \to \sigma_1 \cdots \sigma_k$. □

## Proof of Theorem 3.5

Theorem 3.5 can be stated formally as follows.

**Given:**

I. A set of constructed clauses $S = \{B_1,\ldots,B_n\}$, where $n \geq 2$.

II. A linear derivation $\Delta = \langle I_1,\ldots,I_k \rangle$, where $k \geq 1$, of a goal clause $G$ from $S$ with $C_{init} \in S$, such that

1. every clause in the multiset union $\bigcup_{i=1}^{k} \mathcal{D}(I_i)$ is a variant of a constructed clause from $S$.
2. $Vars(C_{init}) \cap Vars(\bigcup_{i=1}^{k} \mathcal{D}(I_i)) = \{\}$ and for all $C \in \bigcup_{i=1}^{k} \mathcal{D}(I_i)$ and for all $D \in \bigcup_{i=1}^{k} \mathcal{D}(I_i)$, if $C \neq D$ then $Vars(C) \cap Vars(D) = \{\}$.
3. The mgu's $\sigma_1,\ldots,\sigma_k$ resulting from the inferences $I_1,\ldots,I_k$, are idempotent and consistent.
4. For all $1 \leq i \leq k$, no $C(I_i)$ is a *from clause*. Therefore, if $I_i$ is a paramodulation, demodulation, or superposition, then the *from clause* is a variant of a clause from $S$.

**Required to prove:**

Every $\ddot{C}(\mathcal{I}_i)$, $1 \le i \le k$, can be expressed as

$$\ddot{C}(\mathcal{I}_i) = ((B^1_{r_1} \cup \cdots \cup B^m_{r_m}) \setminus (\alpha_{1..i} \cup \beta_{1..i}))\ddot{\sigma}_{1..i}(\tau_{1..i}\ddot{\sigma}_{1..i}),$$

where

- $m$ is the total number of variants of clauses from $S$ used in $\Delta$,
- for all $1 \le j \le m$, $r_j \in \{1,\ldots,n\}$, $n = |S|$ and $B^j_{r_j} = B_{r_j}\theta_j$, where $B_{r_j} \in S$, and $\theta_j$ is variable renaming substitution, such that
- $Ran(\theta_j) \cap (\bigcup_{q=1}^{j-1} Vars(B^q_{r_q}) \cup Vars(C_{init})) = \{\}$,
- $\alpha_{1..i} = \alpha_1 \cup \cdots \cup \alpha_i$, where $\alpha_1 \subseteq C_{init}$ and for all $1 \le j \le i$,
- $\alpha_j \subseteq \mathcal{L}(\mathcal{D}(\mathcal{I}_1) \cup \cdots \cup \mathcal{D}(\mathcal{I}_{j-1}))$,
- $\beta_{1..i} = \beta_1 \cup \cdots \cup \beta_i$, where for all $1 \le j \le i$,
- $\beta_j \subseteq \mathcal{L}(\mathcal{D}(\mathcal{I}_j))$,
- $\sigma_{1..i} = \sigma_1 \cdots \sigma_i$ and $\tau_{1..i} = \tau_1 \cdots \tau_i$.

**Proof:**

For all $1 \le i \le k$, inference $\mathcal{I}_i$ can be written as

$$\frac{Prem(\mathcal{I}_i)}{C(\mathcal{I}_i)}, \gamma_i \qquad \textbf{(EB.1)}$$

$Prem(\mathcal{I}_i)$ is a multiset of clauses implicitly representing a conjunction clauses. Since $\Delta$ is a linear derivation then by definition (see Chapter 2),

$$Prem(\mathcal{I}_i) = \begin{cases} \{C_{init}\} \cup \mathcal{D}(\mathcal{I}_1) & i = 1, \\ \{C(\mathcal{I}_{i-1})\} \cup \mathcal{D}(\mathcal{I}_i) & 2 \le i \le k. \end{cases} \qquad \textbf{(EB.2)}$$

From **EB.2** we can form the multiset $\mathcal{L}(Prem(\mathcal{I}_i))$ as follows.

$$\mathcal{L}(Prem(\mathcal{I}_i)) = \begin{cases} C_{init} \cup \mathcal{L}(\mathcal{D}(\mathcal{I}_1)) & i = 1, \\ C(\mathcal{I}_{i-1}) \cup \mathcal{L}(\mathcal{D}(\mathcal{I}_i)) & 2 \le i \le k. \end{cases} \qquad \textbf{(EB.3)}$$

Definition 3.11 indicates that the conclusion of an inference $\mathcal{I}_i$ can be written in the form $((\boldsymbol{C}_i \setminus \boldsymbol{D}_i) \cup E_i)\sigma_i(\tau_i\sigma_i)$, where $\boldsymbol{C}_i = \mathcal{L}(Prem(\mathcal{I}_i))$, $\boldsymbol{D}_i \subseteq \mathcal{L}(Prem(\mathcal{I}_i))$, $\boldsymbol{E}_i \subseteq \{\neg L_i\}$, and $L_i \in \mathcal{L}(Prem(\mathcal{I}_i))$. Since the conclusion of an inference $\mathcal{I}_i$ can be written in the form $((\boldsymbol{C}_i \setminus \boldsymbol{D}_i) \cup \boldsymbol{E}_i)\sigma_i(\tau_i\sigma_i)$, then a non-constructed version of $C(\mathcal{I}_i)$ can be expressed as

$$\ddot{C}(\mathcal{I}_i) = ((\boldsymbol{C}_i \setminus \boldsymbol{D}_i) \cup \boldsymbol{E}_i)\sigma_i(\tau_i\sigma_i). \qquad \textbf{(EB.4)}$$

In what follows, $\sigma_i$ is the mgu resulting from the unification of some terms or literals from the premises. Similarly, $\tau_i$ is a term replacement list of some terms from the literals of the clauses in the premises of inference $\mathcal{I}_i$. Chapter 2 and **Table 3-4** provide more details on the mgu's and term replacement lists for specific inference rules.

Base case: $k = 1$

$$\begin{aligned} \ddot{C}(\mathcal{I}_1) &= ((\boldsymbol{C}_1 \setminus \boldsymbol{D}_1) \cup \boldsymbol{E}_1)\sigma_1(\tau_1\sigma_1) \\ &= ((C_{init} \cup \mathcal{L}(\mathcal{D}(\mathcal{I}_1)) \cup \boldsymbol{E}_1) \setminus (\alpha_1 \cup \beta_1))\sigma_1(\tau_1\sigma_1) \end{aligned} \qquad \textit{Using EB.3 and EB.4}$$

where $\alpha_1 \subseteq C_{init}$ and $\beta_1 \subseteq \mathcal{L}(\mathcal{D}(\mathcal{I}_1))$. If $\boldsymbol{E}_1 \neq \{\}$ then $\boldsymbol{E}_1$ contains the negation of a literal $L_1$ from $C_{init}$, i.e., $\boldsymbol{E}_1 = \{\neg L_1\}$ where $L_1 \in C_{init}$. Notice that $L_1$ cannot be an element from $\mathcal{L}(\mathcal{D}(\mathcal{I}_1))$ because in equality factoring, there is only one premise, so $\mathcal{L}(\mathcal{D}(\mathcal{I}_1)) = \{\}$.

Case: $k = 2$

$$\ddot{C}(I_2) = ((\boldsymbol{C}_2 \setminus \boldsymbol{D}_2) \cup \boldsymbol{E}_2')\sigma_2(\tau_2\sigma_2) \qquad \textit{Using EB.3 and}$$
$$= ((\ddot{C}(I_1) \cup \mathcal{L}(\mathcal{D}(I_2)) \cup \boldsymbol{E}_2') \setminus (\alpha_2' \cup \beta_2))\sigma_2(\tau_2'\sigma_2), \qquad \textit{EB.4}$$

where $\alpha_2' = \alpha_2\sigma_1(\tau_1\sigma_1)$ and $\alpha_2\sigma_1(\tau_1\sigma_1) \subseteq \ddot{C}(I_1)$, and $\beta_2 \subseteq \mathcal{L}(\mathcal{D}(I_2))$.

If $\boldsymbol{E}_2' \neq \{\}$ then $I_2$ is equality factoring. Therefore, $\mathcal{L}(\mathcal{D}(I_2)) = \{\}$ and $\boldsymbol{E}_2'$ contains the negation of a literal $L_2' = L_2\sigma_1(\tau_1\sigma_1)$ from $\ddot{C}(I_1)$, i.e., $\boldsymbol{E}_2' = \{\neg L_2'\}$ where $L_2' \in \ddot{C}(I_1)$. Let $\boldsymbol{E}_2 = \{\}$ if $\boldsymbol{E}_2' \neq \{\}$ or $\boldsymbol{E}_2 = \{\neg L_2\}$ if $\boldsymbol{E}_2' = \{\neg L_2'\}$ such that $L_2' = L_2\sigma_1(\tau_1\sigma_1)$, then $\boldsymbol{E}_2' = \boldsymbol{E}_2\sigma_1(\tau_1\sigma_1)$.

Substituting the expression for $\ddot{C}(I_1)$ in $\ddot{C}(I_2)$, we get

$$\dddot{C}(I_2) = (((((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1)) \cup \boldsymbol{E}_1) \setminus (\alpha_1 \cup \beta_1))\sigma_1(\tau_1\sigma_1)) \cup$$
$$\mathcal{L}(\mathcal{D}(I_2)) \cup \boldsymbol{E}_2') \setminus (\alpha_2' \cup \beta_2))\sigma_2(\tau_2'\sigma_2).$$

We want to move $\sigma_1(\tau_1\sigma_1)$ to the front of the above expression. In order to do that, we have to show that the application of $\sigma_1$ and $(\tau_1\sigma_1)$ to $\mathcal{L}(\mathcal{D}(I_2))$, $\boldsymbol{E}_2'$, $\alpha_2'$, $\beta_2$ won't affect them.

Since $\alpha_2' = \alpha_2\sigma_1(\tau_1\sigma_1)$ then when $\sigma_1(\tau_1\sigma_1)$ is moved to the front, $\alpha_2'$ is replaced with $\alpha_2$. Similarly, $\boldsymbol{E}_2'$ is replaced with $\boldsymbol{E}_2$.

From the given, it can be deduced that

$Vars(C_{init}) \cap Vars(\mathcal{D}(I_2)) = \{\}$ and $Vars(\mathcal{D}(I_1)) \cap Vars(\mathcal{D}(I_2)) = \{\}$.

Since $Dom(\sigma_1) \subseteq Vars(C_{init}) \cup Vars(\mathcal{D}(I_1))$ then $Dom(\sigma_1) \cap Vars(\mathcal{D}(I_2)) = \{\}$.

Therefore, $\mathcal{L}(\mathcal{D}(I_2))\sigma_1 = \mathcal{L}(\mathcal{D}(I_2))$.

Since $\beta_2 \subseteq \mathcal{L}(\mathcal{D}(I_2))$ then $\beta_2\sigma_1 = \beta_2$.

$\mathcal{L}(\tau_1) \subseteq C_{init} \cup \mathcal{L}(\mathcal{D}(I_1))$ this implies that $\mathcal{L}(\tau_1) \not\subset \mathcal{L}(\mathcal{D}(I_2))$.

Therefore, $\mathcal{L}(\mathcal{D}(I_2))(\tau_1\sigma_1) = \mathcal{L}(\mathcal{D}(I_2))$.

Since $\beta_2 \subseteq \mathcal{L}(\mathcal{D}(I_2))$ then $\mathcal{L}(\tau_1) \not\subset \beta_2$.

Therefore, $\beta_2(\tau_1\sigma_1) = \beta_2$.

$\ddot{C}(I_2)$ can now be written as

$$\ddot{C}(I_2) = ((((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1)) \cup \boldsymbol{E}_1) \setminus (\alpha_1 \cup \beta_1)) \cup \mathcal{L}(\mathcal{D}(I_2)) \cup \boldsymbol{E}_2) \setminus (\alpha_2 \cup \beta_2))\sigma_1(\tau_1\sigma_1)\sigma_2(\tau_2'\sigma_2).$$

*$\sigma_1(\tau_1\sigma_1)$ moved to the end of expression.*

$$\ddot{C}(I_2) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1)) \cup \boldsymbol{E}_1 \cup \mathcal{L}(\mathcal{D}(I_2)) \cup \boldsymbol{E}_2) \setminus (\alpha_1 \cup \beta_1 \cup \alpha_2 \cup \beta_2))\sigma_1(\tau_1\sigma_1)\sigma_2(\tau_2'\sigma_2)$$

*Using the special DU law (see Chapter 2).*

$$\ddot{C}(I_2) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1)) \cup \mathcal{L}(\mathcal{D}(I_2)) \cup \boldsymbol{E}_1 \cup \boldsymbol{E}_2) \setminus (\alpha_1 \cup \alpha_2 \cup \beta_1 \cup \beta_2))\sigma_1(\tau_1\sigma_1)\sigma_2(\tau_2'\sigma_2)$$

*Union is commutative.*

$$\ddot{C}(I_2) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \mathcal{D}(I_2)) \cup \boldsymbol{E}_1 \cup \boldsymbol{E}_2) \setminus (\alpha_1 \cup \beta_1 \cup \alpha_2 \cup \beta_2))\sigma_1(\tau_1\sigma_1)\sigma_2(\tau_2'\sigma_2)$$

$\mathcal{L}(\mathcal{D}(I_1)) \cup \mathcal{L}(\mathcal{D}(I_2)) = \mathcal{L}(\mathcal{D}(I_1) \cup \mathcal{D}(I_2))$

$$\ddot{C}(I_2) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \mathcal{D}(I_2)) \cup \boldsymbol{E}_1 \cup \boldsymbol{E}_2) \setminus (\alpha_1 \cup \beta_1 \cup \alpha_2 \cup \beta_2))\sigma_1\sigma_2(\tau_1\sigma_1\sigma_2)(\tau_2'\sigma_2)$$

*Distributing $\sigma_2$ over $\sigma_1$ and $\tau_1\sigma_1$*

We now need to transform $\tau_2'$ into a term replacement list, $\tau_2$, in which every referenced literal $\vec{L}'$ in $\tau_2'$ is replaced by a reference to the *original* literal $\vec{L}$. By *original* literal we mean the literal $\vec{L}$ of the variant of the input clause before $\sigma_1$

and/or $\tau_1$ is applied to $\vec{L}$. The purpose of this transformation is to be able to change the expression $(\tau_1\sigma_1\sigma_2)(\tau_2'\sigma_2)$ into $(\tau_1\tau_2(\sigma_1\sigma_2))$.

Recall that a term replacement list is not concerned with the actual terms being replaced but with their positions. As long as the positions are valid, the actual term being replaced is not important. Also, recall that a term replacement list is an ordered multiset that when applied to a clause, the application is performed from left to right.

$\mathcal{L}(\tau_2') \subseteq C(I_1) \cup \mathcal{L}(\mathcal{D}(I_2))$.

Therefore, for all $(\vec{L}'|_\pi \rightarrow t) \in \tau_2'$, $L' \in C(I_1)$ or $L' \in \mathcal{L}(\mathcal{D}(I_2))$.

If $L' \in C(I_1)$ then $\exists L \in C_{init} \cup \mathcal{L}(\mathcal{D}(I_1)) \cup \boldsymbol{E}_1$ such that $L' = L\sigma_1(\tau_1\sigma_1)$. $L$ is the original literal.

If $L' \in \mathcal{L}(\mathcal{D}(I_2))$ then since $\mathcal{L}(\mathcal{D}(I_2))\sigma_1(\tau_1\sigma_1) = \mathcal{L}(\mathcal{D}(I_2))$ (shown above), then $L'\sigma_1(\tau_1\sigma_1) = L'$. $L'$ is the original literal.

We form $\tau_2$ from $\tau_2'$ as follows.

For all $(\vec{L}'|_\pi \rightarrow t) \in \tau_2'$, if $L' \in C(I_1)$ then we replace it with the original $L$, and if $L' \in \mathcal{L}(\mathcal{D}(I_2))$ then we keep it.

Part II.4 of the given states that no intermediate conclusion is a *from clause*, therefore, for all $(\vec{L}'|_\pi \rightarrow t) \in \tau_2'$, the term $t$ is a term from a literal in $\mathcal{L}(\mathcal{D}(I_2))$. Since $\mathcal{L}(\mathcal{D}(I_2))\sigma_1 = \mathcal{L}(\mathcal{D}(I_2))$ (shown above), then $t\sigma_1 = t$.

$\ddot{C}(I_2)$ can now be written as

$$\ddot{C}(I_2) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \mathcal{D}(I_2))) \setminus (\alpha_1 \cup \beta_1 \cup \alpha_2 \cup \beta_2))\sigma_1\sigma_2(\tau_1\tau_2(\sigma_1\sigma_2))$$

*Combining $\tau_1$ and $\tau_2$.*

Let $\sigma_{1..2} = \sigma_1\sigma_2$, $\tau_{1..2} = \tau_1\tau_2$, $\alpha_{1..2} = \alpha_1 \cup \alpha_2$, and $\beta_{1..2} = \beta_1 \cup \beta_2$. $\ddot{C}(I_2)$ becomes

$$\ddot{C}(I_2) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \mathcal{D}(I_2))) \setminus (\alpha_{1..2} \cup \beta_{1..2}))\sigma_{1..2}(\tau_{1..2}\sigma_{1..2})$$

General case: $k > 2$

Assume that for all $i < k$, that

$$\ddot{C}(I_i) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_i))) \setminus (\alpha_{1..i} \cup \beta_{1..i}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i}).$$

Show that for $k = i+1$,

$$\ddot{C}(I_{i+1}) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_{i+1}))) \setminus (\alpha_{1..i+1} \cup \beta_{1..i+1}))\sigma_{1..i+1}(\tau_{1..i+1}\sigma_{1..i+1}).$$

$$\begin{aligned}\ddot{C}(I_{i+1}) &= (\boldsymbol{C}_{i+1} \setminus \boldsymbol{D}_{i+1})\sigma_{i+1}(\tau_{i+1}\sigma_{i+1}) \\ &= ((\ddot{C}(I_i) \cup \mathcal{L}(\mathcal{D}(I_{i+1}))) \setminus (\alpha_{i+1} \cup \beta_{i+1}))\sigma_{i+1}(\tau_{i+1}\sigma_{i+1}),\end{aligned}$$

where $\alpha_{i+1} \subseteq \ddot{C}(I_i)$ and $\beta_{i+1} \subseteq \mathcal{L}(\mathcal{D}(I_{i+1}))$.

The substitution of the expression for $\ddot{C}(I_i)$ into the expression for $\ddot{C}(I_{i+1})$ leads to

$$\ddot{C}(I_{i+1}) = (((((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_i))) \setminus (\alpha_{1..i} \cup \beta_{1..i}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i})) \cup \mathcal{L}(\mathcal{D}(I_{i+1}))) \setminus (\alpha_{i+1} \cup \beta_{i+1}))\sigma_{i+1}(\tau_{i+1}\sigma_{i+1})$$

$$\ddot{C}(I_{i+1}) = ((((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_i))) \setminus (\alpha_{1..i} \cup \beta_{1..i})) \cup \mathcal{L}(\mathcal{D}(I_{i+1}))) \setminus (\alpha_{i+1} \cup \beta_{i+1}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i})\sigma_{i+1}(\tau_{i+1}\sigma_{i+1})$$

$$\ddot{C}(I_{i+1}) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_i)) \cup \mathcal{L}(\mathcal{D}(I_{i+1}))) \setminus (\alpha_{1..i} \cup \beta_{1..i} \cup \alpha_{i+1} \cup \beta_{i+1}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i})\sigma_{i+1}(\tau_{i+1}\sigma_{i+1})$$

$$\ddot{C}(I_{i+1}) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_i) \cup \mathcal{D}(I_{i+1}))) \setminus (\alpha_{1..i} \cup \beta_{1..i} \cup \alpha_{i+1} \cup \beta_{i+1}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i})\sigma_{i+1}(\tau_{i+1}\sigma_{i+1})$$

$$\ddot{C}(I_{i+1}) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_{i+1}))) \setminus (\alpha_{1..i} \cup \beta_{1..i} \cup \alpha_{i+1} \cup \beta_{i+1}))\sigma_{1..i}\sigma_{i+1}(\tau_{1..i}\sigma_{1..i}\sigma_{i+1})(\tau_{i+1}\sigma_{i+1})$$

$$\ddot{C}(I_{i+1}) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_{i+1}))) \setminus (\alpha_{1..i} \cup \beta_{1..i} \cup \alpha_{i+1} \cup \beta_{i+1}))\sigma_{1..i}\sigma_{i+1}(\tau_{1..i}\tau_{i+1}(\sigma_{1..i}\sigma_{i+1}))$$

Let $\sigma_{1..i+1} = \sigma_{1..i}\sigma_{i+1}$, $\tau_{1..i+1} = \tau_{1..i}\tau_{i+1}$, $\alpha_{1..i+1} = \alpha_{1..i} \cup \alpha_{i+1}$, and $\beta_{1..i+1} = \beta_{1..i} \cup \beta_{i+1}$. $\ddot{C}(I_{i+1})$ becomes

$$\ddot{C}(I_{i+1}) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_{i+1}))) \setminus (\alpha_{1..i+1} \cup \beta_{1..i+1}))\sigma_{1..i+1}(\tau_{1..i+1}\sigma_{1..i+1}).$$

The general expression for an intermediate conclusion can be derived from

$$\ddot{C}(I_i) = ((C_{init} \cup \mathcal{L}(\mathcal{D}(I_1) \cup \cdots \cup \mathcal{D}(I_i))) \setminus (\alpha_{1..i} \cup \beta_{1..i}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i}). \qquad \textbf{(EB.5)}$$

$C_{init} \in S$ and every clause in $\bigcup_{j=1}^{i} \mathcal{D}(I_j)$ is a variant of a clause from $S$. Let $C_{init} = B_{r_1}\theta_1 = B^1_{r_1}$, where $1 \le r_1 \le n$ and $\theta_1$ is a variable renaming substitution set. $\theta_1$ renames the variables in $B_{r_1}$ such that $Ran(\theta_1) \cap Vars(\bigcup_{j=1}^{i} \mathcal{D}(I_j)) = \{\}$. The superscript 1 in $B^1_{r_1}$ means that this is the first clause in the derivation $\Delta$. Let

$$\ell = \left\langle B^1_{r_1}, \ldots, B^m_{r_m} \right\rangle, \text{ where } m = 1 + \left| \bigcup_{j=1}^{i} \mathcal{D}(I_j) \right|,$$

be a list version of the multiset $\bigcup_{j=1}^{i} \mathcal{D}(I_j) \cup \{C_{init}\}$ where the clauses are ordered according to their occurrence in the derivation $\Delta$. Every clause $B^j_{r_j}$, where $1 \le j \le m$, is a variant of a clause from $S$. Let $\theta_j$ be a renaming variable substitution of the clause $B_{r_j} \in S$, such that $B^j_{r_j} = B_{r_j}\theta_j$ . The expression for $\ddot{C}(I_i)$ from **EB.5** can now be written as

$$\ddot{C}(I_i) = ((B^1_{r_1} \cup \cdots \cup B^m_{r_m}) \setminus (\alpha_{1..i} \cup \beta_{1..i}))\sigma_{1..i}(\tau_{1..i}\sigma_{1..i}), \qquad \textbf{(EB.6)}$$

where

$m$ is the total number of variant of clause from $S$ used in $\Delta$,

$1 \le r_j \le n$ for $1 \le j \le m$,

$\alpha_{1..i} \subseteq \bigcup_{j=1}^{m} B^j_{r_j}$, and $\beta_{1..i} \subseteq \bigcup_{j=1}^{m} B^j_{r_j}$.

**EB.6** is the required to prove expression. □

# Appendix C

We assume that when an intermediate conclusion is constructed, it is stored in a linear data structure. A linear structure representation is called a **flatterm** representation. We present a lower bound, for the flatterm representation, on the number of operations required to construct an intermediate conclusion. Flatterm representation is used in CARINE and in other ATPs for short lived clauses, including OTTER, THEO, and VAMPIRE.

## C.1 Flatterm Representation

A flatterm representation is a doubly linked list [Sekar et al. 2001] with an additional pointer to the last term symbol of a term as shown in **Figure C-1**.

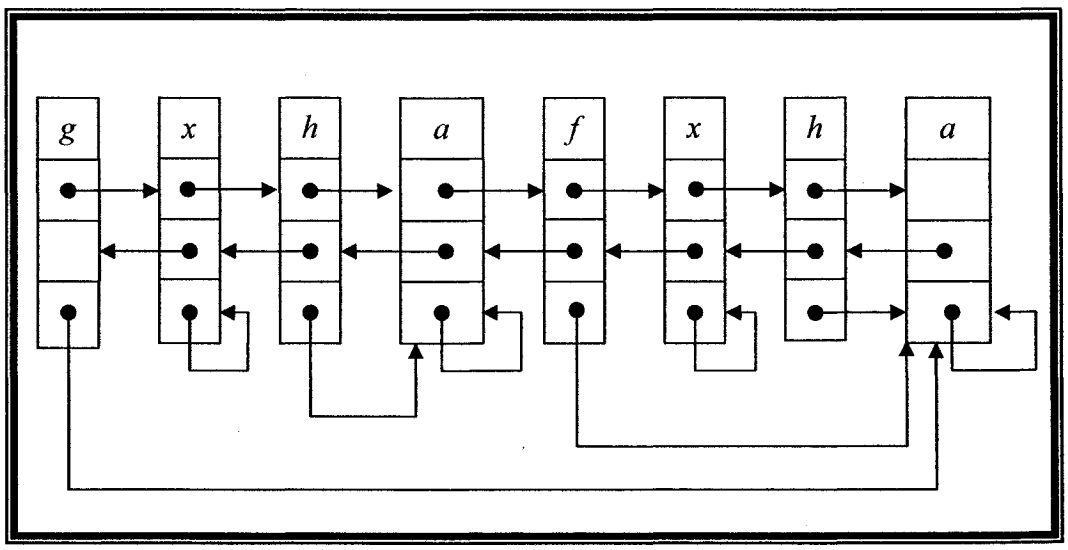



**Figure C-1: Flatterm representation of $g(x,h(a),f(x,h(a)))$**

### C.1.1 Substitution set representation

A substitution set is usually represented as an array of pointers to terms. The indices of the array serve as the hash codes of the variables' identification codes. For example, the set $\sigma = \{x1 \rightarrow f(a), x5 \rightarrow x3, x6 \rightarrow b\}$ can be stored in an array as shown in **Figure C-2**.

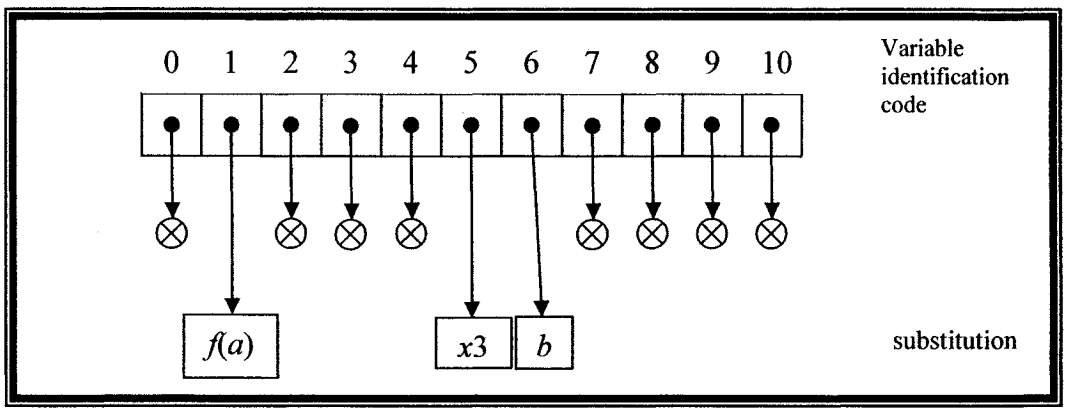



**Figure C-2: Substitution set representation of $\sigma = \{x1 \rightarrow f(a), x5 \rightarrow x3, x6 \rightarrow b\}$ array**

The slots pointing to circle with a cross are assumed to be variables that have no substitutions and the array indices are equal to the identification codes. It is also possible to maintain a substitution set as doubly linked list or as a skip list (for a faster access). Every element in such representation must include the variable identification code and a pointer to the substitution term.

## C.2 Querying under DCC

There are many queries that could be formed to extract valuable information about an inferred clause. Such information can be used to tune the heuristics of an ATP, but more importantly it helps the ATP to evaluate clauses and determine whether they should be discarded or retained. Sometimes discarding a critical

clause may cause the ATP to take a very long time to prove a theorem or it may even never be able to prove such theorem at all. Therefore, it is important to gather and evaluate the necessary information about an inferred clause in order to reduce to a minimum the chance of discarding a critical clause. Depending on the strategies and inference rules implemented in the ATP certain queries may be more useful than others. We discuss only the most common queries that almost all theorem provers exploit.

We use the following notations and definitions that are necessary to demonstrate our analysis:

| | |
|---|---|
| $nt(L) = Weight(L) - 1$ | number of term symbols in the literal $L$ |
| $nt(t) = Weight(t)$ | number of term symbols in the term $t$ |
| $nv(L)$ | number of variables in the literal $L$ |
| $nf(L)$ | number of functions with arity greater than zero in the literal $L$ |
| $nc(L)$ | number of constants in the literal $L$ |
| $ndv(L) = \lvert Vars(L) \rvert$ | number of distinct variables in $L$ |
| $n(L,v)$ | number of occurrences of the variable $v$ in the literal $L$ |
| $\ddot{\sigma}(v)$ or $\sigma(v)$ | the substitution term for the variable $v$ |

$\sigma(v)$ is used instead of $\ddot{\sigma}(v)$ when the substitution term for $v$ is obtained after $\ddot{\sigma}$ is transformed into $\sigma$.

**Example C.1:**

$L = P(x, x, f(x, y, g(h(a, z), b)), y)$

$Vars(L) = \{x, y, z\}$

$\ddot{\sigma} = \{x \rightarrow f(y, a, z), y \rightarrow g(a, b)\}$

$\sigma = \ddot{\sigma} \rightarrow \sigma = \{x \rightarrow f(g(a, b), a, z), y \rightarrow g(a, b)\}$

$nt(L) = 11$, $nv(L) = 6$, $nf(L) = 3$, $nc(L) = 2$,
$ndv(L) = 3$, $n(L,x) = 3$, $n(L,y) = 2$, $n(L,z) = 1$,
$\ddot{\sigma}(x) = f(y,a,z)$, $\sigma(x) = f(g(a,b),a,z)$.

## C.2.1 Computing the weight of a non-constructed clause

One of the most common clause attributes used by ATPs to guide the search is the weight of the inferred clause. Usually, if the weight is greater than some limit, which is either set by the user or calculated automatically by the ATP, the clause is discarded. The weight is also used as part of an ordering relation for clauses and/or literals of the clauses. For example, clauses with lighter weights may have a higher precedence to participate in an inference rule than longer ones when a selection mechanism is exercised. Therefore, it is important to be able to compute such information within DCC in an amount of time that is less than the time it takes to construct a clause. Otherwise, DCC would not provide a significant advantage.

First we establish a formula to compute the weight of a conclusion from the weights and other available data about the premises of an inference rule, and then we compare the time it takes to compute this formula with the time it takes to construct the clause. We also discuss the worst case where it would better to construct the clause (since its takes the same time as computing its weight) and consequently facilitate the execution of complex queries on it rather than be content with simply its weight.

The weight of a clause is the sum of the weights of its literals so it is natural to find out how to compute the weight of a literal without constructing it. This implies we need to compute the weight of every **destination** literal (i.e., a literal from the conclusion) from the weight of the **source** literal (i.e., a literal from the premises).

Using the above definitions, we can compute the number of variables in a literal by summing up the number of occurrences of every distinct variable in this literal as follows,

$$nv(L) = \sum_{i=1}^{ndv(L)} n(L, v_i). \qquad \textbf{(EC.1)}$$

The number of terms in a literal is the sum of the number of variables, functions and constants,

$$nt(L) = nv(L) + nf(L) + nc(L). \qquad \textbf{(EC.2)}$$

By substituting $nv(L)$ with the right hand side of **EC.1** we get

$$nt(L) = \sum_{i=1}^{ndv(L)} n(L, v_i) + nf(L) + nc(L). \qquad \textbf{(EC.3)}$$

Assume that the number of occurrences of a variable $v$ in $L$ is equal to one. If $v$ is replaced by a substitution which is a function, then the number of terms in the new literal, $L'$, increases by the weight of this function minus one. We can write this simple formula as: $nt(L') = nt(L) + nt(\sigma(v)) - 1$. Notice that if the substitution is a variable or a constant, then $nt(\sigma(v)) = 1$ and thus, $nt(L') = nt(L)$. Therefore, the number of terms would remain unchanged.

**Example C.2:**

$L = P(x, y, y)$, $nt(L) = 3$, $n(L, x) = 1$, $\sigma = \{x \to f(a)\}$,

$L' = L\sigma = P(f(a), y, y)$.

The substitution for $x$ is a function since $\sigma(x) = f(a)$, $nt(\sigma(x)) = 2$. Therefore, $nt(L') = nt(L) + nt(\sigma(x)) - 1 = 3 + 2 - 1 = 4$.

If the number of occurrences of some variable $v_j$ in $L$ is greater than one, where $1 \le j \le ndv(L)$, then the number of terms in $L'$ becomes

$$\begin{aligned} nt(L') &= nt(L) + n(L, v_j) \cdot nt(\sigma(v_j)) - n(L, v_j) \\ &= nt(L) + n(L, v_j) \cdot (nt(\sigma(v_j)) - 1) \end{aligned}$$

If *every* distinct variable in $L$ has a substitution in $\sigma$, then the number of terms in $L'$ is computed by

$$nt(L') = nt(L) + \sum_{i=1}^{ndv(L)} \left( n(L, v_i) \cdot (nt(\sigma(v_i)) - 1) \right) \qquad \textbf{(EC.4)}$$

with $nt(\sigma(v_i)) \ge 1$.

However, since it is not necessary for all the distinct variables in $L$ to have a substitution in $\sigma$, we can write the above formula as

$$nt(L') = nt(L) + \sum_{i=1}^{|Dom_L(\sigma)|} \left( n(L, v_i) \cdot (nt(\sigma(v_i)) - 1) \right), \qquad \textbf{(EC.5)}$$

where $|Dom_L(\sigma)|$ is the cardinality of the set of variables in $L$ that have a substitution in $\sigma$, i.e., $Dom_L(\sigma) = Vars(L) \cap Dom(\sigma)$.

**Example C.3**

$L = P(x, y, f(a, g(z, x), y, y)$

$\sigma = \{x \to f(w, a, b, w), y \to c, u \to w\}$

$nt(L) = 9$, $nv(L) = 6$, $ndv(L) = 3$, $n(L, x) = 2$, $n(L, y) = 3$,

$\sigma(x) = f(w, a, b, w)$, $\sigma(y) = c$,

$nt(\sigma(x)) = 5$, $nt(\sigma(y)) = 1$,

$Dom_L(\sigma) = \{x, y\}$, $|Dom_L(\sigma)| = 2$

If we construct $L'$, we get

$$L' = L\sigma = P(f(w,a,b,w),c,f(a,(g(z,f(w,a,b,w)),c,c))$$

and the number of terms would be 17. Now, if we apply the formula **EC.5**, we get

$$\begin{aligned} nt(L') &= 9 + n(L,x)\cdot(nt(\sigma(x))-1) + n(L,y)\cdot(nt(\sigma(y))-1) \\ &= 9 + 2\cdot(5-1) + 3\cdot(1-1) \\ &= 17 \end{aligned}$$

Since the number of terms in a literal $L$ does not change when the substitution of a variable from $L$ is a variable or a constant, we can apply the summation only on the variables that have a substitution which is a function. If we denote the set of variables $v_i$ in $Dom_L(\sigma)$ that have substitutions which are functions (i.e. $nt(\sigma(v_i)) > 1$) by $\Gamma_L(\sigma)$, then **EC.5** can be written as

$$nt(L') = nt(L) + \sum_{i=1}^{|\Gamma_L(\sigma)|} \left(n(L,v_i)\cdot(nt(\sigma(v_i))-1)\right), \qquad \textbf{(EC.6)}$$

where $v_i \in \Gamma_L(\sigma)$. Notice that $\Gamma_L(\sigma) \subseteq Dom_L(\sigma)$ so $|\Gamma_L(\sigma)| \leq |Dom_L(\sigma)|$.

If we apply **EC.6** to our above example we get

$$\begin{aligned} nt(L') &= 9 + n(L,x)\cdot(nt(\sigma(x))-1) \\ &= 9 + 2\cdot(5-1) \\ &= 17. \end{aligned}$$

Here, $\Gamma_L(\sigma) = \{x\}$ because $x$ is the only variable in $Dom_L(\sigma)$ that has a substitution which is a function, i.e., $nt(\sigma(x)) = 5 > 1$.

We conclude that computing the number of terms in any literal of the inferred clause is linear in the number of distinct variables that have a substitution term whose weight is greater than one and thus, takes no more than $O(|\Gamma_L(\sigma)|)$ operations.

Let the literals that are not deleted, resolved away, or factored out from the premises of the inference rule be labeled $L_1,\ldots,L_n$ and $P=\{L_1,\ldots,L_n\}$, computing the weight of the conclusion $C$ of length $n$ would then take

$$\sum_{i=1}^{n}\left|\Gamma_{L_i}(\sigma)\right| \le n \cdot \underset{L\in P}{Max}\{\left|\Gamma_L(\sigma)\right|\}, \qquad \textbf{(EC.7)}$$

operations. However, there is an additional hidden cost that is not taken into account. This cost is the time to maintain the weights of the substitution terms.

We have assumed, so far, the number of terms in every substitution for every variable in $\sigma$ can be obtained in constant time. In other words, $nt(\sigma(v_i))$ must be stored within a table that can be accessed in $O(1)$ after the unification process is complete. The additional time to maintain such information within the table is hidden within the unification process. In order to fairly evaluate the performance of DCC when querying the non-constructed clauses for their weights, we need to investigate the extent of the effect of maintaining a table of the weights of the substitution terms. In other words, we need to determine the amount of time consumed by the process which maintains such information. For one resolution the process is quite simple. However, it becomes complicated when a sequence of resolutions is performed because of the variables' dependencies. A directed graph of the dependencies must be constructed and updated after every unification.

## C.2.2 Variable dependencies

In DCC p-idempotent substitution set are used and so variables may very likely be bound to a substitution term which contains one or more variables. For example, if $\ddot{\sigma}=\{x\rightarrow y, y\rightarrow f(w,g(z)), z\rightarrow a\}$, then $x$ depends on $y$, and $y$ depends on $w$ and $z$. We call $x$ and $y$ the *dependent* variables and $w$ and $z$ the *independent* variables. A **independent variable** in a p-idempotent substitution set either has no substitution or its substitution contains no variables. All other variables are considered dependent. Notice that $x$ is *directly* dependent on $y$ and *indirectly*

dependent on $w$ and $z$. The **depth of dependency** of a variable $v$ from the domain of a p-idempotent substitution set is the *longest* path in the dependency directed graph from the variable $v$ to an independent variable. We denote the depth of dependency of a variable $v$ within a p-idempotent substitution set $\ddot{\sigma}$ by the function $DD_{\ddot{\sigma}}(v)$. If $\ddot{\sigma}$ is implied within the context, then we may drop the subscript and simply write $DD(v)$. It is clear from the definition that the depth of dependency of an independent variable is zero. In the above example, $DD_{\ddot{\sigma}}(x)=2$, $DD_{\ddot{\sigma}}(y)=1$, $DD_{\ddot{\sigma}}(w)=0$ and $DD_{\ddot{\sigma}}(z)=0$. The **depth of dependency of a p-idempotent substitution set** is the maximum depth of dependency of any of its variables. Formally, the depth of dependency of a substitution set $\ddot{\sigma}$ is defined as

$$DD(\ddot{\sigma}) = \underset{v \in Dom(\ddot{\sigma})}{Max} \{DD_{\ddot{\sigma}}(v)\}.$$

The depth of dependency is not fixed in DCC. It may change as the derivation sequence gets longer as demonstrated in the following example.

**Example C.4:**

Given the clauses:

$$B_1 = \{D(x_1, f(x_2, x_5, x_3, x_4), x_3, x_4), \neg P(x_6), \neg Q(x_1, x_6)\}$$

$$B_2 = \{\neg D(x_{11}, x_{12}, x_{13}, x_{13})\}$$

$$B_3 = \{Q(g(x_{21}), x_{22})\}$$

consider the following sequence of two resolutions ($\mathcal{R}$ means "the resolution of"):

$$C_1 = \mathcal{R}(B_1, B_2) = \{\neg P(x_6), \neg Q(x_{11}, x_6)\} \text{ with}$$

$$\ddot{\sigma}_1 = \left\{ x_1 \underset{1}{\overset{1}{\rightarrow}} x_{11}, x_{12} \underset{2}{\overset{5}{\rightarrow}} f(x_2, x_5, x_3, x_4), x_3 \underset{1}{\overset{1}{\rightarrow}} x_{13}, x_4 \underset{1}{\overset{1}{\rightarrow}} x_{13} \right\}$$

and

$$C_2 = \mathcal{R}(B_3, C_1) = \{\neg P(x_{22})\} \text{ with}$$

$$\ddot{\sigma}_2 = \left\{ x_{11} \xrightarrow[1]{2} g(x_{21}), x_6 \xrightarrow[1]{1} x_{22} \right\}$$

The arrows are labeled with the weight of substitution term on the top and the depth of dependency on the bottom. The weight is calculated after the p-idempotent substitution is transformed into an idempotent substitution. For example, $nt(\sigma(x_{12})) = 5$, where $\sigma$ is obtained from $\ddot{\sigma} \rightarrow \sigma$. $DD_{\ddot{\sigma}_1}(x_{12}) = 2$.

Since in DCC we are not really constructing the clauses $C_1$ and $C_2$, we have to combine the substitution sets $\ddot{\sigma}_1$ and $\ddot{\sigma}_2$ to keep track of the changes of the variables' substitutions. When we perform the union of the two sets, we get

$$\ddot{\sigma}_2' = \ddot{\sigma}_1 \cup \ddot{\sigma}_2$$

$$= \left\{ x_1 \xrightarrow[1]{1} x_{11}, x_{12} \xrightarrow[2]{5} f(x_2, x_5, x_3, x_4), x_3 \xrightarrow[1]{1} x_{13}, x_4 \xrightarrow[1]{1} x_{13}, x_{11} \xrightarrow[1]{2} g(x_{21}), x_6 \xrightarrow[1]{1} x_{22} \right\}.$$

We notice that neither the weights of the substitution terms (i.e. numbers above the arrows) nor the depths of the dependencies (i.e. number on the bottom of the arrows) reflect the correct values. For example, since $x_{11}$ changed, $nt(\sigma_2'(x_1))$, where $\ddot{\sigma}_2' \rightarrow \sigma_2'$, should be 2 instead of 1 and $DD_{\ddot{\sigma}_2'}(x_{11})$ should be 2 instead of 1. The concept of depth of dependency is only needed for our analysis of the worst case scenario and is not actually necessary to be implemented within an ATP and thus, we will not concern ourselves with the amount of time required to update its values.

In order to maintain a valid reference to the weight of the substitution of a variable, we have to update the weight of the substitution of every dependent

variable related to an independent variable once the independent variable is bound to a substitution term which is a function, as in the case of $x_{11}$. Furthermore, the constructed directed graph of the dependencies should be updated after every successful unification whenever an independent variable changes its state to become dependent.

**Figure C-3** shows the graph of the dependencies of the variables from the set $\ddot{o}_2''$. Notice that the directed graph never has any cycles since the substitution set is p-idempotent and hence has no circular references.

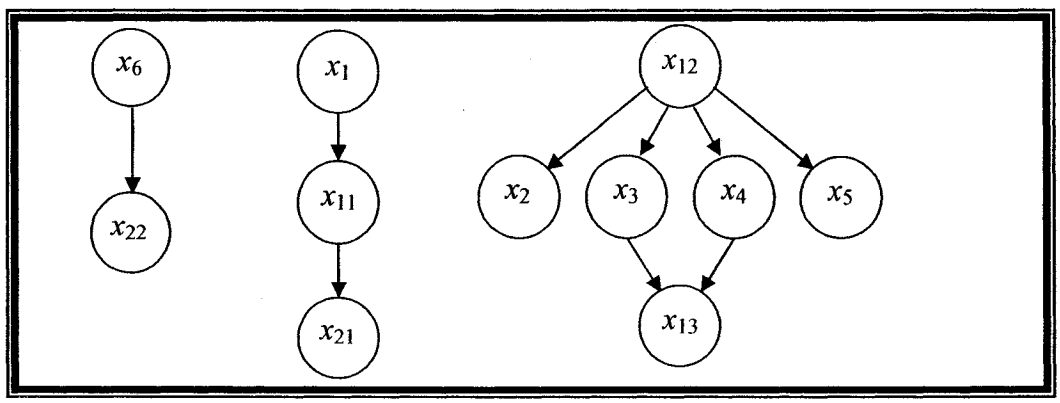



**Figure C-3: Graph representation of variable dependencies.**

The variable dependency graph is represented as an adjacency list. The list contains all the distinct variables of the premises. With each variable $x$, a linked list is attached containing the variables that depend directly on $x$. **Figure C-4** shows the adjacency list of $\ddot{o}_2''$. Notice that the arrows in the adjacency list are reversed as opposed to the graph.

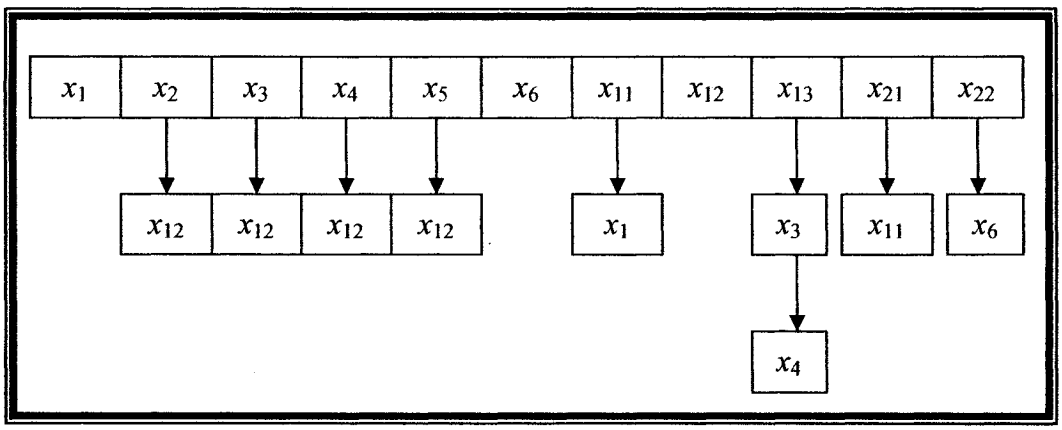



**Figure C-4: Adjacency list of variable dependencies**

If a variable dependency graph is built to maintain information about the weights of the substitution terms, then it is important to calculate the time needed to update the graph and, consequently, the weights of the substitution terms in the worst case scenario so that we may compare the total time needed to compute the weight of a clause with the time needed to construct the clause.

Suppose the substitution set is ordered by dependency, such that if the variable $v_i$ depends on the variable $v_j$, then $i < j$ and $v_i$ is listed before variable $v_j$. This is not necessarily a *total order* relation since if $i < j$ then it does not imply that $v_i$ depends on $v_j$. For example, in $\ddot{\sigma}_1$ the variable $x_1$ depends on $x_{11}$ but not on any of the other variables, so it can be listed anywhere as long as it is before $x_{11}$. A total order occurs when every variable depends on all the variables that follow it. For example, over $\ddot{\sigma} = \{x \to y, y \to z, z \to f(w), w \to g(u,v)\}$ the variable dependency relation is a total order. A total order is necessary albeit not sufficient for the worst case to occur.

When input clauses in a theorem contain functions and such input clauses are involved in the derivation sequence, there is a high probability that the domain of

the substitution set contains variables that are dependent on other variables whose substitution terms are functions. The worst case occurs when all the distinct variables of the input clauses involved in a derivation sequence belong to the domain of the substitution set, and the substitution set is totally ordered by the variable dependency relation such that every variable is directly dependent on all the variables that follow it within the substitution set. Formally, given a set of input clauses $S = \{B_1, \ldots, B_n\}$, let $\prec$ denotes the ordering dependency relation, such that $v_i \prec v_j$ implies that $v_i$ depends (directly or indirectly) on $v_j$. The worst case scenario occurs at depth $d$ in a derivation when the following three conditions are met:

(1) $\bigcup_{j=1}^{d} Vars(B_{i_j}^j) = Dom(\ddot{\sigma})$, where $B_{i_j}^j = B_{i_j}\theta_j$ for all $i_j \in \{1, \ldots, n\}$, is a variant of an input clause in $S$, and $\theta_j$ is a variable renaming substitution such that $\forall j,k \;\; j \neq k$, $Ran(\theta_j) \cap Ran(\theta_k) = \{\}$, and

(2) $\forall v_i, v_j \in Dom(\ddot{\sigma})$ where $i \neq j$, if $i < j$ then $v_i \prec v_j$, and

(3) $\forall (v_i \rightarrow t_i) \in \ddot{\sigma}, Dom_{t_i}(\ddot{\sigma}) = \{v_j : j > i\}$.

The following example demonstrates the worst case scenario.

**Example C.5:**

Suppose that after the application of an inference rule leading to depth $d-1$ in DCC the accumulating substitution set is

$$\ddot{\sigma}_{d-1} = \left\{ x \overset{13}{\rightarrow} f(y,z,w,u), y \overset{7}{\rightarrow} f(z,u,w,w), z \overset{3}{\rightarrow} g(w,u), w \overset{1}{\rightarrow} u \right\}$$

and all the distinct variables of the input clauses involved in this derivation sequence are $x$, $y$, $z$, $w$, and $u$. Now suppose that an inference rule is applied extending the derivation length to $d$, such that no additional variables appear in

the premises of this inference except the independent variable $u$. Suppose that $u$ binds with the substitution term $h(a)$. The resulting substitution set $\ddot{\sigma}_d$ is

$$\ddot{\sigma}_d = \left\{ x \overset{13}{\rightarrow} f(y,z,w,u), y \overset{7}{\rightarrow} f(z,u,w,w), z \overset{3}{\rightarrow} g(w,u), w \overset{1}{\rightarrow} u, u \rightarrow h(a) \right\}.$$

The weights of the substitution terms are no longer correct and they require updating. $\ddot{\sigma}_d$ reflects the worst case scenario since all the above three conditions are met. All the variables in the input clauses involved in the derivation sequence are in $\ddot{\sigma}_d$ (condition 1), the set is totally ordered; $x \prec y \prec z \prec w \prec u$ (condition 2), and every variable is dependent on all the variables that follow it; $Dom_{\ddot{\sigma}_d(x)}(\ddot{\sigma}_d) = \{y,z,w,u\}$, $Dom_{\ddot{\sigma}_d(y)}(\ddot{\sigma}_d) = \{z,w,u\}$, and $Dom_{\ddot{\sigma}_d(w)}(\ddot{\sigma}_d) = \{u\}$ (condition 3). To update all the weights of the substitution terms, we proceed first with the last dependent variable, $w$, in $\ddot{\sigma}_d$ and walk our way back to the first element in $\ddot{\sigma}_d$. Notice that we only need to update the variables that are directly dependent on the variable we are working with. Since all the variables depend directly on $u$, we have to update them all. This requires $|\ddot{\sigma}_d| - 1$ updates. Following that, we have to update all the variables that depend directly on $w$. This requires $|\ddot{\sigma}_d| - 2$ updates. We continue updating all the references to the weights of the substitution terms until we reach $x$. The total number of updates is $|\ddot{\sigma}_d| - 1 + |\ddot{\sigma}_d| - 2 + \ldots + |\ddot{\sigma}_d| - |\ddot{\sigma}_d| + 1 = 4+3+2+1 = 10$ updates.

From our above example, we conclude that we need $|\ddot{\sigma}_d| \cdot (|\ddot{\sigma}_d| - 1)/2$ operations to update the weights of the substitution terms in the worst case. Therefore, whenever the worst case occurs, $O(|\ddot{\sigma}|^2)$ operations are required to update the references to the weights of the substitution terms of the substitution set $\ddot{\sigma}$. Let $\ddot{\sigma} \rightarrow \sigma$ then $|\ddot{\sigma}| = |\sigma|$ and $O(|\ddot{\sigma}|^2) = O(|\sigma|^2)$, since $|\sigma| = |Dom(\sigma)|$ (see Definition

2.21) and from the properties of p-idempotent substitution set, $Dom(\ddot{\sigma}) = Dom(\sigma)$ (see Definition 3.5). Adding this hidden cost to **EC.7**, we obtain the following upper bound on the total number of operations needed to compute the weight of a derived non-constructed clause

$$n \cdot \underset{L \in P}{Max}\{|\Gamma_L(\ddot{\sigma})|\} + O(|\sigma|^2). \qquad \textbf{(EC.8)}$$

Since we are dealing with the worst case, the first condition dictates that the size of the substitution set is equal to the total number of distinct variables within the input clauses involved in the derivation sequence and the third condition implies that each of those variables has a substitution which is a function and therefore, we can write

$$\begin{aligned} n \cdot \underset{L \in P}{Max}\{|\Gamma_L(\ddot{\sigma})|\} + O(|\sigma^2|) &\le n \cdot |\sigma| + O(|\sigma^2|) \\ &= O(Len(C) \cdot |\sigma| + |\sigma|^2). \end{aligned} \qquad \textbf{(EC.9)}$$

In order for the DCC to be effective, the time to compute the weight of $C$ as formulated in **EC.9** must be less than the time to construct the clause $C$ which is $O(Weight(C))$. In other words, $Len(C) \cdot |\sigma| + |\sigma|^2$ must be at most $Weight(C)$. We need $Len(C) \cdot |\sigma| + |\sigma|^2 \le Weight(C)$ or equivalently[1]

$$|\sigma| \le \frac{-Len(C) + \sqrt{Len(C)^2 + 4 \cdot Weight(C)}}{2} \qquad \textbf{(EC.10)}$$

Notice that if the size of the substitution set is larger than the value returned from the evaluation of the right hand side of the above inequality, then there may or may not be a worst case scenario. For example, if $C_1 = P(x,a) \vee Q(x)$ is resolved

---

[1] This is a quadratic inequality of the form $Ax^2 + Bx + C \le 0$ whose roots are $x = (-B \mp \sqrt{B^2 - 4AC}\,/\,2A$. Of course, the negative square root is useless to us here.

with $C_2 = \neg P(y, y)$ to produce $C = Q(a)$ with substitution set $\ddot{\sigma} = \{x \to y, y \to a\}$, then the size of the substitution set is larger than the right hand side of **EC.10**, i.e., $|\sigma| = 2 > (-1 + \sqrt{1 + 4 \times 2})/2 = 1$, where $\ddot{\sigma} \to \sigma$. Here we have the worst case scenario because all three conditions are met. However, we also have the best case scenario and the reason is that $\ddot{\sigma}$ contains only variables located at depth one within the input clauses, $C_1$ and $C_2$. If $Dom(\ddot{\sigma})$ contains only variables located at depth one within the input clauses involved in the derivation sequence of a DCC, then the size of $\ddot{\sigma}$ does not affect the time it takes to update the references to the weights of the substitution terms. In fact, there would be no updating at all since all variables in $Dom(\ddot{\sigma})$ would be substituted by either constants or variables whose weights are always one, thus, $\Gamma_L(\ddot{\sigma})$ would be empty. The time to compute the weight of the inferred clause $C$ in such cases is simply the time to sum up the weights of all the literals remaining from the premises. The best case requires only $O(Len(C))$ operations to compute the weight of the inferred clause. Over 630 theorems from the TPTP library contain clauses with no functions in them. Those theorems are common in the GRP (group theory), NLP (natural language processing), PUZ (puzzle), SYN (syntactic) categories. On the other hand, if the size of the substitution set is less than the right hand side of **EC.10**, then we are guaranteed that the worst case has not occurred and thus, it is faster to compute the weight of a clause by the formula **EC.11** (below) rather than construct the clause.

Suppose that DCC is used and $C = \{L_1', \ldots, L_n'\}$ is a non-constructed intermediate conclusion and $\ddot{\sigma}$ the resulting p-idempotent substitution set, such that for $1 \le i \le n$, $L_i' = L_i\sigma$ where $L_1, \ldots, L_n$ are the literals that are not deleted, resolved away, or factored out from the premises and $\ddot{\sigma} \to \sigma$. The weight of $C$ is computed by the formula

$$Len(C) = \sum_{i=1}^{n} nt(L_i')$$
$$= \sum_{i=1}^{n} \left( nt(L_i) + \sum_{j=1}^{|\Gamma_{L_i}(\sigma)|} \left( n(L_i, v_j) \cdot (nt(\sigma(v_j)) - 1) \right) \right) \qquad \textbf{(EC.11)}$$

A quick and easy way to test for the possible existence of the worst case is to compare the depth of dependency of a substitution set with its size. A necessary condition but not sufficient condition for the worst case to occur is that the depth of variable dependency should be equal or greater than the size of the substitution set. In other words, if $DD(\ddot{\sigma}) < |\ddot{\sigma}| - 1$ then we can be sure that the worst case did not occur and we can continue using DCC; otherwise we have to perform further tests before deciding whether to proceed with DCC or not. We ran our ATP on 2323 theorems and computed $|\ddot{\sigma}|$, as well as the right hand side (rhs) of **EC.10** after every binary resolution and binary factoring. We counted the number of times the size of $\ddot{\sigma}$ came out at least as large as the rhs taking into consideration that all the substitution terms are functions and that $DD(\ddot{\sigma}) \geq |\ddot{\sigma}| - 1$. The results revealed that it may (but not definitely) have been better to construct the clauses rather than simply compute their weights for only 0.39% on average of all the generated clauses. The highest percentage was 71.89% and this high percentage occurred for only one theorem, PLA002-1. There were four theorems (ANA003-2, ANA004-2, ANA005-2, ANA003-4) with a percentage between 25% and 28%, one theorem (ANA004-4) with 13.82% and the rest (from almost all the categories within the TPTP set) were between 0% and 5.41%. **Figure C-5** shows the number of clauses where it may (but not for sure) have been faster to construct the clauses and determine their weights while constructing them rather than compute their weights and delay their construction.

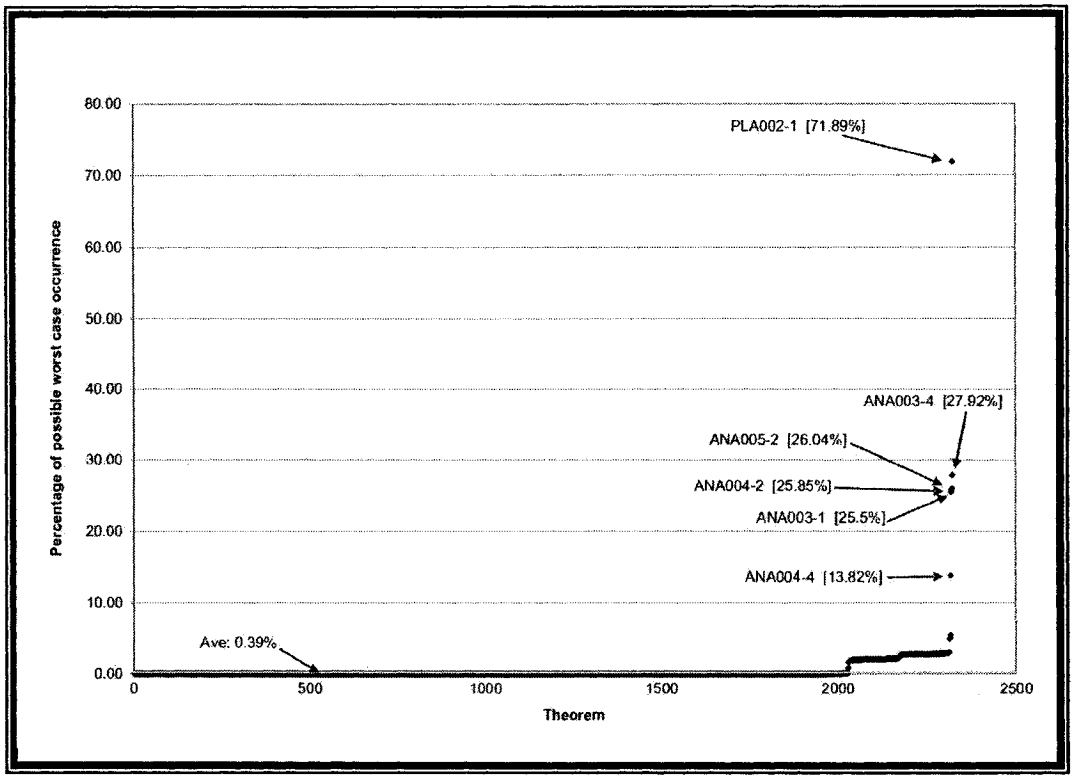



**Figure C-5: Number of cases where it may have been better to construct a clause rather than simply compute its weight.**

We conclude that it is much faster to use DCC and calculate the weight of the generated clause without constructing it using **EC.11** even though it may be occasionally (less than 0.5% on average) better to construct a clause in order to determine its weight.

## C.2.3 Computing the maximum literal depth

Many ATPs abandon a search path by discarding a clause that contains a literal whose maximum term depth is above a certain threshold. Our test results from THEO [Newborn 2001] have shown that the theorems that THEO was able to prove contained no clause within the proof with a literal whose maximum literal depth is greater than the maximum literal depth plus two of any literal from the input clauses. In other words, we could have allowed the maximum literal depth for any literal of any derived clause to be at most equal to the maximum literal depth plus

two of any literal from the input clauses and still got all the proofs. Therefore, it may be helpful to determine the maximum literal depth of the literals in a generated clause if the ATP makes use of this information.

Suppose one of the premises of an inference contains the literal $L$ such that $L' = L\sigma$ is a literal of the conclusion and $\sigma$ is the mgu. We want to compute the maximum literal depth of $L'$ without constructing it. From the definition of the maximum literal depth (see **Definition 2.14**), we realize that we need to compare the maximum term depth of every argument in $L'$. Since $L'$ is not constructed, we have to compare the maximum term depths of the arguments in $L$ after we compute their updated maximum term depths.

Notice that only the maximum term depths of the arguments in $L$ that contain variables that gained a substitution which is a function need to be updated in order to maintain the correct value of maximum literal depth. For example, if $L = P(x, f(y))$ and $\sigma = \{x \rightarrow a, y \rightarrow w\}$, then $L' = L\sigma = P(a, f(w))$ and thus, the maximum literal depth remains the same, i.e. $MaxDepth(L) = MaxDepth(L') = 2$, since none of the variables $x$ and $y$ have been bounded to terms that are functions. On the other hand, for $L = P(x, y)$ and $\sigma = \{x \rightarrow f(w), y \rightarrow w\}$, $L' = L\sigma = P(f(w), w)$ and $MaxDepth(L') = 2$ which is different from $MaxDepth(L) = 1$, since $x$ is bounded to a term which is a function. Furthermore, if a variable occurs more than once in the literal, then we only need to compare the updated depth of the deepest occurrence of such variable. For example, in $L = P(f(f(x)), x, g(u))$ the variable $x$ occurs twice. The first occurrence, which is an argument of a function, has a term depth greater than the second one and so we only need to compare the term depth of the deepest term within the substitution of $x$ with the maximum literal depth.

If $\pi$ is a position string, then the length of the string $\pi$, denoted by $Len(\pi)$, is the number of integers separated by dots. For example, if $\pi = 2.1.1$ then $Len(\pi) = 3$. We define $td_{\max}(L,t)$ to be the function that returns the deepest occurrence of a term $t$ in a literal $L$. Formally, if $t \in^{o} L$, then

$$td_{\max}(L,t) = Max\{Len(\pi_1),\ldots,Len(\pi_k)\}$$

where $\pi_1,\ldots,\pi_k$ are positions of $t$ in $L$; i.e., for all $1 \le i \le k$, $L|_{\pi_i} = t$.

**Example C.6:**

$L = P(x, g(f(y,z)), g(g(x)))$

$td_{\max}(L,x) = 3, \quad td_{\max}(L,g) = 2, \quad td_{\max}(L,f) = 2$

The updated maximum literal depth of the literal $L'$ is computed by the formula

$$MaxDepth(L') = Max(MaxDepth(L), \underset{v \in \Gamma_L(\sigma)}{Max}\{td_{\max}(L,v) + MaxDepth(\sigma(v)) - 1\}) \qquad \textbf{(EC.12)}$$

We define $Max\{\} = 0$, so if no variable is bounded to a function term, then $\Gamma_L(\sigma) = \{\}$ and

$$\begin{aligned} MaxDepth(L') &= Max(MaxDepth(L), Max\{\}) \\ &= Max(MaxDepth(L), 0) \\ &= MaxDepth(L) \end{aligned}$$

The number of comparisons required to compute $MaxDepth(L')$ is $1 + |\Gamma_L(\sigma)| - 1$, because we need one comparison to obtain the maximum between $MaxDepth(L)$ and the result of $\underset{v \in \Gamma_L(\sigma)}{Max}\{td_{\max}(L,v) + MaxDepth(\sigma(v)) - 1\})$, and $|\Gamma_L(\sigma)| - 1$

comparisons between the values in $\underset{v\in\Gamma_L(\sigma)}{Max}\{td_{\max}(L,v)+MaxDepth(\sigma(v))-1\})$. Therefore, to compute $MaxDepth(L')$ we have to perform $|\Gamma_L(\sigma)|$ comparisons.

Similar to computing the weight of a clause, there is an additional hidden cost which is the updating of the maximum term depth of the substitution terms. The same issue discussed about variable dependency exists here. Therefore, when the worst case occurs as described in the previous section, the upper bound on the number of operations required to update the maximum depth of the substitution terms is $O(|\sigma|^2)$. The total number of operations required to compute $MaxDepth(L')$ becomes $|\Gamma_L(\sigma)|+O(|\sigma|^2)$. Since we are dealing with the worst case, then $|\Gamma_L(\sigma)|=|\sigma|$ because all the distinct variables in $L$ are bounded to substitution terms which are functions. Therefore, the total number of operations required to compute **EC.12** in the worst case can be written as

$$|\Gamma_L(\sigma)|+O(|\sigma|^2)=O(|\sigma|+|\sigma|^2) \qquad \textbf{(EC.13)}$$

To compute the maximum term depth of any term within the non-constructed conclusion $C$, we find the maximum of all the maximum literal depths of the literals in $C$, i.e., $\underset{L'\in C}{Max}\{MaxDepth(L')\}$. This implies that we need to perform $Len(C)\cdot O(|\sigma|+|\sigma|^2)$ operations. However, the maximum term depth update of the substitution terms need only be done once, Therefore, the total number of operations needed to compute the maximum term depth between all the arguments of all the literals of the clause $C$ is

$$O(Len(C)\cdot|\sigma|+|\sigma|^2) \qquad \textbf{(EC.14)}$$

We notice that the amount of operations in **EC.14** is exactly the same as in **EC.9** and hence, we use the same reasoning to determine the limit at which it would be better to construct the clause rather than simply compute the maximum of maximum literal depths of its literals.

### C.2.4 Computing other queries

There are a large number of interesting queries that ATPs perform to obtain information useful for guiding the search through the selection of certain strategies and also to determine candidate clauses for particular inference rules especially subsumption. We can't go through them all. However, we mention a couple of intuitive ideas that may help to analyze the performance of DCC over a given query.

*Term indexing* [Sekar et al. 2001] and *feature vector indexing* [Schulz 2004] play an important role in retrieving subsets of retained clauses that are candidates for subsumption testing. Even though we will not delve into such a broad subject here, it is worth pointing out that indexing relies on the existence of the clauses in some implemented data structure that is best suited for the ATP. The data structure is usually a variation of a *trie*[1]. Since DCC does not construct clauses, term indexing is quite tricky because it has to be performed on substitution sets rather than actual clauses. Feature vector indexing which performs indexing on a clause characteristics, referred to as features, such as the number of negative and positive literals, the number of occurrences of symbols in a clause, and the maximum literal depth, rather than the actual terms within the clause is easier to implement than term indexing when using DCC. Most of the clause features can be calculated efficiently using a similar reasoning as the ones provided in previous sections: **C.2.1**, **C.2.2**, and **C.2.3**.

[1] Trie comes from the word retrieval. Pronounced like "try". It is a variant of the tree data structure and is generally very efficient for retrieving information.

# Appendix D

A list of the 100 theorems selected from all the domains of the TPTP v2.6.0 library that were used for the experiments whose results are discussed in Chapter 6. The percentage of time spent in clause construction, *PTCC*(*t*), the ratio of the percentage of successful unifications, *RPSU*(*t*), the ratio of the number of unit conflict tests, *RUCT*(*t*), and the inference rate speedup, *IRS* (*t*), are listed for each of the theorems.

| Theorem | *PTCC*(*t*) [%] | *RPSU*(*t*) | *RUCT*(t) | *IRS*(*t*) |
|---|---|---|---|---|
| ALG001-1 | 25.29 | 1.03 | 1.00 | 1.72 |
| ALG003-1 | 0.59 | 1.04 | 1.36 | 0.97 |
| ALG008-1 | 17.11 | 0.93 | 1.03 | 1.41 |
| ALG010-1 | 36.64 | 0.83 | 1.00 | 2.10 |
| ANA001-1 | 4.90 | 0.95 | 1.00 | 1.11 |
| ANA002-4 | 45.06 | 0.98 | 0.00 | 3.11 |
| ANA003-4 | 60.37 | 1.50 | 1.00 | 6.79 |
| ANA004-2 | 7.07 | 0.45 | 1.00 | 1.00 |
| ANA006-2 | 39.89 | 0.88 | 0.00 | 2.37 |
| BOO001-1 | 10.49 | 0.43 | 0.98 | 2.49 |
| BOO008-3 | 31.13 | 1.25 | 1.23 | 2.19 |
| BOO014-2 | 49.63 | 0.59 | 1.00 | 3.36 |
| BOO014-3 | 46.07 | 1.57 | 1.00 | 6.30 |
| BOO019-1 | 9.60 | 0.42 | 1.00 | 2.48 |
| BOO038-1 | 15.64 | 0.74 | 0.00 | 1.12 |
| CAT015-3 | 55.35 | 0.69 | 0.00 | 5.49 |
| CAT020-4 | 37.56 | 0.92 | 0.00 | 2.25 |
| COL001-2 | 0.06 | 1.00 | 1.00 | 1.10 |
| COL003-13 | 27.32 | 0.90 | 1.00 | 1.54 |
| COL065-1 | 53.63 | 0.19 | 1.00 | 1.16 |
| COL078-1 | 54.07 | 0.25 | 1.00 | 1.59 |
| COM003-1 | 46.16 | 0.95 | 0.00 | 3.01 |
| COM004-1 | 22.46 | 1.09 | 1.00 | 1.63 |
| FLD003-1 | 65.57 | 1.01 | 1.00 | 8.66 |
| FLD015-1 | 63.83 | 1.01 | 1.00 | 9.86 |
| FLD042-3 | 29.62 | 2.06 | 1.00 | 2.38 |

| Theorem | *PTCC(t)* [%] | *RPSU(t)* | *RUCT*(t) | *IRS(t)* |
|---|---|---|---|---|
| FLD043-5 | 38.67 | 1.21 | 1.00 | 2.34 |
| FLD080-4 | 36.16 | 1.81 | 1.00 | 2.34 |
| GEO004-1 | 25.92 | 0.82 | 3.48 | 1.38 |
| GEO009-2 | 37.53 | 1.00 | 1.09 | 2.76 |
| GEO074-2 | 29.29 | 1.09 | 1.01 | 3.03 |
| GEO089-1 | 32.82 | 0.44 | 3.40 | 1.40 |
| GEO160-1 | 36.02 | 1.20 | 1.00 | 2.33 |
| GRA001-1 | 43.85 | 1.00 | 0.00 | 2.74 |
| GRP196-1 | 4.87 | 0.90 | 0.00 | 1.15 |
| GRP207-1 | 12.99 | 0.75 | 1.00 | 1.06 |
| GRP252-1 | 40.62 | 0.81 | 0.00 | 2.68 |
| GRP392-1 | 36.56 | 0.78 | 0.00 | 2.14 |
| GRP506-1 | 14.05 | 0.73 | 1.00 | 1.06 |
| HEN004-2 | 45.51 | 1.00 | 1.23 | 4.78 |
| HWC003-2 | 46.15 | 1.00 | 0.00 | 3.23 |
| HWV022-2 | 32.06 | 1.08 | 1.00 | 1.99 |
| HWV029-2 | 32.40 | 1.08 | 1.00 | 1.98 |
| HWV033-2 | 30.99 | 1.08 | 1.00 | 1.98 |
| HWV037-1 | 38.39 | 0.65 | 1.00 | 2.13 |
| KRS016-1 | 39.57 | 0.98 | 0.00 | 2.63 |
| LAT003-1 | 55.19 | 0.24 | 1.00 | 2.43 |
| LAT004-1 | 55.86 | 0.19 | 1.00 | 2.03 |
| LAT037-1 | 51.52 | 0.64 | 1.00 | 3.25 |
| LCL161-1 | 49.33 | 0.28 | 1.00 | 1.29 |
| LCL229-1 | 50.09 | 1.20 | 1.00 | 4.34 |
| LCL248-1 | 50.59 | 1.22 | 1.00 | 4.36 |
| LCL426-1 | 0.54 | 1.03 | 3.11 | 0.50 |
| LCL427-1 | 30.65 | 1.41 | 1.00 | 2.18 |
| LDA011-1 | 34.65 | 0.52 | 1.00 | 1.48 |
| LDA014-1 | 36.06 | 0.87 | 1.00 | 2.08 |
| MGT035-2 | 32.66 | 0.84 | 1.12 | 1.94 |
| MGT063-1 | 22.48 | 1.09 | 1.47 | 1.71 |
| MSC007-1.008 | 38.86 | 1.00 | 0.00 | 2.36 |
| MSC007-2.005 | 34.04 | 1.00 | 1.00 | 2.10 |
| NLP034-1 | 47.08 | 0.98 | 0.00 | 2.91 |
| NLP049-1 | 30.95 | 0.87 | 1.17 | 1.82 |
| NLP199-1 | 37.81 | 1.00 | 0.00 | 2.17 |
| NUM005-1 | 19.15 | 1.19 | 1.00 | 1.58 |
| NUM006-1 | 26.50 | 1.02 | 1.00 | 1.78 |
| NUM030-1 | 32.85 | 1.14 | 1.00 | 1.98 |
| NUM043-1 | 32.71 | 1.14 | 1.00 | 1.97 |
| NUM284-1.014 | 45.46 | 0.98 | 3.50 | 2.07 |
| NUM288-1 | 44.94 | 0.93 | 0.00 | 2.81 |

| Theorem | *PTCC(t)* [%] | *RPSU(t)* | *RUCT(t)* | *IRS(t)* |
|---|---|---|---|---|
| PLA023-1 | 35.97 | 1.17 | 1.00 | 2.25 |
| PUZ015-1 | 8.03 | 0.85 | 1.00 | 1.08 |
| PUZ018-2 | 41.68 | 1.08 | 1.00 | 2.59 |
| PUZ034-1.004 | 40.59 | 1.37 | 1.00 | 2.82 |
| RNG026-6 | 43.85 | 0.43 | 1.00 | 1.51 |
| RNG028-9 | 43.47 | 0.49 | 1.00 | 1.58 |
| ROB007-1 | 19.42 | 0.89 | 1.00 | 1.38 |
| ROB007-4 | 56.98 | 0.19 | 1.00 | 1.30 |
| ROB014-1 | 54.66 | 0.19 | 1.00 | 1.22 |
| ROB024-1 | 21.28 | 0.85 | 1.00 | 1.38 |
| SET002-6 | 31.28 | 1.09 | 1.00 | 1.85 |
| SET012-3 | 25.57 | 1.02 | 1.00 | 1.74 |
| SET040-6 | 31.41 | 1.10 | 1.00 | 1.96 |
| SET550-6 | 31.62 | 1.10 | 1.00 | 1.94 |
| SWC045-1 | 14.82 | 0.76 | 1.00 | 1.08 |
| SWC197-1 | 14.47 | 0.76 | 1.00 | 1.08 |
| SWC344-1 | 14.62 | 0.76 | 1.00 | 1.08 |
| SWC345-1 | 15.38 | 0.79 | 1.00 | 1.10 |
| SWC390-1 | 15.51 | 0.77 | 1.00 | 1.07 |
| SWV014-1 | 20.84 | 1.01 | 0.00 | 1.02 |
| SWV020-1 | 28.87 | 1.15 | 1.00 | 2.03 |
| SYN440-1 | 22.83 | 0.96 | 0.00 | 1.50 |
| SYN615-1 | 58.86 | 1.07 | 1.00 | 6.51 |
| SYN758-1 | 44.79 | 0.99 | 0.00 | 2.67 |
| SYN802-1 | 43.86 | 0.91 | 0.00 | 2.62 |
| SYN810-1 | 44.14 | 0.85 | 0.00 | 2.55 |
| SYN903-1 | 24.58 | 0.99 | 0.00 | 1.55 |
| TOP001-1 | 39.67 | 0.97 | 1.61 | 2.47 |
| TOP002-1 | 29.98 | 1.01 | 1.24 | 1.99 |
| TOP014-1 | 40.00 | 1.01 | 0.00 | 2.35 |
| TOP019-1 | 39.27 | 0.99 | 1.00 | 2.33 |

# Appendix E

**Figure E-1** is a chart of the inference rates from CARINE 0.72 over the TPTP v2.6.0 library. The specifications of the system to run CARINE are the same as those described in Appendix H.

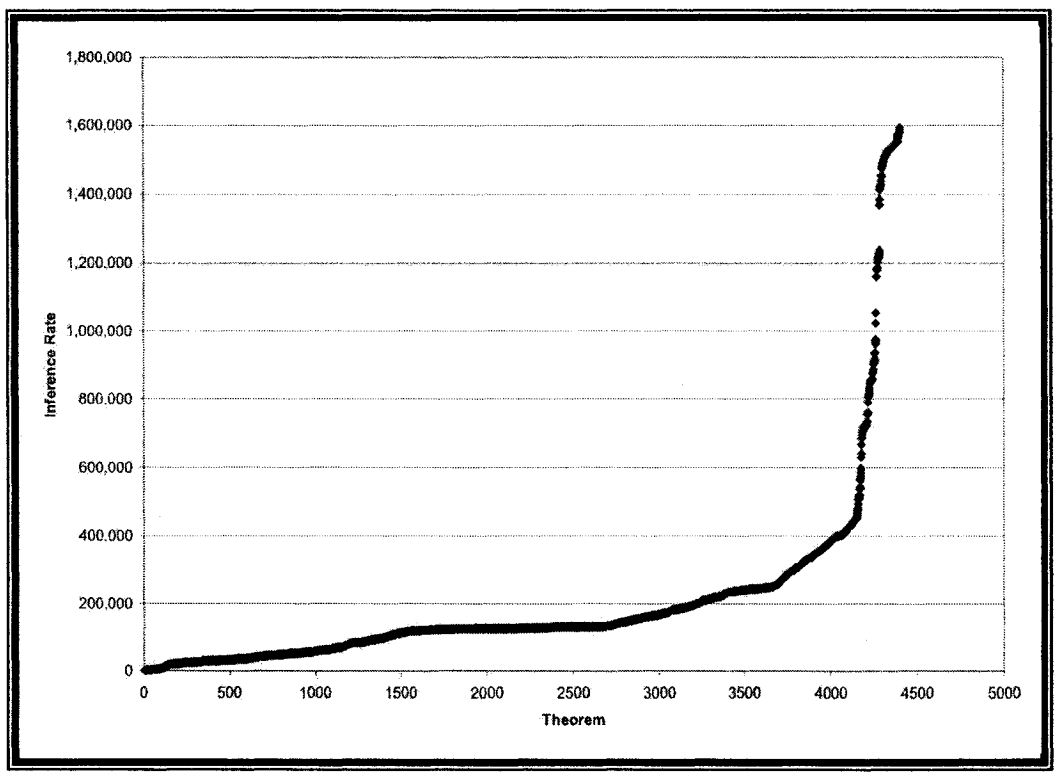



**Figure E-1: Inference rate of CARINE over 4500 theorems**

**Figure E-2** shows the relation between the average branching factor and the number of input clauses up to iteration 4 when tested on over 5003 theorems.

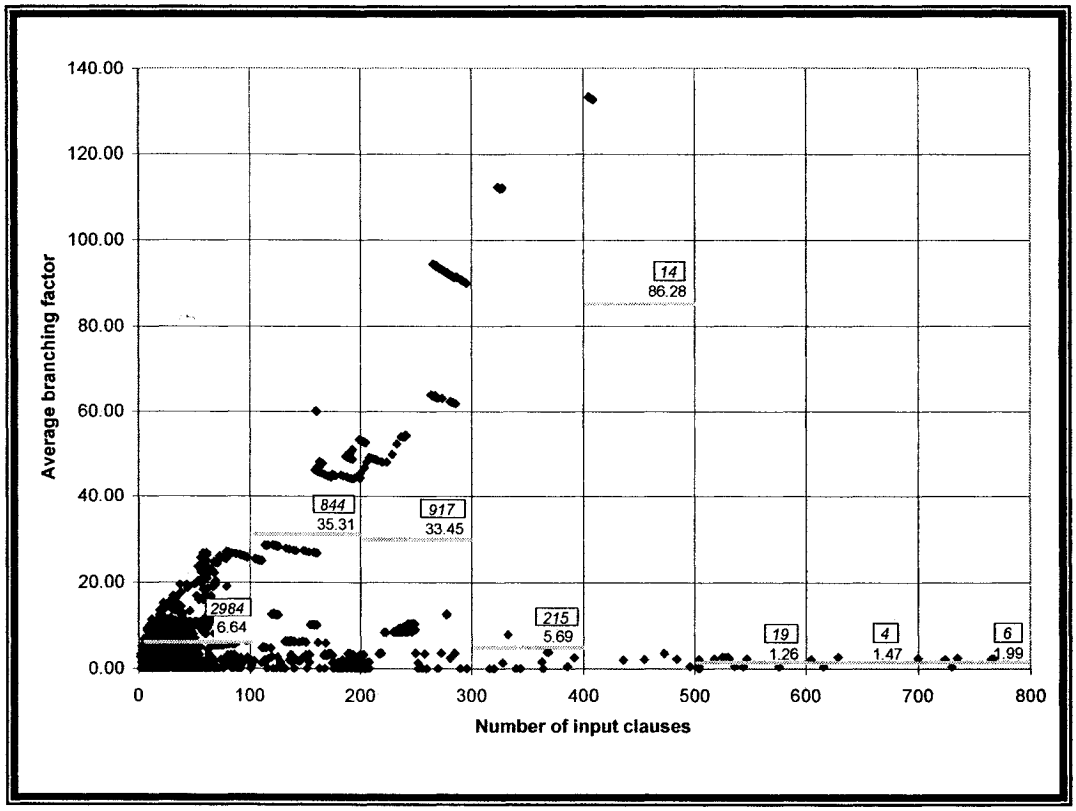



**Figure E-2: Number of input clauses vs. average branching factor up to iteration 4.**

The numbers inside the small boxes and the numbers directly below them indicate, respectively, the number of theorems between two consecutive vertical grid lines and their average branching factor. For example, there are 844 theorems containing between 101 and 200 input clauses and the average of the average branching factors for these 844 theorems is 35.31. The inference rules used are binary resolution and binary factoring. Notice the low branching factor for theorems with over 450 input clauses. Those theorems belong mainly to the SYN and PUZ categories. The ones with a high branching factor are mainly from the NUM and SET categories. There are theorems with more than 800 input clauses,

but we were not able to test them; because either the number of term and literal symbols within the clauses was too large (greater than 500) for our ATP to handle, or because our ATP was not able to reach iteration 4 within a reasonable (less than one hour) amount of time due to the extensive amount of generated clauses.

# Appendix F

The graph in **Figure F-1** shows that the time spent in clause construction is, in general, more than the time spent in unification. Details on the setup of the experiment are given below.

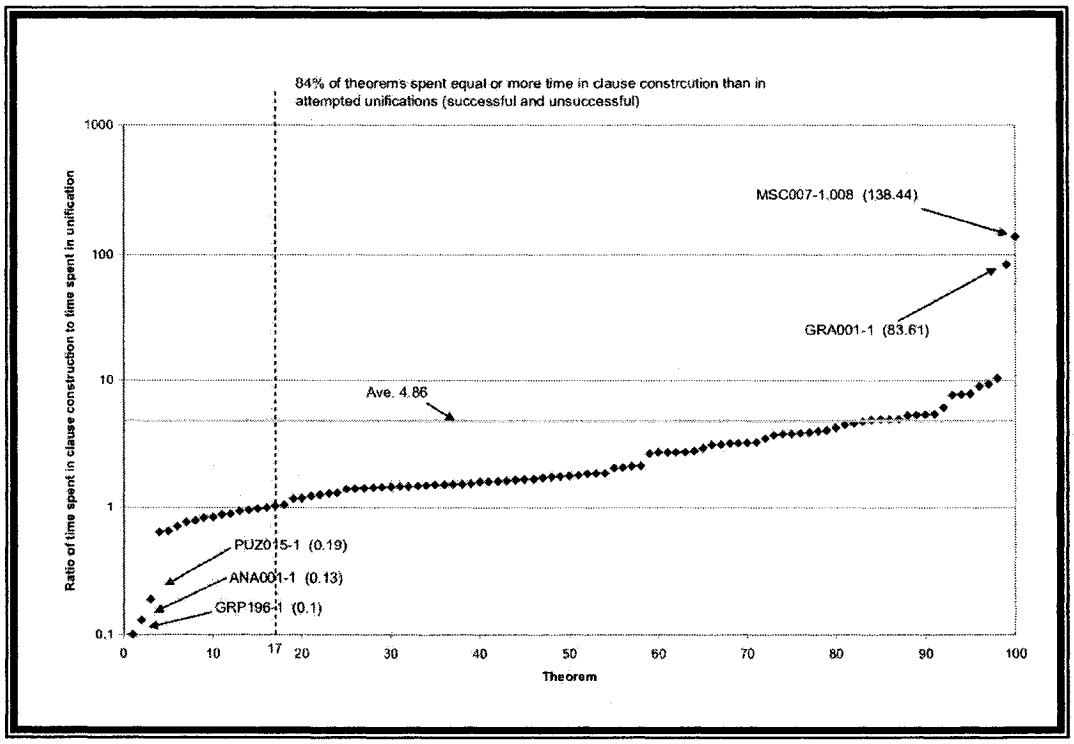



**Figure F-1: Ratio of the time spent in clause construction to the time spent in attempted unifications (successful and unsuccessful).**

Setup: We implemented in CARINE the simplest unification algorithm which was presented in [Robinson 1965]. This algorithm has an exponential time complexity. We also implemented in CARINE a very efficient clause construction algorithm for clauses whose terms are represented as flatterms, i.e. the terms are not shared.

Flatterms are commonly used for fast construction as indicated in [Riazanov 2003]. Clause construction does not include unification. It is, as defined in Chapter 3, the gathering of literals and terms and putting them in a clause structure. Clause construction is done after a successful unification. The construction algorithm is linear in the weight of the clause.

We ran CARINE for 180 seconds on each theorem from the set listed in Appendix D. We measured the time spent in unification (whether the unification was successful or not) of literals or terms during the application of an inference rule. Therefore, the time spent in unification when used in term indexing techniques is not counted. For example, when unification is used to retrieve all the terms that unify with a query term, we do not measure the time spent in such unification.

We measured the time spent in clause construction[1]. The time to calculate the importance of the clause, i.e. weight, and the time to index the clause are not counted as part of the time to construct the clause.

The graph given in **Figure F-1** is the result of our experiment over the 100 theorems that we have selected from the TPTP v2.6.0 (see Appendix D). We conclude that the time spent in unification is in general less than the time spent in clause construction.

[1] The time that the compiler uses to copy symbols or their codes from one memory location to another is not counted. The time to index a clause (i.e., term indexing) or calculate its weight is not counted in the measurement. We just measured the time to traverse the clause.

**Figure F-2** shows that the set *Goals* is generally small (4810 clauses on average). Here the set *Goals* retained only distinct unit clauses.

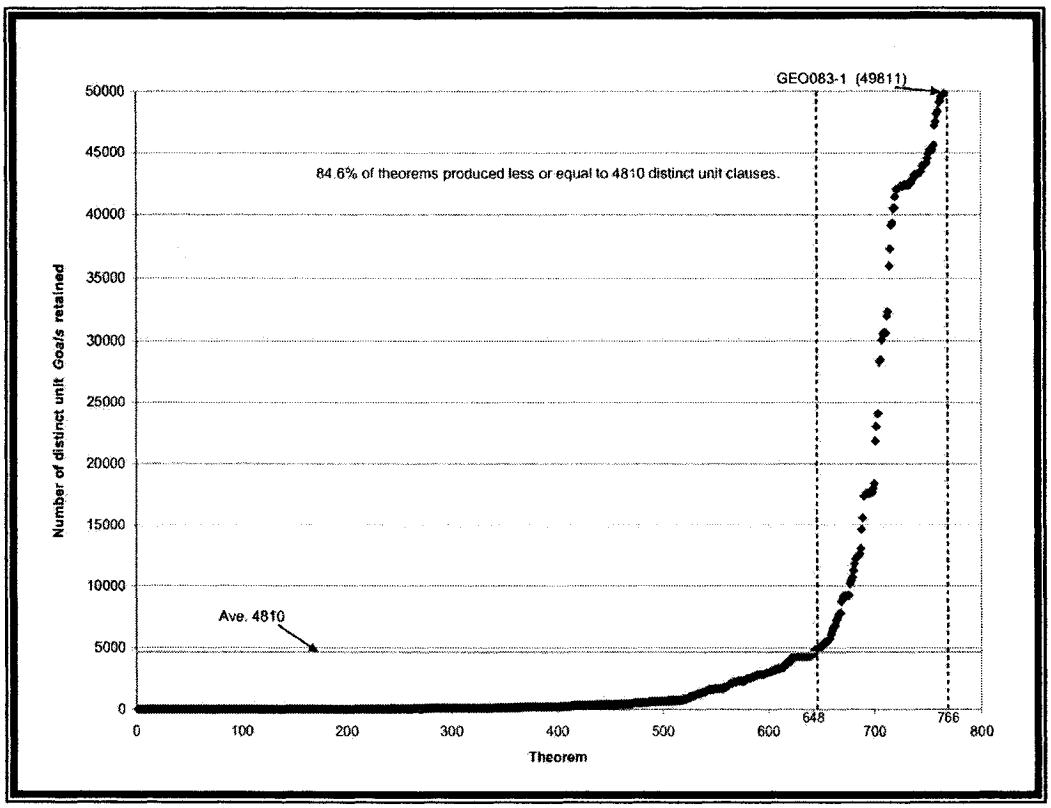



**Figure F-2: Number of distinct unit clauses retained in the set *Goals* in CARINE. The experiment was over 766 theorems that CARINE proved.**

# Appendix G

## G.1 Comparison of different C/C++ compilers

According to experiments done by [Wilson 2004] the Intel's C/C++ compiler version 8.0 can produce code for the Pentium III and Pentium 4 processors that is faster than all the other popular compilers on the market. **Figure G-1** shows that the Intel C/C++ 8.0 complier performed over three times faster than Open Watcom 1.2 on average over the several benchmark tests done by Wilson on a Microsoft Windows based machine with a Pentium class processor.

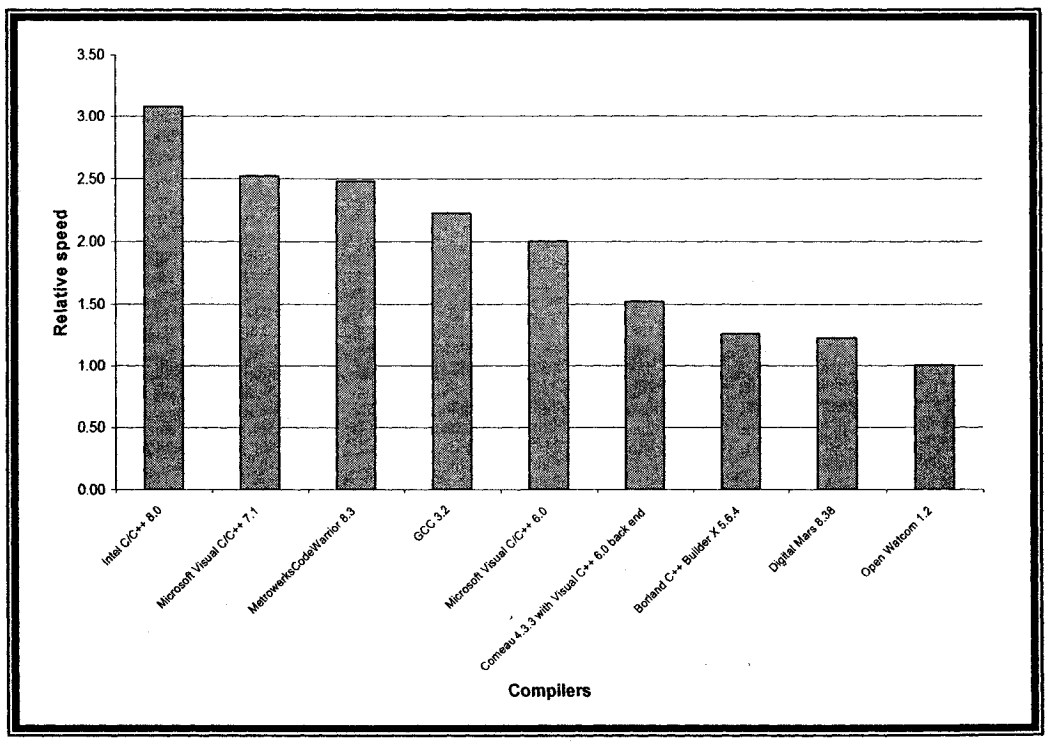



**Figure G-1: Comparisons of relative speeds of popular C/C++ compilers.**

# Appendix H

CARINE proved 949 theorems out of 5473 CNF problems from the TPTP 2.6.0. These are the results on the 949 theorems proved by CARINE. They were obtained from running CARINE 0.72 over the whole set of 5473 CNF problems from TPTP library version 2.6.0. The system used to obtain the results is a Pentium 4 (version *C* which is a bit better than earlier versions *A* and *B* of the processor) machine running at 2.6 GHz equipped with 1GB of DDR RAM. The memory was set to dual channel with 400 MHz bus. The operating system is Microsoft Windows 2000 but the ATP was running under Cygwin (a Linux emulation under Windows 2000). The compiler is gcc version 3.2. The time limit was 180 seconds. We set the size of the unit clauses table, which is part of the *Goals* (see Chapter 4) set, to 32000 entries.

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| ALG002-1 | 0.22 | 22 | 92,627 | 0 | 210 | 1.56 |
| BOO011-1 | 0.14 | 3 | 1,400 | 0 | 32 | 37.65 |
| BOO011-2 | 0.22 | 4 | 19,718 | 0 | 1419 | 33.22 |
| BOO011-4 | 0.55 | 6 | 27,811 | 0 | 5244 | 34.67 |
| BOO018-4 | 0.2 | 6 | 21,585 | 0 | 850 | 20.10 |
| CAT001-3 | 1.3 | 8 | 157,977 | 0.17 | 5017 | 2.88 |
| CAT001-4 | 0.59 | 8 | 129,300 | 0 | 2777 | 4.67 |
| CAT002-3 | 0.22 | 7 | 10,782 | 0.08 | 693 | 33.09 |
| CAT002-4 | 0.22 | 10 | 16,091 | 0.17 | 661 | 20.87 |
| CAT003-3 | 0.22 | 6 | 11,591 | 0.08 | 800 | 34.74 |
| CAT003-4 | 0.2 | 6 | 9,425 | 0 | 562 | 33.69 |
| CAT004-3 | 4.06 | 13 | 51,606 | 0.25 | 6341 | 3.81 |
| CAT004-4 | 0.2 | 10 | 16,190 | 0.17 | 599 | 20.58 |
| CAT005-1 | 0.69 | 17 | 110,978 | 0 | 1574 | 2.47 |
| CAT006-1 | 1.89 | 13 | 58,080 | 0 | 2382 | 2.65 |
| CAT006-3 | 0.75 | 15 | 147,079 | 0.17 | 3557 | 3.28 |
| CAT006-4 | 0.31 | 15 | 101,229 | 0.17 | 1842 | 6.04 |
| CAT007-1 | 0.19 | 7 | 64,326 | 0 | 636 | 8.44 |
| CAT007-3 | 0.14 | 3 | 207 | 0 | 11 | 73.33 |
| CAT008-1 | 40.45 | 13 | 111,194 | 0 | 3536 | 0.09 |
| CAT010-4 | 0.36 | 13 | 118,628 | 0.17 | 1480 | 4.37 |
| CAT011-1 | 0.16 | 5 | 1,394 | 0 | 70 | 50.36 |
| CAT011-2 | 0.19 | 6 | 4,453 | 0 | 276 | 34.94 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| CAT011-3 | 0.94 | 22 | 137,030 | 0.25 | 2689 | 2.39 |
| CAT011-4 | 0.44 | 33 | 111,702 | 0.17 | 1637 | 3.88 |
| CAT012-1 | 0.16 | 5 | 1,388 | 0 | 69 | 51.11 |
| CAT012-3 | 0.2 | 8 | 7,815 | 0.08 | 438 | 31.29 |
| CAT012-4 | 0.19 | 8 | 3,458 | 0.17 | 215 | 37.99 |
| CAT013-1 | 0.16 | 5 | 1,581 | 0 | 89 | 56.69 |
| CAT013-3 | 0.19 | 7 | 8,258 | 0.08 | 441 | 31.39 |
| CAT013-4 | 0.17 | 7 | 3,894 | 0 | 217 | 38.07 |
| CAT014-1 | 0.16 | 5 | 1,650 | 0 | 92 | 55.76 |
| CAT014-2 | 0.2 | 7 | 10,040 | 0 | 484 | 25.50 |
| CAT014-3 | 1.14 | 22 | 139,179 | 0.25 | 3075 | 2.16 |
| CAT014-4 | 0.44 | 25 | 123,286 | 0.17 | 1757 | 3.72 |
| CAT016-3 | 0.2 | 5 | 7,845 | 0.08 | 455 | 31.86 |
| CAT016-4 | 0.16 | 5 | 4,263 | 0 | 230 | 38.92 |
| CAT017-3 | 0.23 | 5 | 6,826 | 0.08 | 455 | 31.86 |
| CAT017-4 | 0.17 | 6 | 3,918 | 0 | 220 | 38.33 |
| CAT018-1 | 0.91 | 17 | 114,127 | 0 | 1265 | 1.41 |
| CAT019-1 | 0.12 | 2 | 17 | 0 | 10 | 100.00 |
| CAT019-2 | 0.19 | 2 | 11 | 0 | 8 | 100.00 |
| COL001-1 | 26.56 | 12 | 2,183 | 0 | 15727 | 27.18 |
| COL007-1 | 0.22 | 1 | 0 | 0 | 3 | 0.00 |
| COL008-1 | 0.55 | 3 | 553 | 0 | 168 | 59.15 |
| COL009-1 | 0.53 | 6 | 658 | 0 | 97 | 31.29 |
| COL010-1 | 0.55 | 3 | 158 | 0 | 44 | 61.97 |
| COL012-1 | 0.55 | 1 | 0 | 0 | 3 | 0.00 |
| COL013-1 | 0.56 | 1 | 0 | 0 | 4 | 0.00 |
| COL014-1 | 0.2 | 1 | 0 | 0 | 3 | 0.00 |
| COL015-1 | 0.55 | 3 | 556 | 0 | 169 | 59.09 |
| COL016-1 | 0.56 | 1 | 0 | 0 | 3 | 0.00 |
| COL017-1 | 0.56 | 3 | 361 | 0 | 112 | 61.54 |
| COL018-1 | 0.55 | 1 | 0 | 0 | 4 | 0.00 |
| COL019-1 | 0.58 | 5 | 819 | 0 | 134 | 30.73 |
| COL020-1 | 0.67 | 10 | 9,572 | 0 | 690 | 12.26 |
| COL021-1 | 0.56 | 3 | 288 | 0 | 90 | 62.94 |
| COL022-1 | 0.56 | 3 | 316 | 0 | 97 | 61.78 |
| COL023-1 | 0.66 | 6 | 11,073 | 0 | 579 | 8.84 |
| COL024-1 | 0.56 | 3 | 288 | 0 | 90 | 62.94 |
| COL025-1 | 0.55 | 3 | 202 | 0 | 55 | 59.14 |
| COL026-1 | 0.55 | 6 | 704 | 0 | 124 | 36.36 |
| COL027-1 | 0.67 | 6 | 13,464 | 0 | 587 | 7.10 |
| COL029-1 | 0.55 | 1 | 0 | 0 | 3 | 0.00 |
| COL030-1 | 0.56 | 7 | 2,152 | 0.09 | 370 | 32.06 |
| COL031-1 | 0.55 | 4 | 1,169 | 0 | 473 | 76.91 |
| COL035-1 | 0.64 | 9 | 5,955 | 0.09 | 1447 | 38.84 |
| COL039-1 | 0.56 | 8 | 4,546 | 0.09 | 871 | 35.16 |
| COL045-1 | 0.56 | 3 | 257 | 0 | 79 | 62.70 |
| COL048-1 | 0.55 | 3 | 325 | 0 | 97 | 61.01 |
| COL050-1 | 0.22 | 6 | 1,600 | 0 | 88 | 28.03 |
| COL051-1 | 0.22 | 7 | 2,927 | 0 | 129 | 22.28 |
| COL052-1 | 0.23 | 10 | 8,517 | 0 | 373 | 20.32 |
| COL052-2 | 0.61 | 11 | 139,105 | 0 | 1912 | 2.45 |
| COL053-1 | 0.62 | 3 | 10,476 | 0 | 4967 | 77.04 |
| COL054-1 | 0.22 | 7 | 3,268 | 0 | 176 | 28.03 |
| COL055-1 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| COL056-1 | 0.22 | 6 | 3,173 | 0.09 | 258 | 39.75 |
| COL058-1 | 0.2 | 9 | 5,220 | 0 | 44 | 5.63 |
| COL058-2 | 0.69 | 15 | 11,326 | 0 | 1683 | 25.32 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| COL070-1 | 0.66 | 6 | 11,333 | 0 | 671 | 10.00 |
| COL083-1 | 0.2 | 1 | 0 | 0 | 4 | 0.00 |
| COL084-1 | 0.19 | 1 | 0 | 0 | 4 | 0.00 |
| COL085-1 | 0.17 | 1 | 0 | 0 | 2 | 0.00 |
| COL086-1 | 0.17 | 1 | 0 | 0 | 2 | 0.00 |
| COM001-1 | 0.19 | 6 | 168 | 0 | 16 | 80.00 |
| COM002-1 | 0.2 | 12 | 1,650 | 0 | 65 | 39.88 |
| COM002-2 | 0.2 | 12 | 2,390 | 0 | 93 | 36.90 |
| COM003-2 | 0.17 | 8 | 441 | 0 | 15 | 65.22 |
| FLD005-3 | 1.69 | 10 | 266,434 | 0 | 4675 | 1.18 |
| FLD006-1 | 0.34 | 7 | 53,888 | 0 | 3550 | 19.68 |
| FLD006-3 | 0.19 | 3 | 42 | 0 | 9 | 100.00 |
| FLD007-3 | 1.99 | 11 | 273,834 | 0.12 | 10571 | 2.29 |
| FLD009-3 | 1.76 | 12 | 263,438 | 0 | 4907 | 1.20 |
| FLD010-1 | 72.02 | 12 | 235,208 | 0 | 10580 | 0.06 |
| FLD010-3 | 0.15 | 4 | 640 | 0 | 33 | 62.26 |
| FLD013-3 | 11 | 18 | 292,228 | 0 | 18707 | 0.64 |
| FLD014-3 | 1.72 | 13 | 262,640 | 0.12 | 4744 | 1.19 |
| FLD015-3 | 1.69 | 12 | 266,821 | 0 | 4616 | 1.16 |
| FLD016-3 | 10.91 | 9 | 294,668 | 0 | 18663 | 0.64 |
| FLD017-3 | 10.92 | 5 | 294,206 | 0 | 18535 | 0.63 |
| FLD018-3 | 0.5 | 7 | 222,332 | 0 | 1806 | 1.93 |
| FLD019-3 | 0.53 | 6 | 215,494 | 0 | 1871 | 1.95 |
| FLD020-3 | 1.69 | 7 | 270,220 | 0 | 4696 | 1.17 |
| FLD021-1 | 0.38 | 6 | 77,763 | 0 | 6971 | 23.82 |
| FLD021-3 | 1.62 | 7 | 277,404 | 0 | 4541 | 1.15 |
| FLD022-3 | 10.84 | 12 | 296,490 | 0 | 18631 | 0.64 |
| FLD023-1 | 0.36 | 8 | 82,069 | 0.12 | 6967 | 23.81 |
| FLD023-3 | 1.62 | 7 | 277,606 | 0 | 4571 | 1.15 |
| FLD024-3 | 1.64 | 7 | 274,161 | 0 | 4571 | 1.15 |
| FLD025-3 | 10.89 | 23 | 295,237 | 0 | 18652 | 0.64 |
| FLD027-3 | 1.87 | 14 | 252,631 | 0 | 5132 | 1.24 |
| FLD028-3 | 11.05 | 10 | 295,370 | 0 | 18937 | 0.64 |
| FLD029-3 | 16.28 | 22 | 286,444 | 0.12 | 20741 | 0.48 |
| FLD030-1 | 0.28 | 4 | 23,511 | 0 | 2462 | 38.53 |
| FLD030-3 | 4.55 | 9 | 291,989 | 0 | 9808 | 0.82 |
| FLD030-4 | 10.73 | 5 | 299,413 | 0 | 18531 | 0.63 |
| FLD031-3 | 0.55 | 8 | 207,684 | 0 | 1869 | 1.94 |
| FLD031-5 | 0.53 | 10 | 219,077 | 0 | 1977 | 2.04 |
| FLD032-3 | 0.58 | 7 | 199,022 | 0 | 1967 | 2.04 |
| FLD033-3 | 1.67 | 8 | 276,687 | 0 | 4731 | 1.16 |
| FLD034-1 | 0.39 | 6 | 75,762 | 0 | 6968 | 23.81 |
| FLD034-3 | 1.62 | 7 | 277,401 | 0 | 4541 | 1.15 |
| FLD035-3 | 10.88 | 13 | 297,298 | 0 | 18764 | 0.64 |
| FLD036-3 | 10.89 | 13 | 297,071 | 0 | 18776 | 0.64 |
| FLD037-3 | 1.64 | 8 | 281,435 | 0 | 4713 | 1.16 |
| FLD038-3 | 1.73 | 8 | 267,829 | 0 | 4863 | 1.19 |
| FLD039-1 | 0.28 | 2 | 86 | 0 | 13 | 100.00 |
| FLD039-3 | 0.56 | 6 | 207,093 | 0 | 2067 | 2.14 |
| FLD055-3 | 4.73 | 7 | 281,403 | 0 | 10011 | 0.84 |
| FLD056-3 | 0.5 | 4 | 213,228 | 0 | 1724 | 1.86 |
| FLD058-1 | 0.41 | 8 | 66,541 | 0 | 6757 | 25.06 |
| FLD058-3 | 1.77 | 9 | 261,625 | 0 | 4852 | 1.19 |
| FLD059-3 | 0.53 | 9 | 206,994 | 0 | 1731 | 1.87 |
| FLD059-4 | 1.7 | 7 | 265,191 | 0 | 4639 | 1.17 |
| FLD060-3 | 18.7 | 18 | 241,132 | 0.12 | 12451 | 0.28 |
| FLD064-3 | 0.52 | 8 | 210,940 | 0 | 1727 | 1.87 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| FLD065-3 | 0.53 | 6 | 206,962 | 0 | 1727 | 1.87 |
| FLD067-1 | 0.36 | 7 | 74,475 | 0.12 | 6539 | 24.65 |
| FLD067-4 | 4.64 | 6 | 286,401 | 0 | 9837 | 0.82 |
| FLD068-4 | 4.69 | 11 | 283,407 | 0 | 9904 | 0.83 |
| FLD069-3 | 1.76 | 13 | 260,142 | 0 | 4846 | 1.20 |
| FLD070-3 | 1.67 | 10 | 269,048 | 0 | 4563 | 1.15 |
| FLD070-4 | 4.78 | 7 | 278,053 | 0 | 9848 | 0.82 |
| FLD071-1 | 0.36 | 3 | 74,561 | 0 | 6555 | 24.69 |
| FLD071-3 | 1.66 | 6 | 270,788 | 0 | 4585 | 1.16 |
| FLD071-4 | 4.72 | 4 | 281,555 | 0 | 9824 | 0.82 |
| GEO001-1 | 0.19 | 17 | 25,426 | 0.25 | 251 | 7.63 |
| GEO001-2 | 0.17 | 17 | 21,847 | 0.25 | 226 | 9.13 |
| GEO002-1 | 0.2 | 15 | 39,220 | 0.25 | 298 | 4.81 |
| GEO002-2 | 1.56 | 15 | 176,915 | 0.25 | 728 | 0.29 |
| GEO002-3 | 0.27 | 2 | 7 | 0.08 | 20 | 100.00 |
| GEO002-4 | 0.19 | 16 | 24,695 | 0 | 53 | 1.77 |
| GEO003-1 | 0.17 | 5 | 429 | 0.08 | 13 | 92.86 |
| GEO003-2 | 0.19 | 5 | 353 | 0.08 | 13 | 92.86 |
| GEO003-3 | 0.27 | 2 | 4 | 0.08 | 18 | 100.00 |
| GEO006-1 | 7.42 | 33 | 159,701 | 0.25 | 2952 | 0.26 |
| GEO006-3 | 26.34 | 10 | 85,948 | 0.17 | 27864 | 1.40 |
| GEO011-2 | 0.19 | 4 | 6,074 | 0.08 | 72 | 16.29 |
| GEO011-4 | 0.3 | 4 | 22,323 | 0.08 | 515 | 13.34 |
| GEO011-5 | 0.28 | 4 | 23,989 | 0.08 | 482 | 12.43 |
| GEO014-2 | 0.19 | 3 | 295 | 0 | 11 | 100.00 |
| GEO015-2 | 0.19 | 5 | 689 | 0.08 | 26 | 66.67 |
| GEO015-3 | 0.17 | 3 | 418 | 0.08 | 14 | 100.00 |
| GEO016-2 | 0.19 | 3 | 337 | 0.08 | 14 | 100.00 |
| GEO016-3 | 0.17 | 3 | 529 | 0.08 | 18 | 64.29 |
| GEO017-2 | 0.17 | 7 | 1,241 | 0.08 | 36 | 37.89 |
| GEO017-3 | 0.19 | 4 | 400 | 0.08 | 19 | 86.36 |
| GEO018-2 | 0.19 | 5 | 1,089 | 0.08 | 34 | 37.36 |
| GEO018-3 | 0.17 | 5 | 553 | 0.08 | 22 | 62.86 |
| GEO019-2 | 0.17 | 3 | 371 | 0.08 | 13 | 100.00 |
| GEO019-3 | 0.17 | 3 | 300 | 0.08 | 16 | 100.00 |
| GEO020-2 | 0.19 | 7 | 1,111 | 0.08 | 36 | 37.89 |
| GEO020-3 | 0.17 | 3 | 306 | 0.08 | 17 | 100.00 |
| GEO021-2 | 0.19 | 5 | 1,089 | 0.08 | 34 | 37.36 |
| GEO021-3 | 0.19 | 4 | 495 | 0.08 | 22 | 62.86 |
| GEO022-2 | 0.17 | 7 | 1,759 | 0.08 | 92 | 52.57 |
| GEO022-3 | 0.17 | 4 | 2,541 | 0.08 | 43 | 14.14 |
| GEO024-2 | 0.19 | 5 | 358 | 0.08 | 15 | 93.75 |
| GEO024-3 | 0.17 | 5 | 3,465 | 0 | 70 | 19.77 |
| GEO035-2 | 0.17 | 2 | 6 | 0.08 | 9 | 100.00 |
| GEO035-3 | 0.17 | 2 | 6 | 0.08 | 10 | 100.00 |
| GEO036-2 | 0.17 | 14 | 5,259 | 0.25 | 62 | 17.08 |
| GEO038-2 | 0.19 | 4 | 416 | 0.08 | 15 | 88.24 |
| GEO038-3 | 0.17 | 4 | 1,529 | 0.08 | 23 | 20.18 |
| GEO039-2 | 0.19 | 10 | 9,221 | 0.17 | 422 | 28.77 |
| GEO040-2 | 0.19 | 12 | 10,663 | 0.25 | 144 | 14.46 |
| GEO041-2 | 8.17 | 42 | 139,064 | 0.42 | 3363 | 0.32 |
| GEO041-3 | 0.31 | 5 | 5,794 | 0.25 | 574 | 41.53 |
| GEO042-2 | 9.03 | 27 | 131,464 | 0.25 | 3587 | 0.32 |
| GEO043-2 | 8.58 | 39 | 134,309 | 0.42 | 3581 | 0.33 |
| GEO054-2 | 0.17 | 4 | 635 | 0.08 | 14 | 43.75 |
| GEO054-3 | 0.17 | 2 | 6 | 0.08 | 12 | 100.00 |
| GEO055-2 | 0.17 | 4 | 641 | 0.08 | 16 | 47.06 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| GEO055-3 | 0.17 | 4 | 2,682 | 0.08 | 45 | 17.58 |
| GEO056-2 | 0.19 | 12 | 4,642 | 0.08 | 184 | 28.75 |
| GEO056-3 | 0.17 | 6 | 2,847 | 0.08 | 69 | 22.55 |
| GEO057-2 | 0.19 | 5 | 558 | 0.08 | 19 | 47.50 |
| GEO057-3 | 0.19 | 2 | 5 | 0.08 | 14 | 100.00 |
| GEO058-2 | 0.19 | 12 | 3,184 | 0.17 | 106 | 26.97 |
| GEO058-3 | 0.2 | 8 | 11,685 | 0.08 | 190 | 11.27 |
| GEO059-2 | 0.38 | 16 | 8,784 | 0.17 | 1323 | 43.74 |
| GEO059-3 | 0.28 | 5 | 4,146 | 0.17 | 339 | 39.70 |
| GEO064-2 | 0.17 | 14 | 9,012 | 0.17 | 104 | 16.43 |
| GEO064-3 | 0.27 | 3 | 1,219 | 0.17 | 93 | 94.90 |
| GEO065-2 | 0.17 | 14 | 9,488 | 0.25 | 121 | 18.03 |
| GEO065-3 | 0.3 | 3 | 1,097 | 0.17 | 93 | 94.90 |
| GEO066-2 | 0.2 | 14 | 7,395 | 0.25 | 99 | 16.20 |
| GEO066-3 | 0.3 | 3 | 1,097 | 0.17 | 93 | 94.90 |
| GEO079-1 | 0.16 | 3 | 13 | 0 | 4 | 100.00 |
| GEO080-1 | 0.22 | 3 | 9 | 0.08 | 5 | 100.00 |
| GEO081-1 | 0.23 | 7 | 665 | 0.08 | 31 | 60.78 |
| GEO082-1 | 5.67 | 13 | 126,348 | 0.17 | 2257 | 1.13 |
| GEO084-1 | 0.25 | 10 | 32,792 | 0.25 | 421 | 11.03 |
| GEO085-1 | 0.22 | 7 | 16,382 | 0.08 | 159 | 14.34 |
| GEO086-1 | 0.25 | 6 | 11,648 | 0 | 129 | 16.43 |
| GEO087-1 | 0.22 | 8 | 14,718 | 0.08 | 119 | 14.99 |
| GEO117-1 | 0.28 | 5 | 1,075 | 0.08 | 73 | 52.14 |
| GEO118-1 | 0.3 | 5 | 1,047 | 0.08 | 76 | 52.05 |
| GEO147-1 | 1.34 | 7 | 35,394 | 0.17 | 29671 | 63.18 |
| GRP001-1 | 4.55 | 40 | 185,071 | 0 | 3017 | 0.36 |
| GRP001-5 | 0.17 | 10 | 4,629 | 0 | 36 | 6.20 |
| GRP003-1 | 0.17 | 7 | 1,747 | 0 | 17 | 9.50 |
| GRP003-2 | 0.3 | 21 | 75,950 | 0 | 280 | 1.27 |
| GRP004-1 | 0.17 | 7 | 2,182 | 0 | 22 | 9.40 |
| GRP004-2 | 0.33 | 22 | 93,091 | 0 | 198 | 0.66 |
| GRP005-1 | 0.17 | 4 | 441 | 0 | 7 | 100.00 |
| GRP006-1 | 1.02 | 7 | 136,421 | 0 | 342 | 0.25 |
| GRP007-1 | 0.16 | 3 | 588 | 0 | 18 | 58.06 |
| GRP009-1 | 2.7 | 36 | 181,506 | 0 | 2657 | 0.55 |
| GRP010-1 | 2.03 | 35 | 153,810 | 0 | 1851 | 0.60 |
| GRP010-4 | 3.66 | 22 | 33,662 | 0 | 7366 | 6.22 |
| GRP012-1 | 0.3 | 11 | 53,397 | 0 | 1284 | 9.65 |
| GRP012-2 | 3.31 | 35 | 183,554 | 0 | 2499 | 0.41 |
| GRP012-4 | 3.14 | 31 | 72,183 | 0 | 18234 | 8.31 |
| GRP013-1 | 0.77 | 16 | 50,610 | 0 | 3509 | 10.41 |
| GRP017-1 | 0.23 | 10 | 70,335 | 0 | 1400 | 10.66 |
| GRP018-1 | 0.17 | 3 | 494 | 0 | 16 | 51.61 |
| GRP019-1 | 0.17 | 3 | 500 | 0 | 17 | 53.13 |
| GRP020-1 | 0.19 | 3 | 595 | 0 | 32 | 53.33 |
| GRP021-1 | 0.2 | 3 | 560 | 0 | 31 | 52.54 |
| GRP022-1 | 0.33 | 10 | 90,324 | 0 | 2772 | 10.91 |
| GRP022-2 | 0.66 | 12 | 106,909 | 0 | 3418 | 5.38 |
| GRP023-1 | 0.17 | 3 | 506 | 0 | 17 | 51.52 |
| GRP023-2 | 0.22 | 4 | 2,332 | 0 | 195 | 40.71 |
| GRP028-1 | 0.17 | 4 | 124 | 0 | 4 | 100.00 |
| GRP028-2 | 0.19 | 4 | 25,337 | 0 | 46 | 1.65 |
| GRP028-3 | 0.17 | 4 | 306 | 0 | 5 | 100.00 |
| GRP028-4 | 0.16 | 4 | 238 | 0 | 4 | 100.00 |
| GRP029-1 | 0.56 | 14 | 143,516 | 0 | 451 | 0.58 |
| GRP029-2 | 0.48 | 14 | 132,767 | 0 | 340 | 0.55 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| GRP030-1 | 0.62 | 21 | 133,118 | 0 | 604 | 0.75 |
| GRP031-1 | 1.47 | 20 | 126,023 | 0 | 2167 | 1.19 |
| GRP031-2 | 0.25 | 13 | 45,200 | 0 | 232 | 2.15 |
| GRP032-3 | 0.16 | 4 | 28,194 | 0 | 245 | 8.87 |
| GRP034-4 | 18 | 49 | 144,337 | 0 | 4058 | 0.16 |
| GRP036-3 | 0.28 | 15 | 71,468 | 0 | 1067 | 6.66 |
| GRP037-3 | 0.2 | 9 | 11,980 | 0 | 543 | 26.84 |
| GRP038-3 | 0.2 | 4 | 31,865 | 0 | 494 | 12.25 |
| GRP041-2 | 0.19 | 3 | 63 | 0 | 5 | 100.00 |
| GRP042-2 | 0.17 | 5 | 253 | 0 | 15 | 100.00 |
| GRP043-2 | 0.17 | 7 | 747 | 0 | 36 | 60.00 |
| GRP044-2 | 0.19 | 6 | 1,479 | 0 | 45 | 42.06 |
| GRP045-2 | 0.23 | 9 | 42,439 | 0 | 226 | 2.47 |
| GRP046-2 | 0.17 | 12 | 1,824 | 0 | 41 | 25.63 |
| GRP047-2 | 0.47 | 37 | 123,696 | 0 | 349 | 0.61 |
| GRP048-2 | 0.73 | 60 | 151,053 | 0.25 | 928 | 0.85 |
| GRP123-1.003 | 4.72 | 29 | 284,995 | 0 | 42 | 0.02 |
| GRP123-3.003 | 5.62 | 29 | 270,148 | 0 | 68 | 0.03 |
| GRP123-6.003 | 0.39 | 21 | 252,756 | 0 | 67 | 0.14 |
| GRP123-7.003 | 0.39 | 21 | 252,756 | 0 | 72 | 0.15 |
| GRP123-8.003 | 0.42 | 21 | 236,902 | 0 | 78 | 0.16 |
| GRP123-9.003 | 0.39 | 21 | 252,756 | 0 | 67 | 0.14 |
| GRP124-2.004 | 129.05 | 87 | 172,617 | 0 | 105 | 0.00 |
| GRP125-1.003 | 18 | 21 | 398,941 | 0 | 39 | 0.00 |
| GRP125-4.003 | 18.05 | 27 | 307,354 | 0 | 39 | 0.00 |
| GRP126-2.004 | 99.28 | 56 | 322,706 | 0 | 85 | 0.00 |
| GRP128-4.003 | 18.02 | 41 | 398,733 | 0 | 37 | 0.05 |
| GRP130-4.003 | 18 | 33 | 412,751 | 0 | 31 | 0.10 |
| GRP135-1.002 | 18 | 25 | 438,119 | 0 | 15 | 0.01 |
| GRP135-2.002 | 18.02 | 25 | 425,357 | 0 | 17 | 0.01 |
| GRP136-1 | 0.3 | 4 | 42,513 | 0 | 3640 | 28.77 |
| GRP137-1 | 0.31 | 4 | 41,142 | 0 | 3640 | 28.77 |
| GRP139-1 | 0.44 | 7 | 46,136 | 0 | 3739 | 18.57 |
| GRP142-1 | 0.27 | 3 | 34,604 | 0 | 2629 | 28.44 |
| GRP143-1 | 0.33 | 8 | 44,876 | 0 | 2687 | 18.32 |
| GRP144-1 | 0.39 | 6 | 46,244 | 0 | 2931 | 16.40 |
| GRP145-1 | 0.28 | 5 | 33,429 | 0 | 2641 | 28.51 |
| GRP146-1 | 0.45 | 7 | 45,111 | 0 | 3739 | 18.57 |
| GRP150-1 | 0.3 | 4 | 31,210 | 0 | 2644 | 28.54 |
| GRP151-1 | 0.2 | 1 | 0 | 0 | 7 | 0.00 |
| GRP152-1 | 0.83 | 9 | 38,790 | 0 | 5130 | 16.07 |
| GRP153-1 | 0.28 | 4 | 33,364 | 0 | 2628 | 28.43 |
| GRP154-1 | 0.3 | 5 | 36,847 | 0 | 3139 | 28.66 |
| GRP155-1 | 0.3 | 5 | 36,847 | 0 | 3139 | 28.66 |
| GRP156-1 | 0.41 | 9 | 43,398 | 0 | 3307 | 18.76 |
| GRP157-1 | 0.3 | 5 | 36,847 | 0 | 3139 | 28.66 |
| GRP158-1 | 0.3 | 5 | 36,847 | 0 | 3139 | 28.66 |
| GRP160-1 | 0.22 | 1 | 0 | 0 | 17 | 0.00 |
| GRP161-1 | 0.22 | 1 | 0 | 0 | 9 | 0.00 |
| GRP162-1 | 1.08 | 10 | 37,355 | 0 | 6355 | 15.86 |
| GRP182-1 | 0.27 | 4 | 34,604 | 0 | 2629 | 28.44 |
| GRP182-2 | 0.3 | 4 | 39,110 | 0 | 3297 | 28.35 |
| GRP182-3 | 0.27 | 4 | 34,600 | 0 | 2628 | 28.43 |
| GRP182-4 | 0.28 | 4 | 41,886 | 0 | 3292 | 28.32 |
| GRP188-1 | 0.83 | 9 | 38,790 | 0 | 5130 | 16.07 |
| GRP188-2 | 1.88 | 9 | 27,320 | 0 | 8545 | 16.75 |
| GRP189-1 | 0.27 | 4 | 34,600 | 0 | 2628 | 28.43 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| GRP189-2 | 0.3 | 4 | 39,090 | 0 | 3291 | 28.31 |
| GRP454-1 | 0.89 | 7 | 25,991 | 0 | 20782 | 90.21 |
| GRP457-1 | 0.86 | 7 | 24,316 | 0 | 18562 | 89.16 |
| GRP460-1 | 0.81 | 7 | 23,081 | 0 | 16346 | 87.87 |
| GRP463-1 | 0.81 | 7 | 23,081 | 0 | 16346 | 87.87 |
| GRP508-1 | 0.53 | 1 | 0 | 0 | 3 | 0.00 |
| GRP512-1 | 0.53 | 1 | 0 | 0 | 3 | 0.00 |
| GRP516-1 | 0.53 | 1 | 0 | 0 | 3 | 0.00 |
| GRP520-1 | 0.55 | 1 | 0 | 0 | 3 | 0.00 |
| GRP524-1 | 0.55 | 1 | 0 | 0 | 5 | 0.00 |
| GRP528-1 | 0.53 | 1 | 0 | 0 | 5 | 0.00 |
| GRP532-1 | 0.55 | 1 | 0 | 0 | 5 | 0.00 |
| GRP536-1 | 0.55 | 1 | 0 | 0 | 6 | 0.00 |
| GRP537-1 | 1.75 | 16 | 37,274 | 0 | 23358 | 35.91 |
| GRP540-1 | 0.55 | 1 | 0 | 0 | 6 | 0.00 |
| GRP544-1 | 0.55 | 1 | 0 | 0 | 6 | 0.00 |
| GRP548-1 | 0.55 | 1 | 0 | 0 | 6 | 0.00 |
| GRP552-1 | 0.53 | 1 | 0 | 0 | 6 | 0.00 |
| GRP556-1 | 0.53 | 1 | 0 | 0 | 4 | 0.00 |
| GRP560-1 | 0.55 | 1 | 0 | 0 | 4 | 0.00 |
| GRP564-1 | 0.55 | 1 | 0 | 0 | 4 | 0.00 |
| GRP568-1 | 0.55 | 1 | 0 | 0 | 6 | 0.00 |
| GRP572-1 | 0.55 | 1 | 0 | 0 | 6 | 0.00 |
| GRP576-1 | 0.53 | 1 | 0 | 0 | 6 | 0.00 |
| GRP580-1 | 0.55 | 1 | 0 | 0 | 6 | 0.00 |
| GRP584-1 | 0.56 | 1 | 0 | 0 | 6 | 0.00 |
| GRP588-1 | 0.55 | 1 | 0 | 0 | 4 | 0.00 |
| GRP592-1 | 0.55 | 1 | 0 | 0 | 4 | 0.00 |
| GRP596-1 | 0.52 | 1 | 0 | 0 | 4 | 0.00 |
| GRP600-1 | 0.55 | 1 | 0 | 0 | 4 | 0.00 |
| GRP604-1 | 0.52 | 1 | 0 | 0 | 4 | 0.00 |
| GRP608-1 | 0.55 | 1 | 0 | 0 | 4 | 0.00 |
| GRP612-1 | 0.53 | 1 | 0 | 0 | 4 | 0.00 |
| GRP616-1 | 0.55 | 1 | 0 | 0 | 4 | 0.00 |
| HEN001-1 | 0.16 | 2 | 6 | 0 | 6 | 100.00 |
| HEN001-3 | 0.22 | 2 | 9 | 0 | 8 | 100.00 |
| HEN001-5 | 0.55 | 1 | 0 | 0 | 2 | 0.00 |
| HEN002-1 | 0.16 | 2 | 6 | 0 | 6 | 100.00 |
| HEN002-2 | 0.15 | 2 | 7 | 0 | 7 | 100.00 |
| HEN002-3 | 0.22 | 2 | 9 | 0 | 8 | 100.00 |
| HEN002-4 | 0.23 | 2 | 9 | 0 | 9 | 100.00 |
| HEN002-5 | 0.53 | 1 | 0 | 0 | 3 | 0.00 |
| HEN003-3 | 0.27 | 12 | 25,681 | 0 | 782 | 11.78 |
| HEN003-4 | 0.28 | 12 | 43,146 | 0 | 1372 | 11.69 |
| HEN004-4 | 3.66 | 16 | 47,499 | 0 | 20842 | 12.70 |
| HEN006-4 | 1.16 | 10 | 121,802 | 0 | 15758 | 11.19 |
| HEN007-4 | 0.47 | 6 | 30,749 | 0 | 4767 | 33.53 |
| HEN008-3 | 0.31 | 14 | 57,268 | 0 | 1368 | 10.15 |
| HEN008-4 | 0.61 | 10 | 28,018 | 0 | 5083 | 30.18 |
| HEN008-6 | 0.25 | 10 | 7,464 | 0 | 755 | 45.05 |
| HEN012-3 | 3.48 | 24 | 147,922 | 0 | 17210 | 3.37 |
| HWV009-1 | 8.98 | 13 | 196,863 | 0.08 | 24159 | 1.67 |
| HWV009-3 | 0.87 | 2 | 7 | 0.08 | 34 | 100.00 |
| HWV009-4 | 0.86 | 2 | 8 | 0.08 | 35 | 100.00 |
| HWV028-1 | 0.31 | 8 | 39,665 | 0.08 | 3936 | 33.27 |
| HWV030-1 | 0.31 | 5 | 39,242 | 0.08 | 3902 | 33.13 |
| HWV030-2 | 8.67 | 4 | 122,545 | 0.08 | 31698 | 3.00 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| HWV032-1 | 0.31 | 5 | 39,242 | 0.08 | 3902 | 33.13 |
| HWV032-2 | 8.72 | 4 | 121,842 | 0.08 | 31698 | 3.00 |
| KRS001-1 | 18 | 11 | 353,498 | 0 | 12 | 0.00 |
| KRS002-1 | 18 | 16 | 308,326 | 0 | 13 | 0.01 |
| KRS003-1 | 18.02 | 11 | 258,699 | 0 | 13 | 0.00 |
| KRS004-1 | 0.17 | 3 | 18 | 0 | 3 | 100.00 |
| KRS012-1 | 0.17 | 8 | 218 | 0 | 6 | 50.00 |
| KRS015-1 | 72 | 22 | 266,891 | 0 | 14 | 0.00 |
| KRS017-1 | 0.15 | 5 | 153 | 0 | 4 | 66.67 |
| LAT014-1 | 0.55 | 5 | 1,473 | 0 | 540 | 70.31 |
| LAT033-1 | 0.22 | 5 | 3,236 | 0 | 180 | 27.03 |
| LAT034-1 | 0.23 | 5 | 3,100 | 0 | 181 | 27.14 |
| LAT039-1 | 0.23 | 4 | 7,739 | 0 | 569 | 32.93 |
| LAT039-2 | 0.22 | 1 | 0 | 0 | 5 | 0.00 |
| LAT090-1 | 14.45 | 16 | 4,261 | 0 | 18338 | 30.99 |
| LCL006-1 | 1.09 | 17 | 6,602 | 0.25 | 2271 | 40.13 |
| LCL007-1 | 0.55 | 3 | 13 | 0 | 4 | 100.00 |
| LCL008-1 | 0.53 | 11 | 40 | 0 | 14 | 93.33 |
| LCL009-1 | 0.88 | 29 | 5,768 | 0 | 2278 | 46.80 |
| LCL010-1 | 0.75 | 19 | 8,484 | 0 | 1616 | 26.87 |
| LCL011-1 | 0.61 | 25 | 3,572 | 0 | 703 | 35.24 |
| LCL013-1 | 0.56 | 5 | 13 | 0 | 4 | 100.00 |
| LCL016-1 | 17.03 | 145 | 32,939 | 0.38 | 11504 | 2.09 |
| LCL018-1 | 144.83 | 147 | 2,720 | 0.5 | 27170 | 7.26 |
| LCL022-1 | 0.59 | 31 | 3,822 | 0 | 555 | 27.07 |
| LCL023-1 | 0.56 | 25 | 4,088 | 0 | 562 | 26.97 |
| LCL027-1 | 0.56 | 7 | 284 | 0 | 135 | 91.84 |
| LCL033-1 | 0.56 | 13 | 354 | 0 | 51 | 38.93 |
| LCL035-1 | 0.55 | 13 | 711 | 0 | 83 | 27.39 |
| LCL041-1 | 2.11 | 11 | 13,447 | 0 | 3279 | 15.16 |
| LCL043-1 | 0.55 | 5 | 309 | 0 | 117 | 76.47 |
| LCL044-1 | 0.55 | 13 | 325 | 0 | 124 | 76.54 |
| LCL045-1 | 0.56 | 13 | 414 | 0.12 | 160 | 74.42 |
| LCL046-1 | 0.53 | 5 | 68 | 0 | 28 | 100.00 |
| LCL076-2 | 0.56 | 3 | 21 | 0 | 9 | 100.00 |
| LCL077-2 | 12.58 | 11 | 11,901 | 0 | 12525 | 9.45 |
| LCL079-1 | 1.14 | 7 | 34,018 | 0 | 2659 | 7.49 |
| LCL080-1 | 1.3 | 25 | 9,721 | 0.38 | 2143 | 21.35 |
| LCL081-1 | 0.67 | 23 | 2,245 | 0 | 1073 | 76.92 |
| LCL082-1 | 0.56 | 17 | 929 | 0 | 167 | 40.34 |
| LCL083-2 | 1.56 | 23 | 5,885 | 0.12 | 4866 | 55.30 |
| LCL086-1 | 4.7 | 37 | 3,480 | 0 | 6080 | 37.89 |
| LCL087-1 | 0.8 | 23 | 10,441 | 0 | 1136 | 14.20 |
| LCL088-1 | 9.27 | 31 | 4,395 | 0 | 6529 | 16.17 |
| LCL089-1 | 60.34 | 27 | 1,669 | 0.12 | 12976 | 12.94 |
| LCL094-1 | 102.31 | 67 | 2,288 | 0.12 | 17432 | 7.98 |
| LCL096-1 | 0.91 | 15 | 2,565 | 0 | 1901 | 81.90 |
| LCL097-1 | 0.56 | 19 | 80 | 0 | 28 | 75.68 |
| LCL098-1 | 0.56 | 19 | 61 | 0 | 21 | 75.00 |
| LCL101-1 | 1.61 | 21 | 55,637 | 0 | 575 | 0.65 |
| LCL102-1 | 0.62 | 25 | 913 | 0 | 445 | 80.47 |
| LCL103-1 | 20.53 | 55 | 17,440 | 0.75 | 11256 | 3.17 |
| LCL104-1 | 0.62 | 19 | 7,365 | 0 | 535 | 12.86 |
| LCL106-1 | 0.56 | 13 | 2,346 | 0 | 196 | 22.32 |
| LCL107-1 | 2.66 | 51 | 6,610 | 0 | 2876 | 16.54 |
| LCL108-1 | 0.58 | 37 | 2,200 | 0 | 199 | 17.56 |
| LCL111-1 | 0.86 | 11 | 10,915 | 0 | 1821 | 24.43 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| LCL117-1 | 0.55 | 13 | 35 | 0 | 11 | 84.62 |
| LCL118-1 | 0.55 | 17 | 1,276 | 0 | 155 | 27.43 |
| LCL120-1 | 0.52 | 15 | 1,652 | 0 | 152 | 22.13 |
| LCL123-1 | 7.47 | 149 | 5,809 | 0.25 | 12602 | 29.22 |
| LCL126-1 | 0.55 | 9 | 164 | 0 | 69 | 85.19 |
| LCL128-1 | 26.66 | 117 | 550 | 0.38 | 6685 | 46.76 |
| LCL130-1 | 0.55 | 17 | 540 | 0.12 | 54 | 25.84 |
| LCL131-1 | 83.78 | 95 | 1,696 | 0.38 | 14840 | 11.84 |
| LCL166-1 | 34.52 | 129 | 12,952 | 0.62 | 9217 | 2.07 |
| LCL169-1 | 0.55 | 2 | 2 | 0 | 7 | 100.00 |
| LCL169-3 | 0.61 | 6 | 7,369 | 0 | 3337 | 75.21 |
| LCL170-1 | 0.56 | 2 | 2 | 0 | 7 | 100.00 |
| LCL170-3 | 0.56 | 6 | 8,023 | 0 | 3335 | 75.20 |
| LCL171-1 | 0.56 | 2 | 2 | 0 | 7 | 100.00 |
| LCL172-1 | 0.56 | 2 | 2 | 0 | 7 | 100.00 |
| LCL173-1 | 0.55 | 2 | 2 | 0 | 7 | 100.00 |
| LCL174-1 | 0.56 | 4 | 54 | 0 | 20 | 100.00 |
| LCL175-1 | 0.53 | 2 | 2 | 0 | 7 | 100.00 |
| LCL175-3 | 0.56 | 2 | 2 | 0 | 9 | 100.00 |
| LCL176-1 | 0.53 | 4 | 40 | 0 | 13 | 100.00 |
| LCL177-1 | 0.55 | 6 | 60 | 0 | 23 | 100.00 |
| LCL178-1 | 0.56 | 6 | 68 | 0 | 25 | 100.00 |
| LCL181-2 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| LCL185-1 | 0.56 | 4 | 38 | 0 | 13 | 100.00 |
| LCL186-1 | 0.53 | 4 | 40 | 0 | 13 | 100.00 |
| LCL187-1 | 0.55 | 6 | 73 | 0 | 27 | 100.00 |
| LCL188-1 | 0.55 | 6 | 62 | 0 | 24 | 100.00 |
| LCL189-1 | 0.58 | 6 | 71 | 0 | 28 | 100.00 |
| LCL190-1 | 0.53 | 4 | 58 | 0 | 21 | 100.00 |
| LCL190-3 | 0.62 | 5 | 7,273 | 0 | 3342 | 75.17 |
| LCL192-1 | 0.72 | 8 | 21,521 | 0 | 2487 | 21.25 |
| LCL193-1 | 0.8 | 6 | 17,670 | 0 | 2950 | 24.46 |
| LCL194-1 | 0.8 | 8 | 16,991 | 0 | 2524 | 21.91 |
| LCL197-1 | 0.7 | 8 | 15,779 | 0 | 1844 | 20.12 |
| LCL199-1 | 0.56 | 12 | 1,021 | 0 | 367 | 66.37 |
| LCL200-1 | 0.58 | 10 | 986 | 0 | 367 | 66.37 |
| LCL202-1 | 1 | 24 | 42,576 | 0 | 10554 | 29.87 |
| LCL203-1 | 1.23 | 24 | 38,433 | 0 | 11674 | 29.51 |
| LCL205-1 | 1.3 | 24 | 36,545 | 0 | 12872 | 32.37 |
| LCL206-1 | 1.24 | 24 | 37,903 | 0 | 12516 | 31.83 |
| LCL226-1 | 0.55 | 4 | 53 | 0 | 19 | 100.00 |
| LCL230-2 | 0.17 | 3 | 24 | 0 | 4 | 100.00 |
| LCL236-1 | 0.55 | 6 | 73 | 0 | 27 | 100.00 |
| LCL238-1 | 0.58 | 10 | 986 | 0 | 386 | 69.80 |
| LCL257-1 | 0.59 | 23 | 4,559 | 0 | 633 | 25.63 |
| LCL355-1 | 0.55 | 3 | 16 | 0 | 5 | 100.00 |
| LCL356-1 | 0.63 | 13 | 10,683 | 0 | 611 | 12.76 |
| LCL357-1 | 0.56 | 5 | 25 | 0 | 9 | 100.00 |
| LCL358-1 | 0.7 | 17 | 18,557 | 0.12 | 1073 | 11.76 |
| LCL359-1 | 0.62 | 29 | 10,855 | 0 | 611 | 12.76 |
| LCL360-1 | 0.56 | 3 | 18 | 0 | 6 | 100.00 |
| LCL361-1 | 0.56 | 11 | 196 | 0 | 80 | 80.00 |
| LCL362-1 | 0.61 | 9 | 1,285 | 0 | 639 | 82.77 |
| LCL366-1 | 0.59 | 13 | 8,280 | 0 | 766 | 20.89 |
| LCL398-1 | 0.55 | 9 | 165 | 0 | 72 | 90.00 |
| LCL414-1 | 0.53 | 5 | 221 | 0 | 89 | 83.18 |
| LCL416-1 | 4.69 | 95 | 29,420 | 0.25 | 2985 | 2.17 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| LDA003-1 | 0.31 | 12 | 61,484 | 0 | 2212 | 22.05 |
| LDA007-3 | 0.91 | 26 | 126,360 | 0 | 6761 | 5.99 |
| MGT001-1 | 137.41 | 44 | 264,354 | 0 | 20 | 0.00 |
| MGT002-1 | 74.36 | 20 | 382,804 | 0 | 13 | 0.00 |
| MGT003-1 | 20.31 | 20 | 376,558 | 0 | 13 | 0.00 |
| MGT004-1 | 49.78 | 35 | 269,115 | 0 | 16 | 0.00 |
| MGT006-1 | 18.06 | 28 | 284,296 | 0 | 14 | 0.00 |
| MGT007-1 | 171.09 | 35 | 243,151 | 0 | 18 | 0.00 |
| MGT008-1 | 2.2 | 21 | 276,660 | 0 | 16 | 0.02 |
| MGT009-1 | 2.73 | 21 | 279,956 | 0 | 16 | 0.02 |
| MGT010-1 | 129.5 | 36 | 305,954 | 0 | 19 | 0.00 |
| MGT013-1 | 77.94 | 28 | 335,253 | 0 | 35 | 0.00 |
| MGT014-1 | 77.94 | 28 | 335,726 | 0 | 35 | 0.00 |
| MGT021-1 | 74.47 | 20 | 230,987 | 0.08 | 137 | 0.00 |
| MGT022-1 | 72.02 | 19 | 380,913 | 0 | 7 | 14.00 |
| MGT022-2 | 72 | 19 | 380,161 | 0 | 7 | 14.00 |
| MGT023-1 | 73.55 | 23 | 209,239 | 0.17 | 195 | 0.00 |
| MGT032-2 | 0.22 | 17 | 500 | 0 | 8 | 21.62 |
| MGT036-1 | 0.95 | 16 | 308,997 | 0 | 15 | 0.02 |
| MGT036-2 | 18 | 16 | 359,237 | 0 | 23 | 0.00 |
| MGT036-3 | 18 | 11 | **473,113** | 0 | 7 | 0.00 |
| MGT041-2 | 0.19 | 10 | 632 | 0 | 8 | 22.22 |
| MGT044-1 | 0.19 | 8 | 22,947 | 0.08 | 46 | 4.43 |
| MGT045-1 | 0.19 | 5 | 1,000 | 0.08 | 27 | 40.30 |
| MGT048-1 | 0.19 | 8 | 22,768 | 0.08 | 44 | 4.26 |
| MGT049-1 | 0.17 | 4 | 1,124 | 0.08 | 30 | 37.04 |
| MGT052-1 | 0.17 | 5 | 353 | 0 | 9 | 50.00 |
| MGT056-1 | 72 | 64 | 189,675 | 0.08 | 790 | 0.01 |
| MGT057-1 | 2.98 | 22 | 179,069 | 0.08 | 688 | 0.19 |
| MGT058-1 | 100.8 | 48 | 205,897 | 0.25 | 456 | 0.00 |
| MGT059-1 | 0.19 | 8 | 5,411 | 0.08 | 40 | 14.55 |
| MGT061-1 | 90.36 | 90 | 140,524 | 0.17 | 7573 | 0.07 |
| MGT065-1 | 100.5 | 61 | 121,149 | 0.25 | 2618 | 0.03 |
| MSC001-1 | 0.3 | 18 | 81,713 | 0 | 178 | 1.25 |
| MSC002-1 | 1.67 | 11 | 130,225 | 0 | 4877 | 2.80 |
| MSC002-2 | 1.64 | 11 | 127,792 | 0 | 4877 | 2.80 |
| MSC003-1 | 0.17 | 8 | 312 | 0 | 7 | 50.00 |
| MSC004-1 | 0.23 | 23 | 4,483 | 0 | 10 | 8.70 |
| MSC005-1 | 0.56 | 9 | 134 | 0 | 49 | 92.45 |
| MSC006-1 | 0.86 | 47 | 307,209 | 0 | 28 | 0.07 |
| NLP141-1 | 0.3 | 33 | 27,357 | 0.08 | 381 | 14.33 |
| NLP143-1 | 0.3 | 33 | 27,993 | 0.08 | 405 | 14.43 |
| NLP145-1 | 0.31 | 33 | 27,290 | 0.08 | 424 | 14.80 |
| NLP147-1 | 0.3 | 33 | 27,993 | 0.08 | 405 | 14.43 |
| NLP149-1 | 0.3 | 33 | 27,357 | 0.17 | 381 | 14.33 |
| NLP204-1 | 0.3 | 12 | 7,377 | 0 | 193 | 27.89 |
| NLP208-1 | 0.3 | 12 | 7,377 | 0.08 | 193 | 27.89 |
| NUM001-1 | 0.23 | 7 | 32,096 | 0 | 752 | 10.34 |
| NUM002-1 | 0.27 | 13 | 28,907 | 0 | 792 | 10.29 |
| NUM003-1 | 0.31 | 7 | 38,581 | 0 | 1457 | 12.30 |
| NUM004-1 | 0.27 | 13 | 28,622 | 0 | 776 | 10.19 |
| NUM009-1 | 9.59 | 5 | 86,351 | 0.08 | 26263 | 3.28 |
| NUM014-1 | 18 | 6 | 409,105 | 0 | 12 | 0.00 |
| NUM015-1 | 0.17 | 13 | 2,353 | 0 | 21 | 18.58 |
| NUM016-1 | 0.16 | 11 | 4,719 | 0 | 27 | 10.42 |
| NUM016-2 | 0.17 | 14 | 1,312 | 0 | 10 | 19.23 |
| NUM019-1 | 0.2 | 3 | 435 | 0 | 35 | 53.03 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| NUM020-1 | 0.2 | 4 | 840 | 0 | 57 | 40.43 |
| NUM022-1 | 0.56 | 9 | 525 | 0 | 11 | 16.18 |
| NUM023-1 | 0.55 | 2 | 2 | 0 | 9 | 100.00 |
| NUM024-1 | 1.3 | 9 | 4,535 | 0 | 3606 | 62.43 |
| NUM025-1 | 0.58 | 3 | 576 | 0 | 152 | 50.84 |
| NUM025-2 | 0.58 | 3 | 576 | 0 | 152 | 50.84 |
| NUM139-1 | 1 | 3 | 13 | 0.25 | 58 | 100.00 |
| NUM228-1 | 1.01 | 3 | 10 | 0.17 | 57 | 100.00 |
| PLA001-1 | 0.22 | 14 | 8,400 | 0.12 | 90 | 7.73 |
| PLA002-1 | 0.17 | 10 | 2,935 | 0 | 12 | 11.01 |
| PLA003-1 | 18 | 8 | 330,024 | 0 | 25 | 0.00 |
| PLA006-1 | 0.47 | 6 | 110,847 | 0 | 3551 | 8.16 |
| PLA017-1 | 1.26 | 10 | 92,193 | 0 | 3729 | 3.53 |
| PLA020-1 | 0.38 | 4 | 113,997 | 0 | 3545 | 10.07 |
| PLA022-1 | 1.64 | 14 | 90,468 | 0.25 | 4273 | 3.50 |
| PLA022-2 | 1.72 | 14 | 86,293 | 0.25 | 4274 | 3.50 |
| PUZ001-1 | 0.16 | 13 | 1,331 | 0 | 19 | 29.23 |
| PUZ001-2 | 0.47 | 24 | 131,517 | 0.17 | 280 | 0.70 |
| PUZ002-1 | 0.17 | 11 | 182 | 0 | 11 | 73.33 |
| PUZ003-1 | 0.17 | 11 | 2,612 | 0 | 15 | 17.65 |
| PUZ004-1 | 0.19 | 10 | 279 | 0 | 11 | 45.83 |
| PUZ005-1 | 72.27 | 68 | 253,191 | 0 | 67 | 0.01 |
| PUZ006-1 | 0.31 | 41 | 90,919 | 0.08 | 108 | 1.15 |
| PUZ008-1 | 0.58 | 2 | 3 | 0 | 8 | 100.00 |
| PUZ008-2 | 0.25 | 12 | 1,704 | 0 | 15 | 19.23 |
| PUZ009-1 | 0.17 | 8 | 6,959 | 0 | 6 | 3.90 |
| PUZ012-1 | 0.17 | 12 | 1,382 | 0 | 25 | 39.68 |
| PUZ013-1 | 0.19 | 11 | 405 | 0 | 8 | 44.44 |
| PUZ014-1 | 0.17 | 36 | 6,988 | 0 | 14 | 12.96 |
| PUZ018-1 | 0.77 | 43 | 150,509 | 0.17 | 60 | 0.81 |
| PUZ020-1 | 0.14 | 5 | 3,950 | 0 | 41 | 14.34 |
| PUZ021-1 | 25.09 | 40 | 188,961 | 0 | 1817 | 0.05 |
| PUZ022-1 | 72.02 | 9 | 127,370 | 0 | 59 | 0.00 |
| PUZ023-1 | 0.17 | 13 | 4,718 | 0 | 8 | 100.00 |
| PUZ024-1 | 18.02 | 7 | 297,501 | 0 | 35 | 0.01 |
| PUZ026-1 | 0.78 | 52 | 257,373 | 0 | 74 | 0.38 |
| PUZ027-1 | 0.22 | 32 | 46,395 | 0 | 30 | 7.46 |
| PUZ029-1 | 18 | 25 | 363,808 | 0 | 14 | 0.00 |
| PUZ032-1 | 0.53 | 10 | 1,177 | 0 | 22 | 15.38 |
| PUZ033-1 | 3.61 | 25 | 438,978 | 0 | 12 | 0.00 |
| PUZ035-1 | 2.2 | 85 | 316,402 | 0 | 14 | 2.62 |
| PUZ035-2 | 4.97 | 85 | 330,780 | 0 | 14 | 1.52 |
| PUZ035-5 | 0.2 | 10 | 1,190 | 0 | 4 | 66.67 |
| PUZ035-6 | 0.56 | 10 | 1,102 | 0 | 4 | 66.67 |
| PUZ047-1 | 0.17 | 8 | 1,229 | 0 | 13 | 32.50 |
| RNG001-3 | 26.5 | 60 | 204,889 | 0.12 | 357 | 0.01 |
| RNG002-1 | 7.53 | 26 | 177,232 | 0 | 5355 | 0.41 |
| RNG003-1 | 7.64 | 24 | 176,038 | 0 | 5368 | 0.40 |
| RNG010-2 | 0.58 | 1 | 0 | 0 | 12 | 0.00 |
| RNG011-5 | 0.56 | 1 | 0 | 0 | 3 | 0.00 |
| RNG038-2 | 0.19 | 6 | 28,137 | 0 | 424 | 14.14 |
| RNG039-2 | 3.03 | 21 | 40,374 | 0.12 | 10446 | 8.67 |
| RNG041-1 | 0.56 | 11 | 134,505 | 0.17 | 3104 | 5.42 |
| SET001-1 | 0.16 | 4 | 56 | 0 | 6 | 100.00 |
| SET002-1 | 0.17 | 15 | 23,741 | 0 | 12 | 4.27 |
| SET003-1 | 0.17 | 5 | 100 | 0 | 7 | 53.85 |
| SET004-1 | 0.16 | 5 | 106 | 0 | 7 | 53.85 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| SET005-1 | 9.84 | 53 | 221,466 | 0 | 174 | 0.15 |
| SET006-1 | 0.16 | 5 | 131 | 0 | 7 | 50.00 |
| SET008-1 | 0.16 | 7 | 500 | 0 | 26 | 78.79 |
| SET009-1 | 0.16 | 12 | 1,206 | 0 | 18 | 51.43 |
| SET011-1 | 4.38 | 34 | 200,193 | 0 | 194 | 0.16 |
| SET014-2 | 10.62 | 19 | 185,731 | 0 | 3277 | 0.17 |
| SET043-5 | 0.17 | 3 | 41 | 0 | 2 | 66.67 |
| SET044-5 | 0.17 | 7 | 671 | 0 | 5 | 33.33 |
| SET045-5 | 0.17 | 4 | 106 | 0 | 8 | 66.67 |
| SET046-5 | 0.16 | 14 | 9,213 | 0 | 6 | 3.47 |
| SET054-6 | 0.81 | 2 | 1 | 0 | 32 | 100.00 |
| SET054-7 | 0.84 | 2 | 1 | 0 | 32 | 100.00 |
| SET060-6 | 4.74 | 4 | 103,497 | 0.08 | 29104 | 5.97 |
| SET060-7 | 4.75 | 4 | 103,194 | 0.08 | 29093 | 5.98 |
| SET062-7 | 0.83 | 2 | 4 | 0.17 | 37 | 100.00 |
| SET064-7 | 0.8 | 3 | 23 | 0.17 | 45 | 100.00 |
| SET078-7 | 0.84 | 2 | 7 | 0.08 | 46 | 100.00 |
| SET080-7 | 0.83 | 2 | 8 | 0.08 | 48 | 100.00 |
| SET196-6 | 2.63 | 4 | 80,654 | 0.25 | 29806 | 14.21 |
| SET231-6 | 0.91 | 3 | 8 | 0.17 | 45 | 100.00 |
| SET296-6 | 0.86 | 1 | 0 | 0.08 | 17 | 0.00 |
| SET786-1 | 0.17 | 14 | 8,035 | 0 | 6 | 4.14 |
| SWV001-1 | 0.94 | 22 | 310,563 | 0 | 10 | 0.03 |
| SWV002-1 | 72.38 | 12 | 89,341 | 0.08 | 17327 | 0.29 |
| SWV003-1 | 0.19 | 3 | 2,484 | 0 | 129 | 38.28 |
| SWV005-1 | 0.95 | 4 | 89,891 | 0.08 | 5412 | 8.28 |
| SWV006-1 | 0.19 | 5 | 2,889 | 0.08 | 157 | 41.42 |
| SWV007-1 | 1.16 | 11 | 91,141 | 0.08 | 6508 | 7.95 |
| SWV008-1 | 74.52 | 18 | 73,406 | 0.17 | 15685 | 0.31 |
| SWV009-1 | 0.17 | 13 | 35,788 | 0 | 11 | 1.44 |
| SWV011-1 | 0.2 | 2 | 5 | 0 | 3 | 100.00 |
| SYN003-1.006 | 0.17 | 16 | 735 | 0 | 19 | 32.76 |
| SYN004-1.007 | 0.27 | 70 | 113,837 | 0 | 14 | 0.16 |
| SYN005-1.010 | 0.17 | 10 | 1,400 | 0 | 11 | 100.00 |
| SYN006-1 | 0.56 | 6 | 50 | 0 | 7 | 87.50 |
| SYN008-1 | 0.17 | 3 | 29 | 0 | 4 | 100.00 |
| SYN009-1 | 0.17 | 6 | 29 | 0 | 7 | 100.00 |
| SYN009-2 | 0.17 | 9 | 406 | 0 | 8 | 25.81 |
| SYN009-3 | 0.19 | 15 | 868 | 0 | 10 | 17.54 |
| SYN009-4 | 0.19 | 21 | 14,395 | 0 | 9 | 3.03 |
| SYN011-1 | 0.19 | 10 | 1,084 | 0 | 5 | 45.45 |
| SYN014-2 | 0.17 | 4 | 353 | 0 | 14 | 60.87 |
| SYN015-2 | 34.13 | 45 | 278,287 | 0 | 37 | 0.00 |
| SYN028-1 | 0.17 | 8 | 259 | 0 | 6 | 26.09 |
| SYN029-1 | 0.16 | 7 | 181 | 0 | 4 | 80.00 |
| SYN030-1 | 0.17 | 14 | 1,853 | 0 | 6 | 17.14 |
| SYN031-1 | 0.17 | 5 | 206 | 0 | 4 | 40.00 |
| SYN032-1 | 0.17 | 21 | 3,847 | 0 | 5 | 11.11 |
| SYN033-1 | 0.17 | 4 | 129 | 0 | 4 | 100.00 |
| SYN034-1 | 0.17 | 14 | 9,388 | 0 | 6 | 3.47 |
| SYN035-1 | 0.17 | 8 | 1,924 | 0 | 3 | 4.62 |
| SYN040-1 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| SYN041-1 | 0.17 | 1 | 0 | 0 | 3 | 0.00 |
| SYN044-1 | 0.16 | 10 | 913 | 0 | 4 | 36.36 |
| SYN045-1 | 0.16 | 4 | 44 | 0 | 4 | 80.00 |
| SYN046-1 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| SYN047-1 | 0.16 | 6 | 175 | 0 | 5 | 62.50 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| SYN048-1 | 0.16 | 1 | 0 | 0 | 2 | 0.00 |
| SYN049-1 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| SYN050-1 | 0.17 | 3 | 24 | 0 | 4 | 100.00 |
| SYN051-1 | 0.16 | 5 | 200 | 0 | 3 | 60.00 |
| SYN052-1 | 0.17 | 5 | 294 | 0 | 3 | 60.00 |
| SYN053-1 | 0.17 | 6 | 612 | 0 | 4 | 14.29 |
| SYN054-1 | 0.17 | 7 | 1,729 | 0 | 8 | 17.39 |
| SYN055-1 | 0.17 | 13 | 724 | 0 | 11 | 26.19 |
| SYN057-1 | 0.17 | 8 | 229 | 0 | 9 | 64.29 |
| SYN058-1 | 0.19 | 5 | 53 | 0 | 7 | 100.00 |
| SYN060-1 | 0.17 | 4 | 29 | 0 | 4 | 100.00 |
| SYN061-1 | 0.17 | 4 | 35 | 0 | 5 | 100.00 |
| SYN062-1 | 0.17 | 6 | 59 | 0 | 6 | 100.00 |
| SYN063-1 | 0.17 | 4 | 112 | 0 | 4 | 100.00 |
| SYN063-2 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| SYN064-1 | 0.16 | 1 | 0 | 0 | 2 | 0.00 |
| SYN065-1 | 0.19 | 3 | 74 | 0 | 4 | 100.00 |
| SYN066-1 | 0.17 | 4 | 94 | 0 | 7 | 77.78 |
| SYN068-1 | 0.17 | 5 | 188 | 0 | 7 | 46.67 |
| SYN069-1 | 0.19 | 19 | 46,158 | 0 | 11 | 0.55 |
| SYN070-1 | 18.03 | 24 | 280,046 | 0 | 35 | 0.00 |
| SYN071-1 | 0.28 | 38 | 133,214 | 0 | 14 | 0.22 |
| SYN073-1 | 0.16 | 2 | 19 | 0 | 4 | 100.00 |
| SYN074-1 | 0.59 | 19 | 164,561 | 0.08 | 203 | 0.30 |
| SYN075-1 | 0.58 | 19 | 169,960 | 0.08 | 206 | 0.31 |
| SYN079-1 | 0.16 | 3 | 44 | 0 | 4 | 100.00 |
| SYN080-1 | 0.19 | 2 | 11 | 0 | 5 | 100.00 |
| SYN081-1 | 0.16 | 6 | 744 | 0 | 4 | 17.39 |
| SYN082-1 | 0.17 | 21 | 15,759 | 0 | 3 | 6.25 |
| SYN083-1 | 0.56 | 3 | 164 | 0 | 43 | 58.11 |
| SYN084-2 | 72 | 46 | 244,093 | 0 | 12 | 0.00 |
| SYN085-1.010 | 0.19 | 11 | 111 | 0 | 12 | 100.00 |
| SYN088-1.010 | 0.2 | 11 | 155 | 0 | 22 | 100.00 |
| SYN089-1.002 | 0.17 | 3 | 71 | 0 | 6 | 100.00 |
| SYN090-1.008 | 62.42 | 170 | 239,112 | 0 | 29 | 0.00 |
| SYN095-1.002 | 0.17 | 3 | 71 | 0 | 6 | 100.00 |
| SYN096-1.008 | 63.62 | 170 | 234,602 | 0 | 29 | 0.00 |
| SYN099-1.003 | 0.19 | 7 | 7,000 | 0 | 23 | 7.90 |
| SYN100-1.005 | 0.3 | 44 | 62,727 | 0 | 33 | 0.72 |
| SYN101-1.002.002 | 0.19 | 7 | 3,421 | 0 | 23 | 14.47 |
| SYN103-1 | 0.22 | 1 | 0 | 0 | 7 | 0.00 |
| SYN104-1 | 0.22 | 1 | 0 | 0 | 6 | 0.00 |
| SYN105-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN106-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN107-1 | 0.31 | 3 | 13 | 0 | 42 | 100.00 |
| SYN108-1 | 0.31 | 3 | 13 | 0 | 42 | 100.00 |
| SYN109-1 | 0.33 | 9 | 19,994 | 0 | 243 | 10.57 |
| SYN110-1 | 0.34 | 11 | 21,476 | 0 | 252 | 10.04 |
| SYN111-1 | 0.31 | 9 | 21,268 | 0 | 243 | 10.58 |
| SYN112-1 | 0.3 | 3 | 17 | 0 | 43 | 100.00 |
| SYN113-1 | 0.34 | 8 | 19,441 | 0 | 245 | 10.62 |
| SYN114-1 | 0.31 | 4 | 1,335 | 0 | 94 | 62.25 |
| SYN115-1 | 0.42 | 13 | 40,460 | 0 | 341 | 5.86 |
| SYN116-1 | 0.59 | 10 | 74,895 | 0 | 382 | 2.19 |
| SYN117-1 | 0.58 | 12 | 76,138 | 0 | 381 | 2.19 |
| SYN118-1 | 0.23 | 1 | 0 | 0 | 8 | 0.00 |
| SYN119-1 | 0.2 | 1 | 0 | 0 | 9 | 0.00 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| SYN120-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN121-1 | 0.34 | 10 | 20,979 | 0 | 248 | 10.03 |
| SYN122-1 | 0.34 | 10 | 20,979 | 0 | 248 | 10.03 |
| SYN123-1 | 0.34 | 7 | 19,394 | 0 | 243 | 10.58 |
| SYN124-1 | 0.31 | 6 | 2,739 | 0 | 136 | 33.17 |
| SYN125-1 | 0.36 | 6 | 18,333 | 0 | 245 | 10.65 |
| SYN126-1 | 0.31 | 5 | 2,445 | 0 | 127 | 35.98 |
| SYN127-1 | 0.3 | 5 | 2,527 | 0 | 127 | 35.98 |
| SYN128-1 | 77.22 | 15 | 145,599 | 0 | 702 | 0.02 |
| SYN129-1 | 77.22 | 15 | 145,363 | 0 | 702 | 0.02 |
| SYN130-1 | 0.22 | 1 | 0 | 0 | 2 | 0.00 |
| SYN131-1 | 0.22 | 1 | 0 | 0 | 2 | 0.00 |
| SYN132-1 | 0.22 | 1 | 0 | 0 | 5 | 0.00 |
| SYN133-1 | 0.22 | 1 | 0 | 0 | 2 | 0.00 |
| SYN134-1 | 0.31 | 4 | 2,687 | 0 | 134 | 33.09 |
| SYN135-1 | 0.3 | 4 | 1,370 | 0 | 91 | 61.49 |
| SYN136-1 | 0.31 | 4 | 1,323 | 0 | 91 | 61.49 |
| SYN137-1 | 1.5 | 22 | 122,523 | 0 | 468 | 0.75 |
| SYN138-1 | 76.77 | 16 | 145,959 | 0 | 652 | 0.02 |
| SYN139-1 | 77.84 | 35 | 145,386 | 0 | 731 | 0.02 |
| SYN140-1 | 77.84 | 34 | 145,635 | 0 | 732 | 0.02 |
| SYN141-1 | 0.41 | 14 | 44,580 | 0 | 367 | 5.84 |
| SYN142-1 | 77.89 | 38 | 145,411 | 0 | 744 | 0.02 |
| SYN143-1 | 77.89 | 38 | 144,964 | 0 | 744 | 0.02 |
| SYN144-1 | 0.56 | 12 | 71,000 | 0 | 379 | 2.46 |
| SYN145-1 | 0.22 | 1 | 0 | 0 | 14 | 0.00 |
| SYN146-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN147-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN148-1 | 0.36 | 8 | 23,533 | 0 | 278 | 9.80 |
| SYN149-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN150-1 | 0.31 | 3 | 94 | 0 | 46 | 100.00 |
| SYN151-1 | 0.31 | 3 | 94 | 0 | 46 | 100.00 |
| SYN152-1 | 0.31 | 3 | 94 | 0 | 46 | 100.00 |
| SYN153-1 | 0.3 | 4 | 1,777 | 0 | 99 | 45.62 |
| SYN154-1 | 0.31 | 4 | 1,710 | 0 | 96 | 44.86 |
| SYN155-1 | 0.41 | 18 | 44,388 | 0 | 365 | 5.82 |
| SYN156-1 | 0.94 | 29 | 111,635 | 0 | 426 | 1.15 |
| SYN157-1 | 0.34 | 11 | 19,612 | 0 | 250 | 10.71 |
| SYN158-1 | 0.33 | 11 | 20,112 | 0 | 245 | 10.56 |
| SYN159-1 | 77.2 | 58 | 145,768 | 0 | 704 | 0.02 |
| SYN160-1 | 0.36 | 9 | 18,444 | 0 | 247 | 10.66 |
| SYN161-1 | 0.33 | 11 | 20,106 | 0 | 246 | 10.63 |
| SYN162-1 | 0.34 | 11 | 19,515 | 0 | 246 | 10.63 |
| SYN163-1 | 77.22 | 45 | 145,875 | 0 | 705 | 0.02 |
| SYN164-1 | 0.22 | 1 | 0 | 0 | 3 | 0.00 |
| SYN165-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN166-1 | 0.31 | 8 | 3,758 | 0 | 162 | 29.72 |
| SYN167-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN168-1 | 0.31 | 4 | 1,335 | 0 | 92 | 61.74 |
| SYN169-1 | 0.3 | 4 | 1,377 | 0 | 91 | 61.49 |
| SYN170-1 | 0.31 | 5 | 2,000 | 0 | 108 | 38.99 |
| SYN171-1 | 76.47 | 31 | 145,692 | 0 | 637 | 0.02 |
| SYN172-1 | 0.22 | 1 | 0 | 0 | 4 | 0.00 |
| SYN173-1 | 0.3 | 3 | 2,777 | 0 | 134 | 33.09 |
| SYN174-1 | 0.31 | 3 | 2,687 | 0 | 134 | 33.09 |
| SYN175-1 | 0.3 | 3 | 113 | 0 | 48 | 100.00 |
| SYN176-1 | 0.34 | 10 | 19,482 | 0 | 243 | 10.55 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| SYN177-1 | 0.31 | 6 | 4,474 | 0 | 183 | 29.80 |
| SYN178-1 | 72.83 | 13 | 145,636 | 0 | 606 | 0.02 |
| SYN179-1 | 0.42 | 17 | 40,495 | 0 | 345 | 5.93 |
| SYN180-1 | 72.83 | 20 | 145,744 | 0 | 607 | 0.02 |
| SYN181-1 | 0.92 | 14 | 114,164 | 0 | 431 | 1.16 |
| SYN182-1 | 0.31 | 6 | 2,458 | 0 | 130 | 36.52 |
| SYN183-1 | 0.3 | 6 | 2,540 | 0 | 130 | 36.52 |
| SYN184-1 | 0.23 | 1 | 0 | 0 | 11 | 0.00 |
| SYN185-1 | 0.2 | 1 | 0 | 0 | 10 | 0.00 |
| SYN186-1 | 0.34 | 5 | 19,382 | 0 | 242 | 10.54 |
| SYN187-1 | 0.33 | 5 | 19,970 | 0 | 242 | 10.54 |
| SYN188-1 | 0.34 | 5 | 19,382 | 0 | 242 | 10.54 |
| SYN189-1 | 0.36 | 7 | 27,019 | 0 | 283 | 8.35 |
| SYN190-1 | 1.38 | 26 | 121,251 | 0 | 450 | 0.79 |
| SYN191-1 | 0.42 | 9 | 45,476 | 0 | 366 | 5.66 |
| SYN192-1 | 0.33 | 12 | 19,970 | 0 | 242 | 10.54 |
| SYN193-1 | 0.33 | 12 | 19,970 | 0 | 242 | 10.54 |
| SYN194-1 | 0.41 | 14 | 44,378 | 0 | 365 | 5.82 |
| SYN195-1 | 0.56 | 13 | 79,077 | 0 | 386 | 2.21 |
| SYN196-1 | 0.31 | 4 | 2,697 | 0 | 135 | 33.09 |
| SYN197-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN198-1 | 0.33 | 4 | 2,533 | 0 | 135 | 33.09 |
| SYN199-1 | 0.31 | 4 | 2,697 | 0 | 135 | 33.09 |
| SYN200-1 | 0.31 | 4 | 2,697 | 0 | 135 | 33.09 |
| SYN201-1 | 0.31 | 6 | 2,687 | 0 | 134 | 33.09 |
| SYN202-1 | 72.81 | 14 | 146,169 | 0 | 600 | 0.02 |
| SYN203-1 | 0.33 | 11 | 19,970 | 0 | 242 | 10.54 |
| SYN204-1 | 77.86 | 22 | 145,290 | 0 | 730 | 0.02 |
| SYN205-1 | 77.84 | 22 | 145,836 | 0 | 730 | 0.02 |
| SYN206-1 | 0.59 | 10 | 79,319 | 0 | 396 | 2.15 |
| SYN207-1 | 0.77 | 19 | 102,794 | 0 | 406 | 1.32 |
| SYN208-1 | 0.34 | 11 | 19,382 | 0 | 242 | 10.54 |
| SYN209-1 | 0.3 | 7 | 2,777 | 0 | 134 | 33.09 |
| SYN210-1 | 0.31 | 7 | 2,687 | 0 | 134 | 33.09 |
| SYN211-1 | 0.31 | 7 | 2,687 | 0 | 134 | 33.09 |
| SYN212-1 | 0.31 | 7 | 2,687 | 0 | 134 | 33.09 |
| SYN213-1 | 0.34 | 17 | 22,926 | 0 | 258 | 9.64 |
| SYN214-1 | 0.36 | 17 | 21,650 | 0 | 258 | 9.64 |
| SYN215-1 | 0.34 | 17 | 22,924 | 0 | 258 | 9.64 |
| SYN216-1 | 0.34 | 5 | 19,388 | 0 | 242 | 10.54 |
| SYN217-1 | 0.33 | 7 | 19,982 | 0 | 243 | 10.58 |
| SYN218-1 | 0.34 | 5 | 19,426 | 0 | 243 | 10.57 |
| SYN219-1 | 0.34 | 10 | 20,979 | 0 | 248 | 10.03 |
| SYN220-1 | 0.31 | 6 | 2,735 | 0 | 136 | 33.17 |
| SYN221-1 | 0.31 | 6 | 2,748 | 0 | 137 | 33.17 |
| SYN222-1 | 0.31 | 6 | 2,745 | 0 | 137 | 33.17 |
| SYN223-1 | 0.3 | 6 | 2,827 | 0 | 136 | 33.17 |
| SYN224-1 | 0.31 | 6 | 2,739 | 0 | 136 | 33.17 |
| SYN225-1 | 0.33 | 6 | 20,000 | 0 | 244 | 10.60 |
| SYN226-1 | 0.31 | 6 | 2,735 | 0 | 136 | 33.17 |
| SYN227-1 | 0.34 | 6 | 19,406 | 0 | 244 | 10.61 |
| SYN228-1 | 0.31 | 5 | 2,687 | 0 | 134 | 33.09 |
| SYN229-1 | 0.3 | 6 | 2,840 | 0 | 137 | 33.17 |
| SYN230-1 | 0.31 | 6 | 2,748 | 0 | 137 | 33.17 |
| SYN231-1 | 0.31 | 6 | 2,745 | 0 | 137 | 33.17 |
| SYN232-1 | 0.31 | 6 | 2,745 | 0 | 137 | 33.17 |
| SYN233-1 | 0.31 | 6 | 2,735 | 0 | 136 | 33.17 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| SYN234-1 | 0.34 | 6 | 19,418 | 0 | 245 | 10.65 |
| SYN235-1 | 0.33 | 6 | 20,009 | 0 | 246 | 10.69 |
| SYN236-1 | 0.33 | 6 | 2,570 | 0 | 136 | 33.17 |
| SYN237-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN238-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN239-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN240-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN241-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN242-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN243-1 | 0.31 | 3 | 2,729 | 0 | 136 | 33.25 |
| SYN244-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN245-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN246-1 | 0.31 | 3 | 2,716 | 0 | 136 | 33.17 |
| SYN247-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN248-1 | 0.36 | 6 | 18,497 | 0 | 244 | 10.52 |
| SYN249-1 | 0.34 | 6 | 19,626 | 0 | 245 | 10.51 |
| SYN250-1 | 0.94 | 14 | 111,898 | 0 | 432 | 1.16 |
| SYN251-1 | 0.3 | 3 | 2,823 | 0 | 136 | 33.25 |
| SYN252-1 | 77.86 | 35 | 145,363 | 0 | 733 | 0.02 |
| SYN253-1 | 77.84 | 34 | 145,850 | 0 | 732 | 0.02 |
| SYN254-1 | 77.83 | 29 | 145,476 | 0 | 731 | 0.02 |
| SYN255-1 | 0.3 | 3 | 97 | 0 | 46 | 100.00 |
| SYN256-1 | 0.3 | 3 | 100 | 0 | 46 | 100.00 |
| SYN257-1 | 0.22 | 1 | 0 | 0 | 24 | 0.00 |
| SYN258-1 | 0.33 | 2 | 3 | 0 | 40 | 100.00 |
| SYN259-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN260-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN261-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN262-1 | 0.31 | 5 | 4,084 | 0 | 166 | 29.02 |
| SYN263-1 | 0.31 | 6 | 3,771 | 0 | 162 | 29.62 |
| SYN264-1 | 0.34 | 6 | 19,500 | 0 | 244 | 10.57 |
| SYN265-1 | 0.3 | 4 | 1,377 | 0 | 91 | 61.49 |
| SYN266-1 | 0.36 | 12 | 23,056 | 0 | 279 | 9.90 |
| SYN267-1 | 0.3 | 5 | 2,067 | 0 | 108 | 38.99 |
| SYN268-1 | 0.3 | 5 | 2,063 | 0 | 108 | 38.99 |
| SYN269-1 | 76.45 | 31 | 145,675 | 0 | 632 | 0.02 |
| SYN270-1 | 0.42 | 11 | 45,469 | 0 | 367 | 5.68 |
| SYN271-1 | 76.47 | 31 | 145,802 | 0 | 637 | 0.02 |
| SYN272-1 | 0.91 | 19 | 113,262 | 0 | 427 | 1.15 |
| SYN273-1 | 76.8 | 11 | 145,183 | 0 | 669 | 0.02 |
| SYN274-1 | 0.2 | 1 | 0 | 0 | 4 | 0.00 |
| SYN275-1 | 0.22 | 1 | 0 | 0 | 4 | 0.00 |
| SYN276-1 | 0.23 | 1 | 0 | 0 | 26 | 0.00 |
| SYN277-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN278-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN279-1 | 0.3 | 3 | 2,783 | 0 | 134 | 33.09 |
| SYN280-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN281-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN282-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN283-1 | 0.3 | 3 | 2,777 | 0 | 134 | 33.09 |
| SYN284-1 | 0.31 | 3 | 2,687 | 0 | 134 | 33.09 |
| SYN285-1 | 0.33 | 6 | 19,991 | 0 | 244 | 10.60 |
| SYN286-1 | 0.3 | 3 | 2,787 | 0 | 135 | 33.09 |
| SYN287-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN288-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN289-1 | 0.3 | 3 | 2,777 | 0 | 134 | 33.09 |
| SYN290-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| SYN291-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN292-1 | 0.31 | 2 | 3 | 0 | 40 | 100.00 |
| SYN293-1 | 0.34 | 5 | 19,453 | 0 | 245 | 10.62 |
| SYN294-1 | 0.34 | 5 | 19,465 | 0 | 246 | 10.65 |
| SYN295-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN296-1 | 0.31 | 3 | 2,687 | 0 | 134 | 33.09 |
| SYN297-1 | 0.3 | 2 | 3 | 0 | 40 | 100.00 |
| SYN298-1 | 0.59 | 10 | 79,410 | 0 | 396 | 2.14 |
| SYN299-1 | 0.58 | 10 | 80,731 | 0 | 396 | 2.14 |
| SYN300-1 | 0.59 | 10 | 79,319 | 0 | 396 | 2.15 |
| SYN301-1 | 0.31 | 7 | 2,687 | 0 | 134 | 33.09 |
| SYN310-1 | 0.28 | 19 | 25,182 | 0 | 2157 | 30.92 |
| SYN312-1 | 0.86 | 21 | 107,037 | 0 | 17871 | 19.45 |
| SYN315-1 | 0.17 | 5 | 200 | 0 | 3 | 60.00 |
| SYN318-1 | 0.19 | 3 | 21 | 0 | 4 | 100.00 |
| SYN319-1 | 0.17 | 5 | 147 | 0 | 6 | 60.00 |
| SYN321-1 | 0.17 | 5 | 200 | 0 | 3 | 60.00 |
| SYN323-1 | 0.17 | 5 | 294 | 0 | 4 | 36.36 |
| SYN325-1 | 0.17 | 3 | 35 | 0 | 3 | 100.00 |
| SYN326-1 | 0.17 | 4 | 35 | 0 | 5 | 100.00 |
| SYN327-1 | 0.17 | 6 | 800 | 0 | 6 | 24.00 |
| SYN328-1 | 0.33 | 57 | 171,045 | 0 | 10 | 2.97 |
| SYN331-1 | 0.17 | 5 | 576 | 0 | 6 | 15.38 |
| SYN333-1 | 0.17 | 6 | 100 | 0 | 3 | 60.00 |
| SYN336-1 | 0.17 | 1 | 0 | 0 | 5 | 0.00 |
| SYN338-1 | 0.17 | 1 | 0 | 0 | 3 | 0.00 |
| SYN339-1 | 0.17 | 1 | 0 | 0 | 2 | 0.00 |
| SYN340-1 | 0.19 | 1 | 0 | 0 | 2 | 0.00 |
| SYN341-1 | 0.17 | 1 | 0 | 0 | 2 | 0.00 |
| SYN343-1 | 0.17 | 3 | 53 | 0 | 2 | 66.67 |
| SYN346-1 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| SYN350-1 | 0.17 | 24 | 13,294 | 0 | 6 | 4.32 |
| SYN354-1 | 0.16 | 9 | 444 | 0 | 5 | 71.43 |
| SYN554-1 | 0.22 | 16 | 47,636 | 0 | 38 | 0.42 |
| SYN555-1 | 0.27 | 5 | 263 | 0 | 14 | 43.75 |
| SYN558-1 | 18.11 | 18 | 274,247 | 0 | 162 | 0.01 |
| SYN563-1 | 0.25 | 14 | 25,924 | 0 | 425 | 9.01 |
| SYN566-1 | 0.28 | 15 | 70,525 | 0 | 98 | 0.68 |
| SYN570-1 | 0.19 | 12 | 3,989 | 0 | 47 | 10.00 |
| SYN574-1 | 72.17 | 25 | 246,273 | 0 | 96 | 0.00 |
| SYN575-1 | 72.17 | 25 | 247,790 | 0 | 96 | 0.00 |
| SYN578-1 | 11.73 | 57 | 142,362 | 0 | 542 | 0.04 |
| SYN579-1 | 11.78 | 57 | 141,758 | 0 | 542 | 0.04 |
| SYN580-1 | 18.03 | 13 | 231,897 | 0 | 108 | 0.00 |
| SYN581-1 | 0.27 | 9 | 40,052 | 0 | 1038 | 11.25 |
| SYN582-1 | 0.28 | 11 | 38,621 | 0 | 1038 | 11.25 |
| SYN583-1 | 0.3 | 11 | 56,687 | 0 | 1683 | 11.45 |
| SYN585-1 | 0.27 | 6 | 50,919 | 0 | 490 | 3.66 |
| SYN590-1 | 0.23 | 13 | 34,965 | 0 | 259 | 5.38 |
| SYN591-1 | 12.25 | 21 | 237,145 | 0 | 284 | 0.01 |
| SYN592-1 | 12.12 | 17 | 239,689 | 0 | 284 | 0.01 |
| SYN618-1 | 0.22 | 14 | 6,818 | 0 | 98 | 13.46 |
| SYN621-1 | 0.8 | 6 | 19,945 | 0 | 8970 | 56.64 |
| SYN624-1 | 2.11 | 7 | 23,464 | 0 | 24111 | 48.88 |
| SYN626-1 | 0.69 | 8 | 4,107 | 0 | 351 | 18.78 |
| SYN627-1 | 18.61 | 15 | 20,053 | 0 | 4446 | 1.23 |
| SYN631-1 | 74.94 | 13 | 213,605 | 0.5 | 6224 | 0.04 |

| Theorem | CPU time [secs] | Length of proof [steps] | Inference Rate [clauses/sec] | Rating | Number of retained unit clauses | % unit retained to total generated |
|---|---|---|---|---|---|---|
| SYN653-1 | 9.73 | 18 | 73,312 | 0.25 | 385 | 0.10 |
| SYN654-1 | 8.77 | 15 | 75,678 | 0 | 350 | 0.10 |
| SYN655-1 | 8.8 | 15 | 75,420 | 0 | 350 | 0.10 |
| SYN721-1 | 0.14 | 4 | 64 | 0 | 5 | 100.00 |
| SYN724-1 | 0.14 | 11 | 2,036 | 0 | 5 | 12.50 |
| SYN726-1 | 0.78 | 46 | 294,846 | 0 | 29 | 0.07 |
| SYN727-1 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| SYN728-1 | 0.31 | 10 | 303 | 0 | 6 | 33.33 |
| SYN729-1 | 0.59 | 6 | 2,273 | 0 | 826 | 62.48 |
| SYN731-1 | 0.17 | 1 | 0 | 0 | 2 | 0.00 |
| TOP001-2 | 42.08 | 16 | 269,821 | 0.12 | 22 | 0.00 |
| TOP002-2 | 0.17 | 2 | 6 | 0 | 3 | 100.00 |
| TOP004-1 | 0.3 | 1 | 0 | 0 | 3 | 0.00 |
| TOP004-2 | 0.17 | 1 | 0 | 0 | 3 | 0.00 |

**Remark:** Notice that the number of retained unit clauses is much less than the number of generated unit clauses. This is an indication that it may not be necessary to have extremely large tables to store unit clauses. In fact, we found that most of the time, the size of a table of 50,000 entries is enough to store distinct unit clauses that conform to the limits (length of a clause, max term depth, etc.) assigned by the user.

# Bibliography


[Albert et al. 1993]

**Albert, Luc**, and **Casas, Rafael**, and **Fages, Francois**, "Average-case analysis of unification algorithms", *Theoretical Computer Science*, vol. 113, (pp. 3-34), 1993.

[Amble 1987]

**Amble, Tore**, *Logic Programming and Knowledge Engineering*, Great Britain: Addison-Wesley, 1987.

[Andrews 1968]

**Andrews, Peter B.**, "Resolution with Merging", *Journal of the ACM (JACM)* vol. 15, issue 3, (pp. 367-381), July 1968.

[Anderson & Bledsoe 1970]

**Anderson, Robert** and **Bledsoe, W. W.**, "A Linear Format for Resolution with merging and a New Technique for Establishing Completeness", *Journal of the Association for Computing Machinery (JACM)* vol. 17, issue 3, (pp. 525-534), July 1970.

[Astrachan 1992]

**Astrachan, O.L.**, *Investigations in Model Elimination Theorem Proving*, Ph.D. thesis, Duke University, 1992.

[Astrachan & Loveland 1991]

**Astrachan, O.L.** and **Loveland, D.W.**, "METEORs: High Performance Theorem Provers using Model Elimination", *A Festschrift for W.W. Bledsoe*, editor: R. Boyer, Kluwer Academic Publishers, 1991.

[Avenhaus et al. 1995]

**Avenhaus, J.**, and **Denzinger, J.**, and **Fuchs, M.**, "DISCOUNT: a System for Distributed Equational Deduction", *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA 95)*, (Lecture Notes in Computer Science vol. 914), (pp. 397-402), Kaiserslautern, 1995.

[Bach 1986]

**Bach, Maurice J.**, *The Design of the Unix Operating System*, New Jersey: Prentice-Hall Inc., 1986.

[Baumgartner & Furbach 1994]

**Baumgartner P.**, and **Furbach, U.**, "PROTEIN: A PROver aith a Theory Extension Interface", in *Proceedings of the 12th International Conference on Automated Deduction* (CADE-12), editor: A. Bundy, LNAI vol. 812, (pp. 769-773), Springer, Berlin, 1994.

[Benanav 1990]

**Benanav, D.** "Simultaneous Paramodulation", *CADE-10, 10th International Conference on Automated Deduction* (Lecture Notes in Artificial Intelligence, vol. 449), editor: M. Stickel, (pp. 442-455), Springer-Verlag, 1990.

[Bibel 1987]

**Bibel, Wolfgang.**, *Automated Theorem Proving (2nd Edition)*, Vieweg, 1987.

[Borowski & Borwein 1991]

**Borowski, E. J.** and **Borwein, J. M.**, *The HarperCollins Dictionary of Mathematics*, HarperPerennial (A division of HarperCollins Publishers), 1991.

[Boyer & Moore 1998]

**Boyer, Robert S.** and **Moore, Strother J.**, *A Computational Logic Handbook* (*2nd Edition*), Great Britain: Academic Press, 1998.

[Boyer & Yu 1992]

**Boyer, Robert S.** and **Yu, Yuan**, "Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor", *CADE-11 11th Conference on Automated Deduction* (Lecture Notes in Artificial Intelligence, vol. 607), (pp. 416-430), 1992.

[Brand 1975]

**Brand, D.**, "Proving theorems with the modification method", *Society for Industrial and Applied Mathematics* (*SIAM*) (Journal on Computing, vol. 4, number 4), (pp. 412-430), 1975.

[Bratko 1991]

**Bratko, Ivan**, Prolog: Programming for Artificial Intelligence, Addison-Wesley, 1991.

[Carson et al. 1965]

**Carson, Daniel F.**, and **Robinson, George A.**, and **Wos, Lawrence**, "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving", *Journal of Association for Computing Machinery*, vol. 12 no. 4, (pp. 536-541), 1965.

[Chang & Lee 1973]

**Chang, Chin-Liang**, and **Lee, Richard Char-Tung**, *Symbolic Logic and Mechanical Theorem Proving*. US: Academic Press Inc., 1973.

[Chou 1988]

**Chou, Shang-Ching**, *Mechanical Geometry Theorem Proving*, Holland: D. Reidel Publishing, 1988.

[Cormen et al. 1992]

**Cormen, Thomas H.**, and **Leiserson, Charles E.**, and **Rivest, Ronald, L.**, *Introduction to Algorithms*, US: McGraw-Hill, 1992.

[Davis 1983]

**Davis, Martin.**, "The Prehistory and Early History of Automated Deduction", *Automation of Reasoning*, Classical Papers on Computational Logic 1957-1966, vol. 1, (pp. 1-22), editors: Jörg Siekmann and Graham Wrightson, Springer Verlag, 1983.

[Davis 2001]

**Davis, Martin**, "The Early History of Automated Deduction", *Handbook of Automated Reasoning Volume I*, chap. 1, (pp. 3-15), Cambridge, MA: Massachusetts Institute of Technology, 2001.

[Davis & Putnam 1960]

**Davis, Martin**, and **Putnam, Hilary**, "A computing procedure for quantification theory", *Journal of the ACM 7 (3)*, (pp. 201-215), 1960.

[Davis et al. 1962]

**Davis, Martin**, and **Logemann George**, and **Loveland, Donald**, "A machine program for theorem proving", *Communications of the ACM 5*, (pp. 394-397), 1962.

[Degtyarev & Voronkov 2001]

**Degtyarev, Anatoli** and **Voronkov, Andrei**, "Equality Reasoning and Sequent-Based Calculi", *Handbook of Automated Reasoning Volume I*, chap. 10 (pp. 611-706), Cambridge, MA: Massachusetts Institute of Technology, 2001.

[Denzinger et al. 1997]

**Denzinger, J.**, and **Kronenberg, M.**, and **Schulz, S.**, "DISCOUNT – A Distributed and Learning Equational Prover", *Journal of Automated Reasoning*, vol. 18, issue 2, (pp. 189-198), 1997.

[Duvanenko 2004]

**Duvanenko, Victor J.**, "Optimizing for Intel Architecture CPU", pp. 28-32, *Dr. Dobb's Journal*, May 2004.

[Duffy 1991]

**Duffy, David**, *Principles of Automated Theorem Proving*, Great Britain: John Wiley, 1991.

[Fleisig et al. 1974]

**Fleisig, S.**, and **Loveland, D.**, and **Smiley, A.**, and **Yarmash, D.**, "An implementation of the model elimination proof procedure", *Journal of the Association for Computing Machinery*, vol. 21, (pp. 124-139), January 1974.

[Fensel & Schönegge 1997]

**Fensel, D.**, and **Schönegge, A.**, "Using KIV to specify and Verify Architectures of Knowledge-Based Systems.", *Proceedings of the 12th IEEE International Conference on Automated Software Engineering* (ASEC-97), Incline Village, Nevada, November 3-5, 1997.

[Fuchs & Fuchs 1999]

**Fuchs, Dirk**, and **Fuchs, Marc**, "Cooperation between Top-Down and Bottom-Up Theorem Provers", *Journal of Artificial Intelligence Research*, vol. 10, (pp. 169-198), 1999.

[Gaillourdet et al. 2003]

**Gaillourdet, Jean-Marie** , and **Hillenbrand, Thomas**, and **Löchner, Bernd**, and **Spies, Hendrik**, "The New Waldmeister Loop at Work", in CADE-19, *19th International Conference on Automated Deduction*, (LNAI 2741), editor: Franz Baader, (pp.317-321), Springer, 2003.

[Gallier 1986]

**Gallier, Jean H.**, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, New York: Harper & Row, 1986.

[Garey & Johnson 1979]

**Garey, Michael R.** and **Johnson, David D.**, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York: Freeman, 1979.

[Gilmore 1960]

**Gilmore, P.**, "A proof method for quantification theory: its justification and realization", *IBM Journal of Research and Development* vol. 4, (pp. 28-35), 1960. Reprinted in *Automation of Reasoning*, Classical Papers on Computational Logic 1957-1966 vol.1, (pp. 151-158), editors: Jörg Siekmann and Graham Wrightson, Springer Verlag, 1983.

[Harel et al. 2000]

**Harel, David** and **Kozen, Dexter** and **Tiuryn, Jerzy**, *Dynamic Logic*, Cambridge, MA: Massachusetts Institute of Technology, 2000.

[Hillenbrand et al. 1997]

**Hillenbrand Th.**, and **Buch A.**, and **Vogt R.**, and **Löchner, B.**, "Waldmeister: High-Performance Equational Deduction", *Journal of Automated Reasoning* vol.18(2), 1997.

[Hillenbrand & Löchner 2002]

**Hillenbrand Th.** and **Löchner, B.**, "A Phytography of Waldmeister", *AI Communications* vol.15(2-3), (pp. 127-133), 2002.

[Huth & Ryan 2000]

**Huth, Michael R. A.**, and **Ryan, Mark D.**, *Logic in Computer Science: Modelling and reasoning about systems*, Cambridge, UK: Cambridge University, 2000.

[Jeff Ho 1999]

**Jeff Ho, Chuen-Hsuen**, "Completeness of the LELS Inference Rule in Automated Theorem Proving", *Journal of Information Science and Engineering*, vol. 15, (pp. 153-164), 1999.

[Kaufmann et al. 2000]

**Kaufmann, Matt**, and **Manolios, Panagiotis**, and **Moore, J. Strother** (editors), *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, June, 2000.

[Kientzle 2004]

**Kientzle, Tim**, "Optimization Techniques", (pp. 22-26), *Dr. Dobb's Journal*, May 2004.

[Korf 1985]

**Korf, Richard E.**, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence*, vol. 27, (pp. 97-109), 1985.

[Lassez et al. 1988]

**Lassez, J-L.**, and **Maher, M. J.**, and **Marriott, K.**, "Unification Revisited", *Computer Science*, (pp. 1-44), 1988.

[Luckham 1970]

**Luckham, D.**, "Refinement theorems in Resolution Theory", *Proc. IRIA 1968 Symp. On Automatic Demonstration*, (Lecture Notes in Mathematics, vol. 125), (pp. 163-190), Springer-Verlag, 1970.

[Letz et al. 1992]

**Letz, R.**, and **Bayerl, S.**, and **Schumann, J.**, and **W. Bibel**, "SETHEO: a High-Performance Theorem Prover", *Journal of Automated Reasoning*, vol. 8(2), (pp. 183-212), 1992.

[Letz & Stenz 2001]

**Letz, Reinhold** and **Stenz, Gernot**, (editors), "Model Elimination and Connection Tableau Procedures", *Handbook of Automated Reasoning Volume II*, chap. 28, Cambridge, MA: Massachusetts Institute of Technology, 2001.

[Levesque & Brachman 2004]

**Levesque, Hector J.** and **Brachman, Ronald J.**, *Knowledge Representation and Reasoning*, Morgan Kaufmann, 2004.

[Levy & Newborn 1991]

**Levy, David**, and **Newborn, Monty**, *How Computers Play Chess*, New York: W.H. Freeman and Company, 1991.

[Loveland 1968]

**Loveland, D. W.**, "Mechanical theorem proving by model elimination", *Journal of the Association for Computing Machinery*, vol. 15(2), (pp. 236-251), April 1968.

[Loveland 1969]

**Loveland, D. W.**, "A simplified format for the model elimination procedure", *Journal of the Association for Computing Machinery*, vol. 16(3), (pp. 349-363), July 1969.

[Loveland 1978]

**Loveland, Donald W.**, *Automated Theorem Proving: a logical basis*, Hungary: North-Holland, 1978.

[Luger & Stubblefield 1998]

**Luger, George F.**, and **Stubblefield, William A.**, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving* (third edition), Reading, MA: Addison-Wesley, 1998.

[McCune 2003]

**McCune, William M.**, *Otter version 3.3 Reference Manual*, http://www-unix.mcs.anl.gov/AR/otter/, 2003.

[McKusick & Neville-Neil 2004]

**McKusick, Marshall Kirk** and **Neville-Neil, George**, *The Design and Implementation of the FreeBSD Operating System*, Addison Wesley Professional, 2004.

[Newborn 2001]

**Newborn, Monty**, *Automated Theorem Proving: Theory and Practice*, New York: Springer-Verlag, 2001.

[Newborn & Wang 2004]

**Newborn, Monty** and **Wang, Zongyan**, "Octopus: Combining Learning and Parallel Search", *Journal of Automated Reasoning*, 2004.

[Newell et al. 1957]

**Newell, A.**, and **Shaw, J.**, and **Simon, H.**, "Empirical explorations with the logic theory machine", *Proc. West. Joint Comp. Conf.*, (pp. 218-239), 1957.

[Nieuwenhuis et al. 2001]

**Nieuwenhuis, Robert**, and **Hillenbrand, Thomas**, and **Riazanov, Alexandre**, and **Voronkov, Andrei**, "On the Evaluation of Indexing Techniques for Theorem Proving", *Proceedings of the First International Joint Conference on Automated Reasoning* (Lecture Notes in Artificial Intelligence, vol. 2083), editors: R. Goré, A. Leitsch, and T. Nipkow, (pp. 257-271), Springer-Verlag, 2001.

[Nieuwenhuis & Rubio 2001]

**Nieuwenhuis, Robert** and **Rubio, Albert**, "Paramodulation-Based Theorem Proving", *Handbook of Automated Reasoning Volume I*, chap. 7 (pp. 371-444), Cambridge, MA: Massachusetts Institute of Technology, 2001.

[Nilsson 1998]

**Nilsson, Nils J.**, *Artificial Intelligence: A New Synthesis*, San Francisco: Morgan Kaufmann, 1998.

[Nilsson & Maluszynski 2000]

**Nilsson, Ulf** and **Maluszynski, Jan**, *Logic Programming and Prolog* (*2nd edition*), John Wiley & Sons Ltd., 2000.

[Paterson & Wegman 1978]

**Paterson, M. S.**, and **Wegman, M. N.**, "Linear Unification", *Journal of Computer and System Sciences*, vol. 16, (pp. 158-167), 1978.

[Plaisted & Zhu 1999]

**Plaisted, David A.** and **Zhu, Yunshan**, *The Efficiency of Theorem Proving Strategies: A Comparative and Asymptotic Analysis* (*2nd edition*), Germany: Vieweg, 1999.

[Prawitz & Voghera 1960]

**Prawitz, D.**, and **Prawitz, H.**, and **Voghera, N.**, "A mechanical proof procedure and its realization in an electronic computer", *Journal of the ACM*, vol. 7(2), (pp. 102-128). Reprinted in *Automation of Reasoning*, Classical Papers on Computational Logic 1957-1966 vol. 1, (pp. 202-228), editors: Jörg Siekmann and Graham Wrightson, Springer Verlag, 1983.

[Quaife 1989]

**Quaife, Art**, "Automated Development of Tarski's Geometry", *Journal of Automated Reasoning*, vol. 5, (pp. 97-118), 1989.

[Randell et al. 1992]

**Randell, D. A.**, and **Cohn, A. G.**, and **Cui, Z.**, "Computing Transitivity Tables: A Challenge for Automated Theorem Provers", *CADE-11 11th Conference on Automated Deduction* (Lecture Notes in Artificial Intelligence 607), (pp. 786-790), 1992.

[Riazanov 2003]

**Riazanov, Alexandre**, *Implementing an efficient theorem prover*, Ph.D. dissertation, Dept. of Computer Science, University of Manchester, 2003.

[Riazanov & Voronkov 2000]

**Riazanov, A.** and **Voronkov, A.**, "Limited Resource Strategy in Resolution Theorem Proving", *Preprint CSPP-7*, Department of Computer Science, University of Manchester, October 2000.

[Riazanov & Voronkov 2002]

**Riazanov, A.** and **Voronkov, A.**, "The Design and Implementation of Vampire", *AI Communications*, vol. 15(2-3), (pp. 91-110), 2002.

[Robinson 1965]

**Robinson, J.**, "A machine oriented logic based on the resolution principle", *Journal of Association Computing Machinery*, vol. 12, (pp. 23-41), 1965.

[Robinson 1983]

**Robinson, J. A.**, "Automatic Deduction with Hyper-Resolution", *Automation of Reasoning*, Classical Papers on Computational Logic 1957-1966 vol. 1, (pp. 416-423), editors Jörg Siekmann and Graham Wrightson, Springer Verlag, 1983. Originally appeared in *International Journal of Computer Mathematics* vol. 1, (pp. 227-234)

[Robinson & Voronkov (1) 2001]

**Robinson, Alan** and **Voronkov, Andrei**, (editors) *Handbook of Automated Reasoning Volume I*, Cambridge, MA: Massachusetts Institute of Technology, 2001.

[Robinson & Voronkov (2) 2001]

**Robinson, Alan** and **Voronkov, Andrei**, (editors) *Handbook of Automated Reasoning Volume II*, Cambridge, MA: Massachusetts Institute of Technology, 2001.

[Robinson & Wos 1969-1]

**Robinson, G. A.** and **Wos, L. T.**, "Paramodulation and theorem proving in first order theories with equality", *Machine Intelligence*, vol. 4, (pp. 133-150), editors R. Meltzer and D. Michie, Edinburgh: Edinburgh University Press, 1969. Reprinted, in *Automation of Reasoning*, vol. 2, (pp. 298-313), editors: J. Siekmann and G. Wrightson, Berlin: Springer-Verlag, 1983.

[Robinson & Wos 1969-2]

**Robinson, G. A.** and **Wos, L. T.**, "Completeness of Paramodulation", *Journal of Symbolic Logic*, vol. 34, (p. 160), 1969.

[Rosen 2000]

**Rosen, Kenneth H.**, *Elementary Number Theory and Its Applications*, US: Addison-Wesley, 2000.

[Rosch 2003]

**Rosch, Winn L.**, *Hardware Bible* (*6th edition*), Que Publishing, 2003.

[Rupley & Clyman 1995]

**Rupley, Sebastian** and **Clyman, John**, "P6:The Next Step?", *PC Magazine*, vol. 14, no. 15, (pp. 102-137), September 12, 1995.

[Schulz 2000]

**Schulz, Stephan**, *Learning Search Control Knowledge for Equational Deduction*, Ph.D. dissertation, Institut für Informatik der Technischen Universität München, 2000.

[Schulz 2002]

**Schulz, Stephan**, "E: A Brainiac Theorem Prover", AI Communications vol. 15(2-3), (pp. 111-126), 2002.

[Schulz 2004]

**Schulz, Stephan**, "Simple and Efficient Clause Subsumption with Feature Vector Indexing", *Empirically Successful First-Order Reasoning* (*ESFOR*) held at *The 2nd International Joint Conference on Automated Reasoning* (*IJCAR*), Ireland, 2004.

[Schumann 1994]

**Schumann, Johann M. Ph.**, "DELTA – A bottom-up Preprocessor for Top-Down Theorem Provers – Systems Abstract", CADE 12 - *12th International Conference on Automated Deduction*, (Lecture Notes in Computer Science 814), editor: Alan Bundy, (pp. 774-777), France, 1994.

[Sekar et al. 2001]

**Sekar R.**, **Ramakrishnan, I. V.** and **Voronkov, Andrei**, "Term Indexing", *Handbook of Automated Reasoning Volume II*, chap. 26 (pp. 1853-1964), Cambridge, MA: Massachusetts Institute of Technology, 2001.

[Shankar 1997]

**Shankar, N.**, *Metamathematics, Machines, and Gödel's Proof*, Cambridge, UK: Cambridge, 1997.

[Silberschatz et al. 2001]

**Silberschatz, Abraham**, and **Galvin, Peter Baer**, and **Gagne, Greg**, *Operating Systems Concepts* (*6th Edition*), John Wiley & Sons Inc., 2001.

[Skiena 1998]

**Skiena, Steven S.**, *The Algorithm Design Manual*, New York: Springer-Verlag, 1998.

[Smullyan 1995]

**Smullyan, Raymond M.**, *First-Order Logic*, New York: Dover, 1995.

[Socher-Ambrosius & Johann 1997]

**Socher-Ambrosius**, **Rolf** and **Johann, Patricia**, *Deduction Systems*, New York: Springer-Verlag, 1997.

[Spelt & Even 1998]

**Spelt, David**, and **Even, Susan**, "An Engineering Approach to Atomic Transaction Verification: Use of a Simple Object Model to Achieve Semantics-based Reasoning at Compile-time.", *Technical Report, Centre for Telematics and Information Technology* (CTIT), University of Twente, Enschede, The Netherlands, Sept. 1998.

[Stallings 2004]

**Stallings, William**, Operating Systems: Internals and Design Principles ($5^{th}$ Edition), Prentice-Hall Inc., 2004.

[Steele 1990]

**Steele, Guy L.**, *Common Lisp the language* (*$2^{nd}$ edition*), Digital Press, 1990.

[Stickel 1984]

**Stickel, M.E.**, "A Prolog technology theorem prover", *New Generation Computing*, vol. 2(4), (pp. 371-383), 1984.

[Stickel 1988]

**Stickel, Mark E.**, "A Prolog technology theorem prover: implementation by an extended Prolog compiler", *Journal of Automated Reasoning* vol. 4, 4, (pp. 353-380), 1988.

[Stickel 1990]

**Stickel, M.**, "A Prolog Technology Theorem Prover", *Proceedings CADE* vol. 10, *Lecture Notes in Computer Science* 449, (pp. 674-675), Springer, 1990.

[Stickel 1992]

**Stickel, Mark E.**, "A Prolog technology theorem prover: a new exposition and implementation in Prolog", *Theoretical Computer Science* vol. 104, (pp. 109-128), 1992.

[Stickel & Tyson 1985]

**Stickel, M.** and **Tyson, M.**, "An Analysis of Consecutively Bounded Depth-First Search with Applications in Automated Deduction", *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, (pp. 1073-1075), San Francisco: Morgan Kaufmann, 1985.

[Sutcliffe & Tabada 1991]

**Sutcliffe, G.**, and **Suttner, C.**, "Compulsory Reduction in Linear Derivation Systems", *Artificial Intelligence* vol. 50, editor Bibel W., (pp. 131-132), Elsevier, Amsterdam, The Netherlands, 1991.

[Sutcliffe & Suttner 2004]

**Sutcliffe, G.**, and **Suttner, C.**, "The CADE-19 ATP System Competition", *Artificial Intelligence Communications*, vol. 17 issue 3, (pp. 103-110), 2004.

[Sutcliffe et al. 1994]

**Sutcliffe, G.**, and **Suttner, C.**, and **Yemenis, T.**, "The TPTP Problem Library", in *CADE-12, 12th International Conference on Automated Deduction*, (LNAI 814), editor A. Bundy, (pp. 252-266), Springer-Verlag, 1994.

[Sutcliffe et al. 2001]

**Sutcliffe, Geoff**, and **Fuchs, Matthias**, and **Suttner, Christian**, "Progress in Automated Theorem Proving, 1997-1999", *Workshop on empirical Methods in Artificial Intelligence*, (pp. 53-60), held at *17th International Joint Conference on Artificial Intelligence* (Seattle, USA), 2001.

[Tammet 1997]

**Tammet, Tanel**, "Gandalf", *Journal of Automated Reasoning*, vol. 18(2), (pp. 199-204), 1997.

[Turban & Aronson 1998]

**Turban, Efraim** and **Aronson, Jay E.**, *Decision Support Systems and Intelligent Systems* (*5th Edition*), New Jersey: Prentice-Hall, 1998.

[Veroff & Wos 1992]

**Verrof, R.** and **Wos, L.**, "The Linked Inference Principle I: The Formal Treatment", *Journal of Automated Reasoning*, vol. 8, no. 2, (pp. 213-274), 1992.

[Veroff 1997]

**Veroff, Robert** (editor), *Automated Reasoning and Its Applications*, Cambridge, MA: Massachusetts Institute of Technology, 1997.

[Voronkov 1995]

**Voronkov, A.**, "The Anatomy of Vampire: implementing Bottom-Up Procedures with Code Trees", *Journal of Automated Reasoning*, vol. 15(2), (pp. 237-265), 1995.

[Wang 1963]

**Wang, Hao**, "Mechanical mathematics and inferential analysis", *Computer Programming and Formal Systems*, editors: Braffort and Hirschberg, North-Holland, (pp. 1-20), 1963.

[Weidenbach et al. 1999]

**Weidenbach, C.**, and **Afshordel, B.**, and **Brahm, U.**, and **Cohrs, C.**, and **Engel, T.**, and **Keen, E.**, and **Theobalt, C.**, and **Topic, D.**, "System Description: SPASS Version 1.0.0", *Automated Deduction CADE 16*,

editor: H. Ganzinger, *International Conference on Automated Deduction* (Lecture Notes in Artificial Intelligence), (pp. 378-382), Trento, Italy, 1999.

[Wilson 2004]

**Wilson, Matthew**, "C/C++ Compiler Optimization: Focusing on speed", (pp. 16-21), *Dr. Dobb's Journal*, May 2004.

[Wos 1993]

**Wos, L.**, "Automated Reasoning Answers Open Questions", *Notices of the AMS 5*, no. 1, (pp. 15-26) , January 1993.

[Wos 1996]

**Wos, Larry**, The Automation of Reasoning: An Experimenter's Notebook with OTTER Tutorial, New York: Academic Press, 1996.

[Wos & McCune 1991]

**Wos, L.**, **McCune, W.**, "Automated Theorem Proving and Logic Programming: A Natural Symbiosis", *Logic Programming* 11, no. 1, (pp. 1-53), July 1991.

[Wos et al. 1964]

**Wos, L.**, and **Carson, D.**, and **Robinson, G.**, "The unit preference strategy in theorem proving", IFIPS *Proceedings* 1964 *Fall Joint Comp. Conf.* vol. 26, (pp. 616-621), Washington D.C.: Spartan Books, 1964. Reprinted in *Automation of Reasoning*, Classical Papers on Computational Logic 1957-1966 vol. 1, (pp. 387-393), editors: Jörg Siekmann and Graham Wrightson, Springer Verlag, 1983.

[Wos et al. 1965]

**Wos, L.**, and **Robinson, G.**, and **Carson, D.**, "Efficiency and completeness of the set of support strategy in theorem proving", *Journal of Association for Computing Machinery*, vol. 12, no. 4, (pp. 536-541), 1965.

[Wos et al. 1967]

**Wos, L. T.**, and **Robinson, G. A.**, and **Carson, D. F.**, and **Shalla L.**, "The concept of demodulation in Theorem Proving", *Journal of Association for Computing Machinery*, vol. 14, no. 4, (pp. 698-709), 1967.

[Wos et al. 1980]

**Wos, L.**, and **Overbeek, R.**, and **Henschen, L.**, "Hyperparamodulation: A Refinement of Paramodulation", *Proceedings of the Fifth Conference on Automated Deduction* (CADE-5) (Lecture Notes in Computer Science, vol. 87), editors: Robert Kowalski and Wolfgang Bibel, (pp. 208-210), New York: Springer-Verlag, 1980.

[Wos et al. 1984]

**Wos, L.**, and **Veroff, R.**, and **Smith, B.**, and **McCune, W.**, "The Linked Inference Principle, II: The User's Viewpoint", in *Proceedings of the Seventh International Conference on Automated Deduction*, vol. 170, (pp. 316-332), *Lecture Notes in Computer Science*, ed. R. E. Shostak, New York: Springer-Verlag, 1984.

[Wos et al. 1991]

**Wos, L.**, and **Overbeek, R.**, and **Lusk, E.**, "Subsumption, a Sometimes Undervalued Procedure", *Festschrift for J.A. Robinson*, ed. J.-L. Lassez and Gordon Plotkin, (pp. 3-40), Cambridge MA: MIT Press, 1991.

[Wos et al. 1992]

**Wos, Larry**, and **Overbeek, Ross**, and **Lusk, Ewing**, and **Boyle, Jim**, *Automated Reasoning: Introduction and Applications* (*2nd Edition*), New York: McGraw-Hill, 1992.

# Bibliography (Web Sites)

The following web sites were valid at the time of submission of this thesis (May 2005).


[CARINE site] http://www.atpcarine.com

[CASC site] http://www.cs.miami.edu/~tptp/CASC/

[E site] http://www.eprover.org
http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html

[Gandalf site] http://deepthought.ttu.ee/it/gandalf/

[Intel site] http://www.intel.com

[Otter site] http://www-unix.mcs.anl.gov/AR/otter/

[ORA Canada site] Odyssey Research Associates Canada (Ottawa, Ontario)
http://www.ora.on.ca/biblio/biblio-welcome.html

[Sutcliffe site] http://www.cs.miami.edu/~tptp/OverviewOfATP.html

[TPTP site] http://www.cs.miami.edu/~TPTP

[Waldmeister site] http://www.waldmeister.org