Imperial College London
Department of Computing

A Theorem Prover for Equality using Lemmas

by

Rosa M. Gutiérrez Escudero

Submitted in partial fulfilment of the requirements for the
MSc Degree in Advanced Computing of Imperial College London

September 2010

Abstract

Equality handling has been traditionally one of the weaknesses of tableau-based theorem provers in general, from which those based on the Connection Tableau Calculus are not an exception. The Intermediate Lemma Extension is based on a simple extension to Logic Programming in which positive subgoals are skipped and all clauses treated as Horn clauses, allowing to derive clauses consisting of only positive literals, called *lemmas*. The purpose of the extension is to replace a large search space dedicated to find a deep, closed tableau by smaller search spaces that lead to a series of more shallow tableaux. Although its ultimate goal is proof finding, the lemma generation procedure shares some similarities with another automated deduction task, consequence finding.

In this project we investigate the use of the Intermediate Lemma Extension for Connection Tableau Calculus and how this could help to reduce the search space usually associated with equality problems. Furthermore, we define a sound and complete extension of the calculus with a built-in equality reasoning mechanism based on RUE resolution [13], originally defined for Disconnection Tableau Calculus [28], eliminating the necessity of explicitly using the equality axioms.

Additionally, we have reviewed the Local Failure Caching procedure, a sophisticated pruning technique originally proposed for the SOL Tableau Calculus for consequence finding and have analysed how languages defined for consequences, production fields, could be adapted to lemma derivation. Resulting from that further analysis we have proposed a sound and complete generalisation of the Intermediate Lemma Extension for not only positive subgoals that has turned out to achieve the best performance results.

Acknowledgements

I owe my deepest gratitude to Dr. Krysia Broda for her inestimable guidance during the completion of this project. I could not have hoped for a more encouraging and supportive supervisor. The endless hours that she has spent working with me, her continuous dedication and her infinite enthusiasm have been invaluable to me.

I would like to thank Jorge and Nani for their unconditional support and unlimited patience throughout the years and for always being willing to help when I needed it the most.

Finally, I also acknowledge “la Caixa” Foundation and its fellowship program for postgraduate studies in Great Britain, which has given me the opportunity to make this project possible.

Contents

Chapter 1

Introduction

Automated Theorem Proving (ATP) is a subfield of automated reasoning that aims to prove mathematical theorems by means of a computer program. This involves testing whether a given theorem follows from a set of axioms. They are usually represented using First Order Logic, leading to first order theorem proving, which is the most developed branch of automated theorem proving. Many known systems such as E [41], Setheo [36], Otter [26], ACL2 [1], Prover9/Mace4 [22] or Vampire [54] are of this kind and there exists a large library of test problems for such systems, Thousand of Problems for Theorem Provers (TPTP) [51]. Other widely known systems like HOL [16] or its successor Isabelle [35] consider Higher Order Logics.

There exist several techniques in Automated Theorem Proving, for example, resolution, superposition and term rewriting, DPLL (Davis-Putnam-Logemann-Loveland algorithm), semantic tableaux and connection calculi. To the latter belongs the Connection Tableau Calculus, a theorem proving method situated halfway between clausal tableaux and model elimination. An extension of this method that seems promising is the Intermediate Lemma Extension. It intends to reduce the search space by turning the inference mechanism into one closer to logic programming.

The general goal of this project is to implement a theorem prover based on the Connection Tableau Calculus with the Intermediate Lemma Extension framework. More concretely, we want to investigate how equality reasoning can be incorporated in a theorem prover of this nature. Equalities are not easily dealt with theorem provers based on the tableau method and not much progress has been achieved in this area. Two strategies to address the problem can be proposed:

1. Add a sound and complete built-in mechanism within the calculus that makes it able to handle equality literals separately from other literals. The mechanism has to reflect the nature of the equality relation, implicit in the equality axioms.

2. Treat equality as a regular predicate, adding the equality axioms to the given set of axioms. It is necessary then to improve the method itself by means of pruning techniques, shortcuts or heuristics in order to reduce the search space, since its size is normally considerably increased by the introduction of the axioms.

Some attempts have been made following the first strategy, usually by adapting an existing technique for equality reasoning based on superposition to a tableau setting. Examples of this are equality elimination for semantic tableaux [4, 12] using Brand's E-modification method [10] and paramodulation-based approaches like connection tableaux with lazy paramodulation [34] or the equality linking rules for the disconnection calculus [29, 28].

Much more work has been done towards the second strategy, not necessarily with incorporation of equality reasoning as the ultimate goal, but to improve the general efficiency of particular theorem provers. Some examples mentioned and explained in this report are the shortcuts proposed for Setheo in [36, 37] or the restricted backtracking approach of leanCoP 2.0 [32]. Apart from them, some successful pruning techniques have been proposed for the SOL Tableau Calculus in [18, 21]. This is a calculus based on connection tableaux for consequence finding, a more general purpose than proof or refutation finding.

1.1 Achievements

We have implemented a fully working automated prover for classical first order logic based on the Connection Tableau Calculus with the Intermediate Lemma Extension. It has been implemented as a Prolog technology theorem prover [45, 46], that is, an extension of Prolog equipped with sound unification, a complete inference system and a complete search strategy. Equality is handled in this prover by explicit use of the equality axioms and it incorporates some simple shortcuts to reduce the search space.

In order to study how a built-in equality mechanism could be integrated in the connection tableau calculus with intermediate lemmas, we have reviewed the attempts mentioned above, in [4, 12, 10, 34, 29]. All of them are very close to each other and we have focused on the last one, equality reasoning in the disconnection calculus. In [29] a variant of lazy paramodulation is combined with the disconnection calculus to implement an equality reasoning mechanism resembling RUE resolution. We have adopted this idea and have incorporated two new rules based on RUE resolution plus the symmetry property in the existing rules, in the Connection Tableau Calculus with the Intermediate Lemma Extension. We have proved this extension of the calculus to be sound and complete.

Regarding explicit use of the equality axioms and reduction of the search space, we have separated slightly from the theorem proving field in the direction of the broader automated deduction area, to explore the referred pruning techniques proposed for the SOL Tableau Calculus [18, 21] for consequence finding. The Intermediate Lemma Extension is somehow related to consequence finding and as those pruning methods have been implemented with encouraging results in SOLAR [17], a high-performance automated deduction system, we have adapted and tested them in another version of prover. Moreover, we have taken another feature of SOL Tableau Calculus, the definition of languages that clauses that are part of a consequence must satisfy. We have combined this feature with an idea from hyper-resolution to create a language-based refinement for the intermediate lemma extension and have developed it as a case study. A thorough analysis of the intermediate lemma extension and the necessary conditions for its soundness and completeness has been performed as consequence. The resulting refinement of the calculus is sound and complete and we have investigated some of its possibilities to improve the performance of the prover.

To evaluate the success of our attempts we have relied on the TPTP Library [51], the facto standard set of test problems for ATP systems. We have found the performance of the prover in all its versions to be below our initial expectations. The lack of control in the lemma generation seems to increase considerably the global search space, which makes the prover to be surpassed by other theorem provers based on regular connection calculi. A deeper analysis has revealed that a substantial amount of the search space is dedicated to generate lemmas that are forward subsumed or not used in the final proof. The direction for further improvements of this framework would be to restrict and control the lemma generation in an effective manner.

1.2 Structure of the report

The first two chapters and most of the third chapter of this report are primarily dedicated to preliminaries. In Chapter 2 we settle the basis of the tableau calculus in general and the Connection Tableau Calculus in particular, together with the technology used to implement our prover. After having established the basic background, more sophisticated extensions like the Intermediate Lemma Extension or the SOL Tableau Calculus are introduced in Chapter 3. To complete the introductory information of this report, Chapter 4 lays the foundations of equality reasoning and the Disconnection Tableau Calculus, needed to present the approach based on RUE resolution developed within our prover.

Chapters 5 and 6 are devoted to the prover itself. We describe how the theoretical grounds presented before have been turned into a working automated theorem prover and outline some results from benchmarking. In Chapter 7 the new approach of defining languages for the intermediate lemma extension is thoroughly described and exemplified in the form of a case study. This approach together with the other versions of the prover are evaluated in Chapter 8. This evaluation and the analysis performed for the case study allows us to conclude the report in Chapter 9, with a final assessment of the results achieved and the general deficiencies of our work. Possible directions for future extensions and research paths are proposed also in this chapter. Besides, we provide a small appendix with some guidelines for installing, using and testing the different versions of the prover.

To finish this introductory chapter, in order to maintain this report as self-contained as possible, the next section collects basic definitions and notation issues employed in the remaining chapters.

1.3 Language and notation

Throughout this report a first order clausal language is used. In this section we outline its basis, together with the two different notations employed in subsequent chapters. We assume a basic knowledge of first order syntax and semantics [14, 43]. We use when necessary common logical symbols present in most first-order languages such as quantifiers ($\forall, \exists$), logical connectives ($\neg, \wedge, \vee, \rightarrow$) and the truth constants $true$ or $\top$ and $false$ or $\bot$.

As mentioned, we employ two different notations in the report to distinguish between Prolog code and classical first-order logic, in particular, we consider:

- An infinite set of variable symbols, $\mathcal{X}$, usually denoted by letters at the end of the alphabet, unless otherwise stated. We use lowercase letters when we refer to First Order Logic language, with appropriate subscripts when necessary, $x, y, z, \ldots$, $x_1, x_2, x_3, \ldots$. We use uppercase letters when they refer to Prolog variables, `X,Y,Z,...`, `X1,X2,X3,...`
- For each $n \geq 0$, an infinite set $\mathcal{P}$ of n-ary predicate symbols, which we denote by uppercase letters in first-order logic language, $P_1, P_2, \ldots, Q_1, Q_2, \ldots$ and by lowercase letters in Prolog code, `p1,p2,...,q1,q2,...`. Predicates of arity 0 can be identified with propositional variables.
- For each $n \geq 0$, an infinite set $\mathcal{F}$ of n-ary function symbols, denoted by lowercase letters in both first-order logic and Prolog languages: $f_1, f_2, \ldots, g_1, g_2, \ldots$,`f1,f2,...,g1,g2,...`. Functions of arity 0 represent constant symbols, that are usually denoted by lowercase letters at the beginning of the alphabet, $a, b, c, \ldots$, `a,b,c,...`.

Unless confusion arise, the arity of a predicate or function will not be explicitly indicated in this report. The sets $\mathcal{P}$ and $\mathcal{F}$ suffice to determine a first-order logic language and the tuple $\langle \mathcal{P}, \mathcal{F} \rangle$ is called the *signature* of the language, which can be denoted by $\mathcal{L}\langle \mathcal{P}, \mathcal{F} \rangle$.

Definition 1. (Term). The set of *terms* of $\mathcal{L}\langle\mathcal{P},\mathcal{F}\rangle$ is the smallest set such that:

- Any variable is a term.
- If f is a n-ary function symbol and $t_1,\ldots,t_n$ are terms, $f(t_1,\ldots,t_n)$ is a term.

Definition 2. (Atomic formula). If P is a n-ary predicate symbol and $t_1,\ldots,t_n$ are terms, $P(t_1,\ldots,t_n)$ is an *atomic formula* or *atom.* If the equality symbol is considered part of the logic (as we do from Chapter 4) and t_1,t_2 are terms, then $t_1=t_2$ is also an atom.

Definition 3. (Literal). A *literal* is an atomic formula L or its negation $\neg L$. The *complement* of a literal L is denoted as $\overline{L}$.

Definition 4. (Clause). A *clause* is a set of literals represented as a disjunction $L_1\vee\ldots\vee L_n$ and the *empty clause* is denoted as $\Box$. All variables are implicitly assumed to be universally quantified. Sometimes we may write clauses as sets, $\{L_1,\ldots,L_n\}$, and the empty clause as $\{\}$. An *unit clause* is a clause composed of a single literal.

If an expression (a term, literal or clause) does not contain any variables at all, it is said to be *ground.*

Definition 5. (Horn clause, definite clause). A *Horn clause* is a clause with at most one positive literal and a *definite clause* is a Horn clause with exactly one positive literal. Usually, (definite) Horn clauses are written as $H\leftarrow(B_1\wedge\ldots\wedge B_n)$, (or in Prolog notation, `A :- B`$_1,\ldots,$`B`$_n$`.`) where H is the positive literal (also the conclusion literal or the *head* of the clause) and $B_1,\ldots,B_n$ are negative literals (also called the *body* of the clause). A clause without body (a positive unit clause `H.`) is known as a *fact* and a clause with only negative literals, `:- B`$_1,\ldots,$`B`$_n$`.`, is named a *query.*

Definition 6. (Substitution). A *substitution* is a mapping σ from the set of variables $\mathcal{X}$ of $\mathcal{L}\langle\mathcal{P},\mathcal{F}\rangle$ to its set of terms. We represent a substitution as $\sigma=\{x_1/t_1,\ldots,x_n/t_n\}$, to indicate $\sigma(x_i)=t_i$ for each x_i. A *renaming* is a particular substitution $\{x_1/y_1,\ldots,x_n/y_n\}$ in which all y_i are new or *fresh* variables. If C is a clause, by $C\sigma$ we denote the result of applying the substitution σ to C, that is, of replacing all occurrences of each variable x_i with the term $\sigma(x_i)$ in C.

Definition 7. (Subsumption). We say that a clause C *subsumes* another clause D if there exist a substitution σ such that $C\sigma\subseteq D$. If C subsumes D but D does not subsume C, we say that C *properly subsumes* D.

Definition 8. (Unifier, Most General Unifier). Given a set of literals $\{L_1,\ldots,L_n\}$ (and in general of any sentences in first-order logic), we say that $L_1,\ldots,L_n$ are *unifiable* if there exists a substitution σ such that $L_1\sigma=L_2\sigma=\ldots=L_n\sigma$. The substitution σ is called an *unifier* of $L_1,\ldots,L_n$. An unifier σ is a *most general unifier (mgu)* for $L_1,\ldots,L_n$ if, for all unifiers μ of $L_1,\ldots,L_n$, there exist a substitution θ such that $L_i\mu=(L_i\sigma)\theta$ for all L_i.

1.3.1 Resolution

Resolution is an inference rule proposed by Alan Robinson in 1965 [40] that leads to a refutation theorem-proving technique for set of clauses in first-order logic. Although this project is concerned with a different technique, semantic (clausal) tableau, we need very basic concepts taken from resolution to understand some of the content presented in next chapters. In order to keep this report as self-contained as possible, in this section we outline informally all these basic ideas.

Definition 9. (Binary resolution rule). Given two clauses $C=L_1\vee\ldots\vee L_n$ and $D=K_1\vee\ldots\vee K_m$ such that L_i and $\overline{K_j}$ are unifiable with mgu σ, derive the clause

$$R=(L_1\vee\ldots\vee L_{i-1}\vee L_{i+1}\vee\ldots\vee L_n\vee K_1\vee K_{j-1}\vee K_{j+1}\vee\ldots\vee K_m)\sigma$$

R is named a *resolvent* of C and D, and can be denoted by $\mathcal{R}(C, D)$. Basically three steps are involved in the rule:

1. First unify one literal L_i in C with a literal K_j in D of opposite sign.
2. Apply the substitution to all other literals in C and D.
3. Form the resolvent $\mathcal{R}(C, D) = C \cup D \backslash \{L_i, K_j\}$.

This rule is sufficient to construct a refutation proof for a given set of clauses. The aim of such a *resolution proof* is to derive the empty clause. When coupled with a complete search algorithm, this rule yields a sound and complete refutation technique for first-order logic. There exist several complete search mechanisms, like saturation search (form all possible resolvents using using the initial set S_0 to give S_1, then all clauses that can be formed from S_0 and S_1 together to give S_2, etc) or a linear strategy (generate resolvents using always the previous resolvent as one of the two clauses involved).

Chapter 2

Background

This chapter, dedicated to preliminaries, is divided into three sections. In Section 2.1 we outline the basics of the Semantic Tableaux method for theorem proving and formalise the rules of inference of the Clausal Tableaux proof procedure. In Section 2.2 we describe the fundamentals of the Connection Tableau Calculus, a particular case of Clausal Tableau Calculus and the proof procedure on which our work is primarily based. In Section 2.3 we introduce all concepts behind the approach that we have taken to implement the core and the different versions of our prover, Prolog compilation techniques and Prolog technology theorem provers.

2.1 Theorem Proving with Semantic Tableaux

The objective of the proof procedure based on semantic tableaux is to show by construction that a set of logical sentences has no model, that is, it aims to find a refutation for them. If the problem is stated in term of a set of axioms S and a conclusion C provable from them, this conclusion is negated and a refutation for $S \cup \{\neg C\}$ is sought.

A semantic tableau is a tree in which each node is labeled with a set of formulas. The initial branch contains the input sentences to be refuted and the tree is developed such that each branch corresponds to an alternative in constructing a model of the formulas in the nodes. We say that a branch *contains* a formula α if α belongs to the set of a node in the branch. A branch is *closed* when it can be established that there is no model for it, for example, if it contains both formulas α and $\neg\alpha$, and a tableau is *closed* when all its branches are closed. In the method of semantic tableaux, expansion rules are systematically applied to branches to build a closed tableau for a given set. Those rules are based on the truth tables for the logical connectives occurring in the formula, but we will formalise such rules only for the particular case of clausal tableaux (semantic tableaux in which all sentences are clauses).

We finish this brief introduction with Example 1.

Example 1. The tree shown in Figure 2.1 represents a closed semantic tableau for the set of formulas

$$S = \{\forall x(P(x) \rightarrow Q(x)), \neg Q(a), P(a)\}$$

The structure of the argument for the refutation depicted can be summarised as follows. S has a model M if M satisfies all formulas in S. Therefore, M must assign *true* to $P(a)$ and $\neg Q(a)$, that is $false$ to $Q(a)$. The formula $\forall x(P(x) \rightarrow Q(x))$ is satisfiable if for all x, $M \models P(x) \rightarrow Q(x)$, in particular it must be $M \models P(a) \rightarrow Q(a)$. Then either $\neg P(a)$ (left branch) is true or $Q(a)$ (right

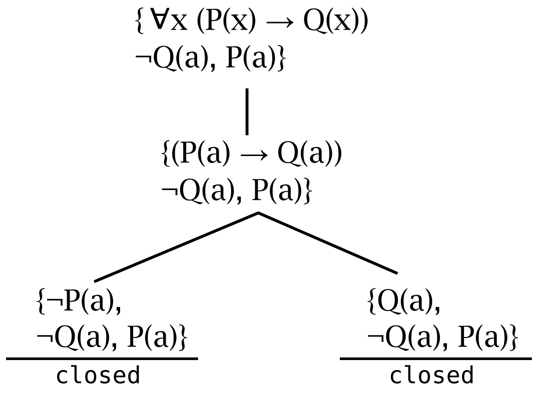


Figure 2.1: A closed semantic tableaux

branch) is true. As we know that $M \models P(a)$, the left branch has to be closed because $\neg P(a)$ cannot be true. Analogously, because $M \models \neg Q(a)$, M cannot assign true to $Q(a)$, so the right branch is closed too and we conclude that the given sentences have no model.

2.1.1 Clausal Tableaux

Rather than proving unsatisfiability for sets of arbitrary first-order formulas, we consider sets in which all sentences are clauses. The reason is that when applied to set of clauses, tableaux methods allow for a number of efficiency improvements. Each first-order logic formula can be converted to an equivalent formula in *conjunctive normal form (CNF)* , i.e., a conjunction of clauses, by means of a transformation based on logical equivalence rules (see Section 5.1.2 for more details). Therefore, in what follows we assume all formulas to be in clausal form.

Definition 10. (Clausal Tableau). A *clausal tableau* T for a set of clauses S is a labeled ordered tree, in which each node except for the root is labeled with a literal. We will identify the nodes with the literals with which they are labeled. If the immediate successors of a node N_j are $K_1, \ldots, K_n$ then there is a substitution σ and a clause $L_1 \vee \ldots \vee L_n \in S$ such that $K_i = L_i\sigma$ for every $1 \leq i \leq n$. $K_1 \vee \ldots \vee K_n$ is called the *tableau clause at* N_i and the tableau clause under the root is called the *start clause.* A tableau is *closed* if every branch contains a literal and its complement.

Definition 11. (Marked and closed tableau). A *marked tableau* is a clausal tableau in which some of the leaves are marked with labels *closed.* The unmarked leaves are called *subgoals.* A tableau is *closed* if all its leaves are marked with *closed.*

We distinguish between *tableau*, static objects that are part of a proof, and *tableau calculus*, a set of inference rules to construct tableau. There are two inference rules in clausal tableau calculus, *extension* and *closure*, plus the single construction step of starting the tableau. They are summarised and formalised in Table 2.1.1 using the tuples $C, S, Branch$. C is the set of subgoals, T is the given set of clauses and $Branch$ is the current branch in the tableau. When a subgoal is marked with *closed*, it is deleted from the current list of subgoals, so that a branch is closed when the current list of subgoals is empty and a tableau is closed when all branches are closed.

The reader should note that we have used the notation $\sigma(K)$ with K being a literal instead of a variable (as it was specified in Section 1.3). Our intention is to differentiate $\sigma(K)$ as a condition of the rule (the result of applying the substitution σ to K) from $K\sigma$ as the effect of performing a step with this rule and actually applying σ to K.

Example 2. Figure 2.2 shows a derivation using Clausal Tableau Calculus for the set of clauses $S = \{P(x) \vee Q(x), R(x) \vee \neg P(a), \neg R(x), \neg Q(a) \vee R(x)\}$

$Start\ (St)\ \dfrac{\{root\}, S, \{\}}{C, S, \{root\}}$ where $C \in S$
$Extension\ (Ext)\ \dfrac{C \cup \{K\}, S, Branch}{C_2, S, Branch \cup \{K\} \;\; \| \;\; C, S, Branch}$ and C_2 is a renamed instance of $C_1 \in S$
$Closure\ (Cl)\ \dfrac{C \cup \{K\}, S, Branch \cup \{L\}}{C\sigma, S, (Branch \cup \{L\})\sigma}$ with $\sigma(K) = \sigma(\overline{L})$

Table 2.1: Rules of the Clausal Tableau Calculus

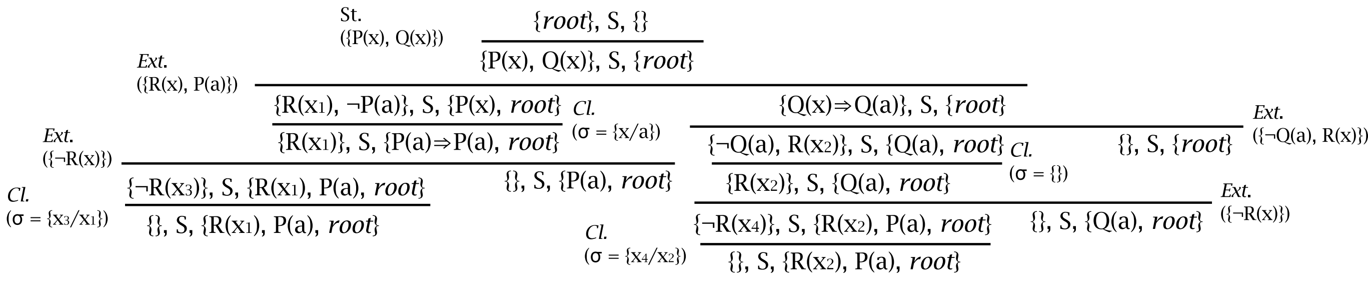


Figure 2.2: Derivation in Clausal Tableau Calculus

Proposition 1. *(Soundness and completeness for the Clausal Tableau Calculus).*

The Clausal Tableau Calculus is sound and complete:

- *If a closed clausal tableau is constructed from a set of clauses S by application of the rules of the calculus, then S is unsatisfiable.*
- *For each unsatisfiable set of clauses S there exists a closed clausal tableau that can be constructed using the rules of the calculus.*

A proof in the clausal tableau calculus can be represented graphically by using a tree (as we did in Figure 2.1 to illustrate semantic tableaux) and we prefer this graphical representation for being more intuitive and clear. To this end and without loss of generality, we can define the previous rules in terms of a the set of clauses S and a marked tableau T for S.

- *Start*: T is a one-node tree with a root only. Choose a clause $L_1 \vee \ldots \vee L_n \in S$ and attach the nodes $L_1, \ldots, L_n$ to the root.
- *Extension rule*: select a subgoal K and a clause $C = L_1 \vee \ldots \vee L_n \in S$. Rename the variables to

make them disjoint from the variables occurring in T and attach new successor nodes $L_1, \ldots, L_n$ to K.

- *Closure rule*: if a subgoal K has an ancestor node L unifiable with $\overline{K}$ with mgu σ, then mark K with *closed* and apply σ to all the literals in T.

Example 3. In Figure 2.3 we present the tableau graphical representation for the derivation in Example 2. We write the correspondences between the tuples $C, S, Branch$ and the branches and nodes in the tree.

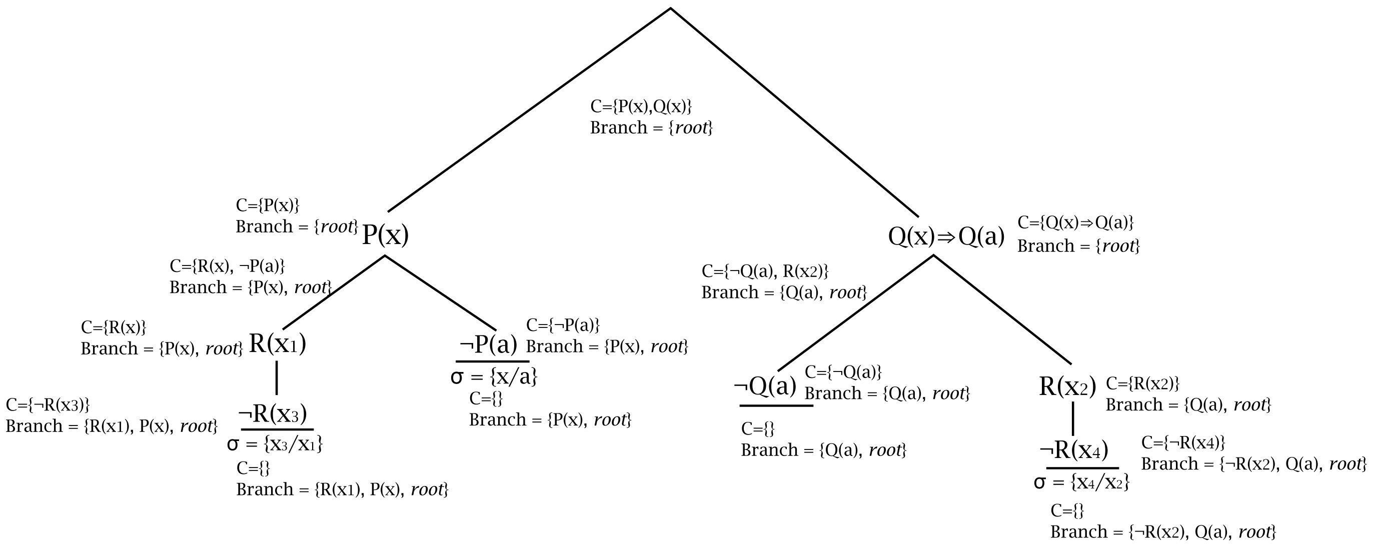


Figure 2.3: Graphical tableau representation of a derivation

Although we will formalise the set of rules for each variant or new calculus introduced as in Table 2.1.1,in the rest of this report the graphical tree representation is always used for its simplicity, clarity and compactness, without implicitly indicating any elements of the tuple $C, S, Branch$, since they can be observed directly in the representation.

2.2 The Connection Tableau Calculus

The Connection tableau calculus is a combination of the Connection Method [8] (briefly introduced in Section 2.3.4) and the tableau calculus, and a generalisation of Model Elimination [30]. Connectedness is a condition over tableau that does not allow to expand a branch using clauses that are unrelated to the literals already in the branch. Connectedness can be defined in two ways:

- *Strong connectedness*: when expanding a branch at a node, use an input clause only if it contains a literal that can be unified with the negation of the literal at the node.
- *Weak connectedness*: when expanding a branch at a node, allow the use of clauses that contain a literal that unifies with the negation of a literal on the branch.

In this project we refer exclusively to the strong version of connectedness.

Through this section we define some basic concepts and the calculus itself in Section 2.2.1, and some simple shortcuts to improve efficiency in Section 2.2.2.

2.2.1 Description of the framework

Definition 12. (Connection Tableau). A *connection tableau* or *model elimination tableau* is a clausal tableau such that each inner node L (except for the root) has an immediate successor labeled with $\overline{L}$. Additionally, a tableau is called *regular* if no two nodes on any branch are labeled with the same literal.

As in general clausal tableau calculus, there are two inference rules for connection tableau calculus, *extension* and *reduction* (corresponding to *clause extension* and *closure*, respectively) plus the single construction step of starting the tableau. Table 2.2.1 contains the definition of those rules. We can observe that only the *extension rule* essentially differ from the rules presented for clausal tableaux, reflecting the restrictions imposed by the connection method.

$Start\ (St)\ \dfrac{\{root\}, S, \{\}}{C, S, \{root\}}$ where $C \in S$
$Extension\ (Ext)\ \dfrac{C \cup \{K\}, S, Branch}{(C_2 \backslash \{L\})\sigma, S, (Branch \cup \{K\})\sigma \ \ \|\| \ \ C\sigma, S, (Branch)\sigma}$ and C_2 is a renamed instance of $C_1 \in S, L \in C_2, \sigma(K) = \sigma(\overline{L})$
$Reduction\ (Red)\ \dfrac{C \cup \{K\}, S, Branch \cup \{L\}}{C\sigma, S, (Branch)\sigma \cup \{L\sigma\}}$ with $\sigma(K) = \sigma(\overline{L})$

Table 2.2: Rules of the Connection Tableau Calculus

As we did for clausal tableau calculus, we can define the previous rules in term of a set of clauses S and a marked tableau T for S, to facilitate the use of the graphical representation.

- *Start*: T is a one-node tree with a root only. Choose a clause $L_1 \vee \ldots \vee L_n \in S$ and attach the nodes $L_1, \ldots, L_n$ to the root.
- *Extension rule*: select a subgoal K and a clause $C = L_1 \vee \ldots \vee L_n \in S$ and rename the variables to make them disjoint from the variables occurring in T, and such that L_i and $\overline{K}$ are unifiable with mgu σ for some $1 \leq i \leq n$. Attach new successor nodes to K, mark L_i with *closed* and apply σ to all the literals in T.
- *Reduction rule*: if a subgoal K has an ancestor node L unifiable with $\overline{K}$ with mgu σ, then mark K with *closed* and apply σ to all the literals in T.

It is obvious that only marked and connected tableau can be generated by application of the above rules.

The indeterminism introduced by the selection of the next subgoal, as we have defined the inference rules, can be solved by fixing a *subgoal selection function*, that is, a mapping ϕ that assigns a subgoal to every marked tableau not closed. In practise, we can work with any selection function without affecting the deductive power of the calculus (for a complete proof, see [27]). It is interesting to note that this is a *strong* independence of the selection function that does not hold for other calculi like resolution. For example, linear resolution is only *weakly* independent of the selection function, which means that completeness is preserved if we change the selection function but it is not guaranteed to obtain the same proof objects. If we only want to show that a set is inconsistent, then this weak version of independence would be sufficient, as the empty clause is always produced by the proof method, whatever the selection function is. However, there are many cases in which we are also interested in the *substitutions* that have been employed to find this empty clause, for example, in Logic Programming. In that case the strong version of independence is desirable.

From now on, we will be using the standard *depth-first left-most* selection function, that select the left-most subgoal. In addition, when applying an extension rule, we will reorder the literals in the new clause C to close K with the left-most literal of C.

We will see now an example of everything that have been introduced so far.

Example 4. Given the following set of clauses (axioms) S and clause C, we show that C follows from S.

$$S = \{\neg Q(x) \vee \neg R(b), \neg R(x) \vee \neg P(b), Q(x) \vee \neg R(x), R(x) \vee P(x)\}, \;\; C = P(a)$$

To find a proof, we construct a closed tableau for $S \cup \{\neg P(a)\}$. We choose $R(x) \vee P(x)$ as start clause. To mark a node as *closed* we underline it, writing below the mgu σ found and we indicate with $\Leftarrow$ that a mgu σ has been applied to a node in the tableau. For the sake of simplicity, we indicate this only where is relevant, that is, on the nodes in the open part in the tableau after application of extension or reduction. In Figure 2.4 we can see the resulting closed tableau. The application of the rules extension and reduction is indicated in the Figure by using the letters E and R, respectively.

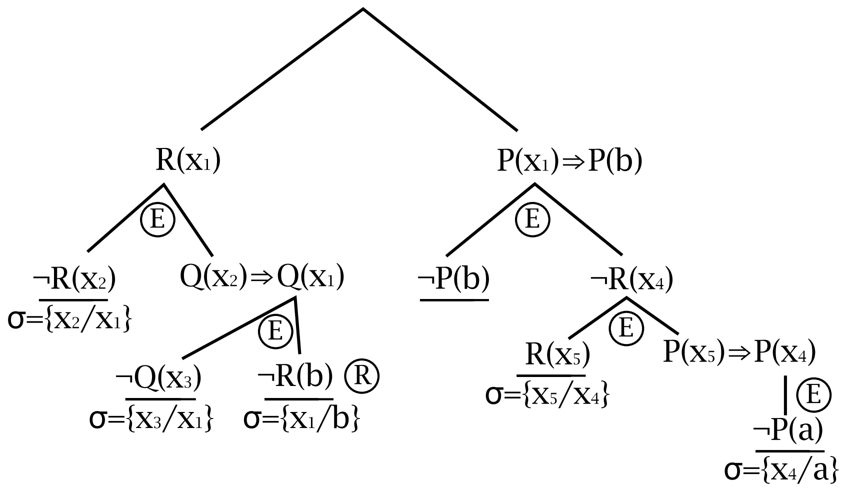


Figure 2.4: Example of refutation finding using connection tableau calculus

In the rest of this report, we will sometimes omit the renaming of clauses and substitutions applied when those involve free variables for simplicity and when no confusion can arise.

Proposition 2. *(Soundness and completeness for the Connection Tableau Calculus).*

The connection tableau calculus is sound and complete:

- *If a closed connection tableau is constructed from a set of clauses S by application of the rules of the calculus, then S is unsatisfiable.*
- *For each unsatisfiable set of clauses S there exists a closed connection tableau that can be constructed using the rules of the calculus.*

Proof. (Soundness and completeness for the Connection Tableau Calculus).

- *Soundness.* We show the contrapositive: if a set of clauses S is satisfiable, then $\forall n \geq 0$, if a connection tableau of depth n is developed from S then it has at least one open branch. Suppose that S is satisfiable. Then there is a Herbrand model for S, which means that any set of ground instances of the clauses in S is satisfiable. We proceed by induction on n.

  $n = 0$. There cannot be any closed nodes, since neither extension nor reduction have been applied. Therefore, the tableau has at least one open branch.

  $n \to n + 1$. Let T be a tableau of depth n developed for S, such that it has at least one open branch. Then we cannot applied reduction and the only possibility to increase the depth of T to $n + 1$ is to perform an extension step. If we could close all the nodes and we bound any variable in the tableau to any ground term, the set of tableau clauses of T is an unsatisfiable set of ground clauses. All tableau clauses are clauses of S, so S would contain a set of unsatisfiable ground clauses, contradicting the assumption that S is satisfiable.

- *Completeness.* Suppose S is unsatisfiable. Then there is at least a set of ground instances S_g of the clauses in S that is unsatisfiable. Then we can construct a closed connection tableau from the clauses in S_g as follows. Start with any top clause C from S_g and iterate the following procedure, while the intermediate tableau are not closed: choose an open node L in the current tableau. If a reduction step can be applied, mark L as closed. Else, select a clause D from S_g containing $\overline{L}$ and perform an extension step with D at node L. This procedure generates only connection tableau and it must terminate, either because all nodes are closed or because no clause D can be found to perform an extension step. In that case, we would have an open branch in the tableau T containing literals $L_1, \ldots, L_n$ contradicting the assumption that S_g is unsatisfiable.

For a more detailed proof, see [11].

□

The task of finding a proof (in this case a refutation proof) for S involves exploring the (in general infinite) space of all possible tableau. A complete procedure has to be utilised to ensure that a closed tableau is obtained. Some search strategies that preserve completeness of the calculus are for example depth-first-search with iterative deepening (DFID) based on backtracking, breadth-first-search (BFS) or DFID with resource distribution (DFIDR see [17] for more details). We return to this topic later in this chapter, in Section 2.3.3.

For the performance of a theorem prover based on tableau it is crucial to employ pruning methods or *shortcuts*, avoiding redundant branches of the search space. It would be interesting also to incorporate some techniques that reduce the length of the proof. In the next sections we will investigate some existing shortcuts.

Notice that in the framework of connection tableau, the search space has already been pruned by introducing restrictions in the tableau that can be produced, since they have to be regular and connected.

2.2.2 Some shortcuts

We will discuss three techniques in this section: *factoring* (a generalisation of another shortcut, *merging*), *re-use* (also known as *folding-up*) and a version of it, *folding-down.* Factoring was introduced for model elimination ([25]) and also used in connection calculus ([8]). Folding-up and folding-down were introduced and proved to preserve completeness of the method in [37], as a way of integrating the cut rule into connection tableau calculi to overcome the weakness of this method because of this *cut-freeness.* There is another tableau calculus, the *KE-tableau* calculus, which makes explicit use of the cut as a rule of the calculus, but it does not have the requirement of connectedness that prevents from the use of the cut rule.

The (atomic) cut rule in clausal tableau is applied as follows: given a marked tableau T select a subgoal K and an arbitrary literal L, attach two successor nodes L and $\neg L$ to K. L is called the *cut formula* of the cut step. Clearly, this cut rule cannot be applied to a connection tableau because the connectedness condition has to be maintained. It enforces that any application of cut at K would have to add the literals K and $\overline{K}$, making no advance at all. A connection version of the cut rule could be defined, by attaching three nodes to K, that is, $\overline{K}$ and two literals L and $\neg L$ (a tautology). However, this does not help to increase the computational power of connection tableau ([37]). The three shortcuts described next do increase this power.

- *Factoring*: consider a closed tableau with two nodes N_1 and N_2 labeled with the same literals and suppose that all ancestor nodes of N_2 are also ancestors of N_1. Then we could have used the closed sub-tableau below N_2 to close N_1 because all the inference steps would have been possible due to the common ancestors. This leads to the introduction of the factoring rule. To avoid a cyclic application of this rule, a dependency relation between nodes must be introduced: for two nodes N_1 and N_2, $N_1 \prec N_2$ means that the solution of N_2 depends on the solution of N_1.

  We can describe the rule as follows: given a marked tableau T, select a subgoal L and another node K unifiable with L with mgu σ ($L\sigma = K\sigma$), such that L is dominated by a node that has K among its immediate successors and if N is the sibling node of K in the branch from the root to L, then $N \not\prec K$. Then we can *factorize* L with K: mark L with *closed*, modify the dependency relation adding $K \prec N$ and computing its transitive closure and finally apply the substitution σ to the whole tableau.

  Example 5. The rule is illustrated in Figure 2.5, where the subgoal L is factorized with the node K, then $K \prec N_1$ is added to the previously empty dependency relation. This denotes that the solution of node N_1 depends on the solution of node K. After that, factoring of the subgoal N_2 with the node N_1 is not possible anymore (as N_1 has a sibling, K, in the branch from the root to N_2, such that $K \prec N_1$). This makes intuitively sense, as we would have a cyclic dependency between solutions in the tableau.

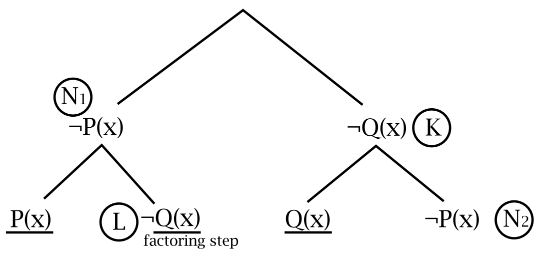


Figure 2.5: Factoring (merging) step in a connection tableau

Notice that we can distinguish two types of application of factoring at a subgoal N_1 with a node N_2: either N_2 has already been closed, or the node N_2 or some of its successors are not yet marked. The first case corresponds to an *optimistic* application of the rule and to the shortcut *merging* mentioned before, whereas the second case would be a *pessimistic* application, equivalent to *re-use*. The latter is more commonly implemented than the former and as it is what we implement in our prover (see Section 5.4), we do not formalise other shortcuts presented in this section.

- *Re-use* or *folding-up*: the theoretical basis for this rule is the extraction of *bottom-up lemmas* from open parts of a tableau (not to be confused with intermediate lemmas discussed in Section 3.1) presented in [36]. The idea behind this rule can be more easily explained with an example.

  Example 6. Consider the tableau in Figure 2.6 (the underlined nodes are those marked with `closed`). The arrow points at the last inference step (reduction), that have closed the dominating nodes $Q(x)$ (2) and $R(x)$ (1). Their predecessor $P(x)$ (3) has been used for the last reduction step. This allows to derivate the lemmas $\neg Q(x)$ and $\neg R(x)$, valid in the context $P(x)$. We memorise them by *folding them up* to the edge that dominates their context (E). They are represented in a box in the tableau on the right. Now, the lemmas in the edges can be used for reduction steps below them, which means that we can close $R(x)$ at (4), solving $P(x)$ at (3) and folding $\neg P(x)$ up also (in this case above the root). Finally, this new lemma allows us to close $P(x)$ at (5).

  Essentially we are *re-using* the proofs below $R(x)$ and $P(x)$ to close subsequent occurrences of them to the right in the tableau.

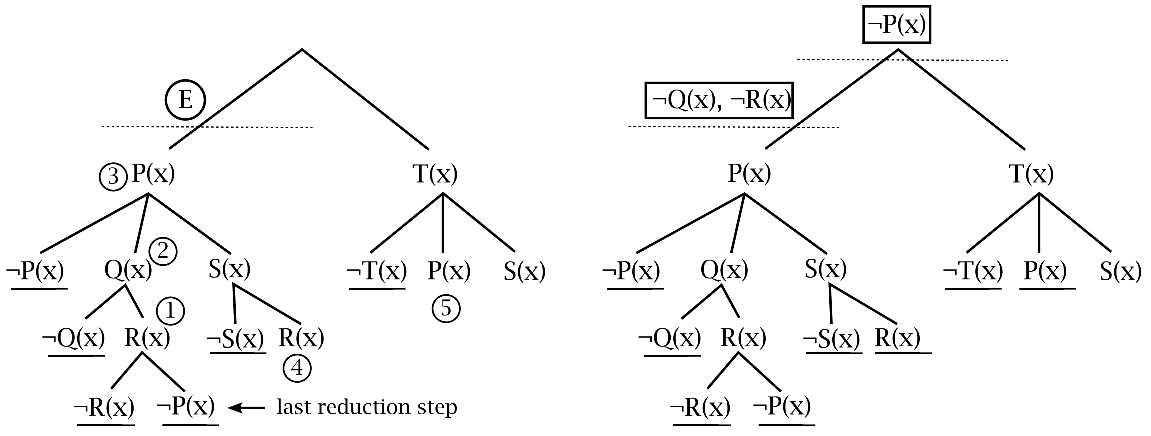


Figure 2.6: Application of the re-use rule twice in a connection tableau

  The folding-up rule is sound and computationally stronger that factoring in connection tableau ([37]). In what follows, we refer to this rule exclusively as re-use, restricting the use of the term ‘lemma’ for the intermediate lemma extension to avoid confusion.

  In order to formalise this shortcut, we need to extend the tuple $C, S, Branch$ to $C, S, Branch, R$, where R keeps track of the literals that can be re-used. R is a set of literals representing the current *lemmas* or literals that we can *re-use*.

  The modified rules of Table 2.2.1 to consider the new argument are shown in Table 2.2.2 and a new closure rule called *Reuse* is added to the set.

- *Folding-down*: this is a restriction of the folding-up rule to allow an *optimistic* labelling of edges (it would be also version of the optimistic application of factoring or merging). If a linear ordering is defined on the successor nodes $L_1, \ldots, L_n$ of any node in a connection tableau, from left to right, we could label the edge leading to any node L_i with the negations of the nodes smaller

$$Start\ (St)\ \frac{\{root\}, S, \{\}, \{\}}{C, S, \{root\}, \{\}} \text{ where } C \in S$$
$$Extension\ (Ext)\ \frac{C \cup \{K\}, S, Branch, R}{(C_2 \backslash \{L\})\sigma, S, (Branch)\sigma \cup \{K\sigma\}\ \ \|\ \ C\sigma, S, (Branch)\sigma, R\sigma \cup \{K\sigma\}}$$ and C_2 is a renamed instance of $C_1 \in S, L \in C_2, \sigma(K) = \sigma(\overline{L})$
$$Reduction\ (Red)\ \frac{C \cup \{K\}, S, Branch \cup \{L\}, R}{C\sigma, S, (Branch)\sigma \cup \{L\sigma\}, R\sigma \cup \{L\sigma\}}$$ with $\sigma(K) = \sigma(\overline{L})$
$$Re-use\ (Reu)\ \frac{C \cup \{K\}, S, Branch, R \cup \{L\}}{C\sigma, S, (Branch)\sigma, R\sigma \cup \{L\sigma\}}$$ with $\sigma(K) = \sigma(\overline{L})$

Table 2.3: Rules of the Connection Tableau Calculus with Re-use

than L_i in the ordering. See Figure 2.7 for an example. The tableau need to be developed from left to right, for this rule to be valid.

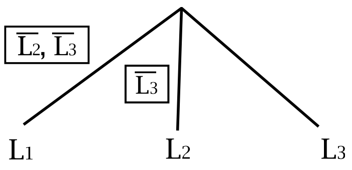


Figure 2.7: Application of the folding-down rule

One real example of theorem prover implemented within this framework, that also incorporate all the previous shortcuts among others is SETHEO [36]. SETHEO is a sound and complete theorem prover for first-order logic whose main proof unit is based on the connection tableau calculus. New improved provers based on it have appeared, like p-SETHEO [56], SPTHEO [53] or E-SETHEO [44], to cite some of them.

2.3 Prolog Technology Theorem Provers

Informally, Logic programming is the use of mathematical logic for computer programming. Logic is used as a purely declarative language and the computation is made through a theorem prover or a model generator. The task of the programmer is to ensure the truth of the programs expressed in logical form. Prolog is one of the best known languages of this declarative programming paradigm. Prolog programs are simply sets of Horn clauses and the inference system is based on SLD-resolution. SLD-resolution is a refinement of resolution, both sound and complete for Horn clauses, and will be discussed in more depth in Section 2.3.2.

According to the definition of M.E. Stickel in [45, 46], a Prolog technology theorem prover (PTTP) is an extension of Prolog that is complete for the full first-order predicate calculus. It relies on fast execution of small-logical steps and it will increase its performance when Prolog machine technology progresses. PTTP are equipped with sound unification, a complete inference system and a complete search strategy. Regarding the last two, PTTP are characterised by model elimination inference procedure and depth first iterative deepening search procedure. These particular inference and search methods can be implemented taking advantage of Prolog's inference engine and can be adapted to several reasoning paradigms or proof procedures. In the next sections we explain briefly what the three extensions involve in general and provide an example of a PTTP, leanCoP, for illustration. In Chapter 5 we detail how we have approach this extension to Prolog in our prover.

2.3.1 Sound unification

Most Prolog implementations omit by default *occurs-check* in unification for efficiency purposes. That is, it allows $f(x)$ and x to be unified, not checking whether a variable is bound to a term that contains the same variable, which in theorem proving yields to erroneous deductions. For example, we could deduce $y < predecessor(y)$ from $predecessor(x) < x$, unifying y with $predecessor(x)$ and x with $predecessor(predecessor(x))$ and hence creating a circular binding for x.

The solution can rely either in writing an alternative Prolog predicate `unify` to replace Prolog unification algorithm whenever necessary or simply to use the built-in predicate `unify_with_occurs_check/2` included in ISO Prolog implementations. A third option offered by some Prolog implementations is to switch to sound unification for all unifications by means of a runtime flag.

2.3.2 Complete inference system

Prolog's inference system is referred to as *SLD-resolution* (*Selective Linear Definite* clause resolution). This type of resolution imposes an *input* restriction in the inference steps, allowing to resolve derived clauses only with input clauses. Unlike SL-resolution, in which derived clauses can be resolved with input clauses or their own ancestor clauses and it is complete in general, SLD-resolution is complete only for Horn clauses.

The *Model Elimination* (ME) procedure [30] is similar to SL-resolution, it is complete for Horn and non-Horn clauses and represents a very convenient way to provide Prolog with a complete inference system, thanks to the *ME reduction inference rule.* Recall that the Connection Tableau Calculus introduced in Section 2.2 is a generalisation of the ME procedure. We will introduce this with an example.

Example 7. Consider the clauses $p \vee q, \neg p \vee q$, that is, $\neg p \rightarrow q, p \rightarrow q$ and let us show that Prolog fails to prove q from them. We can see that all the contrapositive clauses of the previous set, $\neg p \rightarrow q, \neg q \rightarrow p, p \rightarrow q, \neg q \rightarrow \neg p$, represented as a Prolog program (notice that `-p` represents $\neg p$ instead of using the negation as failure inference rule of prolog, *not p*):

```
q :- -p.
p :- -q.
```

```
q :- p.
-p :- -q.
```

are not sufficient to reduce `:- q.` to the empty list of literals. Prolog's inference system can be thought as *reducing* the initial list of literals in the query (`:-q.` in this case) to the empty list. Each step matches the leftmost literal in the list with the head of a clause, eliminates it and adds the body of the clause to the list. However, the ME procedure retain the left most literal as a *framed* literal (represented by `[p]`) and the ME reduction rule uses framed literals to eliminate complementary literals, employing reasoning by contradiction. This makes it possible to prove q from $p \vee q, \neg p \vee q$:

```
:- q.                % initial goal
:- p, [q].           % resolve with q :- p.
:- -q, [p], [q].     % resolve with p :- -q.
:- [p], [q].         % use ME reduction rule with [q] and -q.
:-.                  % delete left most framed literals.
```

The list of framed literals to the right of a literal is just the list of the goal's ancestors. We can see that this rule corresponds to the Connection Tableau Calculus reduction rule introduced in Section 2.2, in which the idea of ancestors is clearer and more intuitive.

2.3.3 Complete search strategy

In Prolog, a search for a proof is performed in an unbounded depth first search manner, in which rules are chosen in order. This is clearly incomplete, as for example we can not prove the goal `:- p.` with the rules:

```
p :- p.
p.
```

The first rule leads to an infinity branch in the search tree that Prolog keeps exploring, without ever applying the second rule.

It is then necessary to use a search strategy that allows us to find all possible proofs. Although one of the most obvious strategies would be Breadth First Search, due to its storage requirements (on each step we need to store all the nodes explored so far) most implementations of Prolog technology theorem provers use depth first search with iterative deepening. This is a complete search strategy that combines both depth first and breadth first strategies. Basically, it tries to find a proof up to a certain depth limit and if it fails, this limit is increased and the search starts again. This process is iterated until a proof is found. In each iteration the entire search space, up to a depth is explored completely. The effect is similar to breadth-first search but results from earlier depth's levels are recomputed rather than stored. These lower storage requirements compensate for the re-computation costs, because the amount of space required is proportional to the depth and the amount of time required, although exponential, is only a constant factor more expensive than breadth-first search [23].

There are several alternatives for defining the *depth* that will limit the size of the search space explored. Some examples are:

- Maximum height of the proof tree.

- Size of the proof tree, i.e., number of nodes.
- Number of inference steps performed (such as resolution steps or extension steps in the case of tableau).

In Section 5.3 our specific bound is explained in detail, together with some particularities induced by our proof procedure (Connection Tableau Calculus with the Intermediate Lemma Extension).

2.3.4 leanCoP

leanCoP is a compact automated theorem prover for classical first order logic based on Connection Calculi and implemented in Prolog, as a Prolog technology theorem prover. The most recent version is leanCoP 2.1., nevertheless, description and performance results are available only for the older versions leanCoP 2.0. [31, 32] and leanCoP 1.0 [33]. leanCoP 2.0. adds regularity, the re-use shortcut (denoted by the authors as *lemmata*) and a technique for restricting backtracking, exhaustively described in Section 3.3, to leanCoP 1.0. Furthermore, it provides a procedure to translate first order formulas to clausal form (see remarks below about their clausal form representation) based on a definitional transformation. Contrary to its predecessor, leanCoP 2.0. stores the input clauses as part of the Prolog's database in order to use Prolog's built-in indexing mechanism. We have studied leanCoP 2.0. in depth, due to the similarities shared with the prover goal of this project and to some extent, the simpler version of our prover described in Chapter 5, without the built-in equality described in Section 4.3.2 or the rules and pruning methods of the SOL tableau calculus (see Section 3.2), resembles the structure and architecture of leanCoP.

As mentioned, the proof procedure chosen by leanCoP's authors is based on the Connection Calculi. More specifically, they employ Bibel's connection method [8, 7], which is essentially the same as the Connection Tableau Calculus introduced in Section 2.2, but using a *matrix characterisation* rather than a tableau representation. A *matrix* is a set of clauses, $\{C_1, \ldots, C_n\}$, and in its graphical representation, clauses are arranged horizontally and the literals of each clause are arranged vertically: $[C_1, \ldots, C_n]$.

The rules of the connection method have their basis on the concepts of *connection*, *path* and *term substitution*. A *connection* is a set that contains two literals of the form $\{P(s_1, \ldots, s_n), \neg P(t_1, \ldots, t_n)\}$. A *path* through a matrix $M = \{C_1, \ldots, C_n\}$ is a set of literals that contains one literal from each clause $C_i \in M$. A *term substitution* σ is a mapping from the set of variables to the set of terms, and $\sigma(L)$ represents the literal L with all its variables substituted according to their mapping in σ. The rules of this calculus can be found in Table 2.3.4.

Regarding their clausal form representation, a difference between leanCoP and our approach is that they define a clause C as a *conjunction of literals* instead of a *disjunction of literals* (Section 1.3), $L_1 \wedge \ldots \wedge L_n$. Then a formula is said to be in clausal form if it is in *disjunctive normal form*, that is, $C_1 \vee \ldots \vee C_n$ where each C_i is a clause. Because of this representation, they employ a *positive representation*, which means that their calculus is used to characterise *validity* rather than *unsatisfiability*. For First Order Logic (and classical logics in general), the implications of this difference are insignificant.

As stated, the calculus is correct and complete: a first order formula M in clausal form (disjunctive normal form) is valid iff there is a connection proof for ϵ, M, ϵ, i.e., a derivation in for ϵ, M, ϵ in the connection calculus, so that all leaves are axioms. Additionally and without loss of completeness or correctness, they consider only all-positive clauses (clauses with only positive literals) as possible start clauses.

Example 8 illustrates the differences between the connection calculus with the matrix representation

Axiom (Ax) $\dfrac{\{\}, M, Path}{}$	*Start* (St) $\dfrac{\epsilon, M, \epsilon}{C_2, M, \{\}}$ and C_2 is a copy of $C_1 \in M$
Extension (Ext) $\dfrac{C \cup \{L_1\}, M, Path}{C_2 \backslash \{L_2\}, M, Path \cup \{L_1\} \;\; \vert\vert \;\; C, M, Path}$ and C_2 is a copy of $C_1 \in M$, $L_2 \in C_2, \sigma(L_1) = \sigma(\overline{L_2})$	
Reduction (Red) $\dfrac{C \cup \{L_1\}, M, Path \cup \{L_2\}}{C, M, Path \cup \{L_2\}}$ with $\sigma(L_1) = \sigma(\overline{L_2})$	

Table 2.4: Rules of the connection method

and the connection tableau calculus.

Example 8. Let $S = \{\{P(a), R(b)\}, \{\neg P(a), Q(x)\}, \{\neg Q(c), \neg P(b)\}, \{\neg R(y), P(b)\}\}$ be a set of clauses and M be its corresponding matrix. We show a connection proof of its validity (for disjunctive normal form) or unsatisfiability (for conjunctive normal form) in matrix and tableau graphical representations.

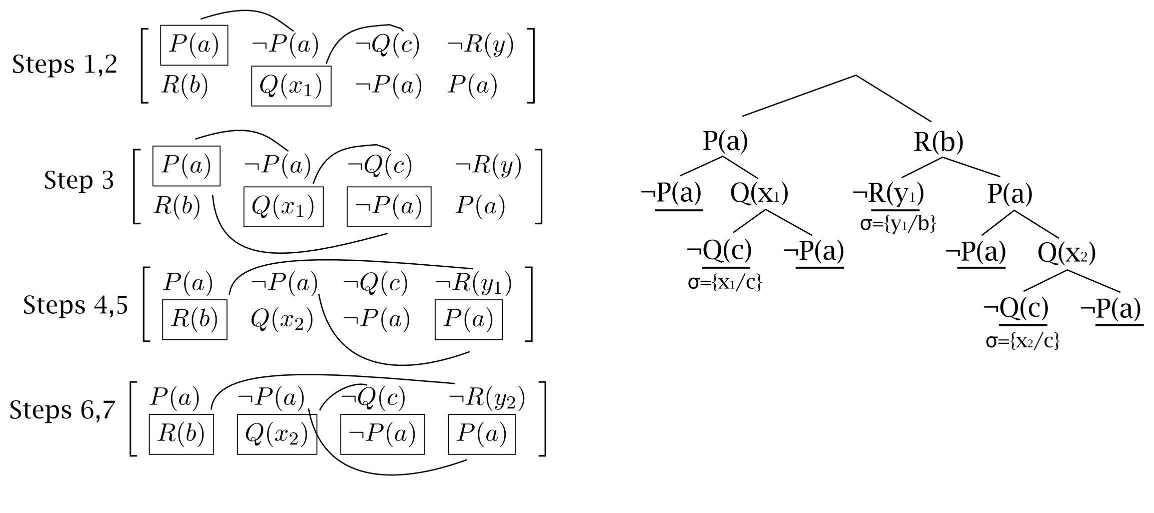


Figure 2.8: Connection proof using the graphical matrix and tableau representations

Arcs in the matrix representations correspond to connections and literals of the active path are boxed.

The search strategy employed by leanCoP is based on iterative deepening depth first search, bounded by the number of extension steps involving not ground clauses. There is not any form of clause subsumption implemented in leanCoP, neither any built-in equality reasoning mechanism. For equality problems, all equality axioms are added to the input set.

Chapter 3

Extensions of Connection Tableau Calculus

In the previous chapter, in Section 2.2, we have established the basis of the framework in which our theorem prover is located. Besides, we presented some basic shortcuts that can help to reduce the search space. In this chapter we aim to go further along this matter, describing and analysing more sophisticated extensions to the previously described framework. In particular, we detail the Intermediate Lemma Extension, which constitutes the main purpose of this project, the SOL Tableau Calculus, a variant of the connection calculus for consequence finding, and a restriction for backtracking in connection calculi.

3.1 Intermediate lemma extension

In Chapters 1 and 2 we have seen that Prolog programs are basically sets of Horn clauses, that is, clauses with at most one positive literal. This limitation makes it possible to use a more efficient and restricted version of resolution, SLD-resolution, as complete inference system. We discussed in Section 2.3.2 that SLD-resolution is not complete for arbitrary clauses, so despite being very efficient, it is not applicable for general theorem proving purposes.

The *intermediate lemma extension* is a generalisation of Prolog for tableau to arbitrary clauses, i.e., clauses that can have any number of positive literals. It aims to take advantage of the efficiency of linear resolution as inference system without loosing the completeness required for a theorem prover.

Prior to define new rules to present the extension as a new calculus (variant of connection tableau calculus) it is simpler and more intuitive to introduce it informally, describing the whole procedure as a whole within the framework of connection tableau. In Section 5.2, we will further refine our description of the procedure when we explain how we have implemented it in the prover and combined with shortcuts such as regularity constraints or re-use. In Chapter 7 we study in depth some other aspects of intermediate lemma generation like additional restrictions or necessary conditions for completeness, among other features developed as a case study.

In this method, to show that a given set of clauses S is unsatisfiable, a positive literal is selected from each non-Horn clause in S as the *conclusion literal* and the remaining positive literals are the *lemma literals*. Now, to construct a closed connection tableau T for S, we proceed as in Prolog, starting from a clause with only negative literals (a *query*). Such a clause is guaranteed to exist if S is unsatisfiable. Otherwise, every clause in S would have at least one positive literal, so we could assign $\top$ to any ground instance of each positive literal that appears in S. This would be a model for S because it satisfies any clause $C \in S$ and hence S would not be unsatisfiable.

Then the extension rule is applied by using only the conclusion literals to close negative literals, ignoring other positive literals that arise (i.e., leaves with positive literals are not pursued as goals), unless a reduction step can be executed. When no further inference step can be performed and no closed tableau has been found, if all the positive leaf literals in T that have not been closed are ignored, the tableau can be considered as solved. We label them as *skipped* and derivate a clause made of all skipped literals from T as an *intermediate lemma*, *skip(T)*. Then, we add $skip(T)$ to S and start the process again, removing clauses that are subsumed by other clauses in $S \cup skip(T)$. We finish when we find a closed tableau T with $skip(T) = \emptyset$. Instead of constructing a single closed tableaux for a set S as in the general connection tableau calculus, we construct a sequence of tableau $T_1, T_2, \ldots, T_n$, with $n \geq 1$, T_n being a closed tableau and $skip(T_1), skip(T_2), \ldots, skip(T_{n-1})$ representing the intermediate lemmas generated in the process. Not all of them necessarily appear in the final closed tableau, some of them can be subsumed by other clauses during the process or do not contribute to the final proof found.

Example 9. To illustrate the mechanism, consider the set of clauses

$$S = \{\neg R(x), \neg P(x) \vee \neg S(x), \underline{R(x)} \vee P(x) \vee \neg Q(x), \underline{S(x)}, T(x) \vee \underline{Q(x)}, \neg T(x)\}$$

where we have underlined the literals chosen as conclusion literals. For the sake of simplicity, the renaming of the clauses and substitutions are omitted. We start with clause $\neg R(x)$ and, as we can see in tableau T_1 in Figure 3.1, derive the lemma $T(x) \vee P(x)$. We add it to S and start again choosing $\neg P(x) \vee \neg S(x)$ as top clause of tableau T_2. This time we derive the lemma $T(x)$. We add it to our set of clauses, removing the previous lemma $T(x) \vee P(x)$ that is subsumed by the new one. Finally, we can close the tableau T_3 starting with $\neg T(x)$ and using the last generated lemma.

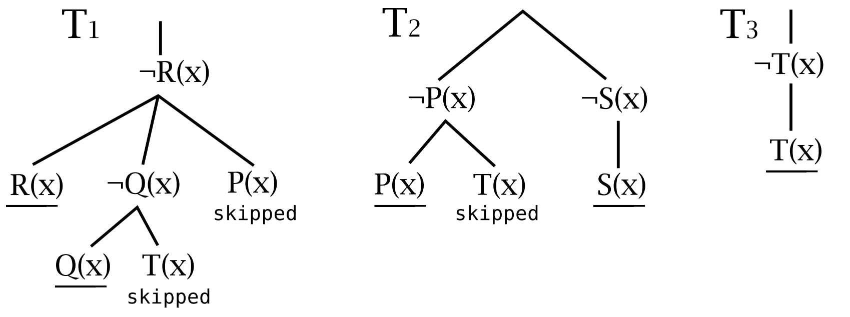


Figure 3.1: Generation and use of intermediate lemmas

Although we could piece together the tree tableau used to find a refutation for S to form a closed tableau, it would not be a connection nor regular tableau.

Proposition 3. *(Soundness and completeness for the Connection Tableau Calculus with the Intermediate Lemma Extension).*

The Connection Tableau Calculus with the Intermediate Lemma Extension is sound and complete.

- *If a sequence of connection tableau $T_1, \ldots, T_n$ with T_n closed can be constructed from S by application of the rules of the calculus, then S is unsatisfiable.*
- *If S is unsatisfiable, then there exists a sequence of connection tableau $T_1, \ldots, T_n$ constructed using the rules of the calculus and such that T_n is closed.*

Proof. (Soundness and completeness for the Connection Tableau Calculus with the Intermediate Lemma Extension).

- *Soundness.* Let S be a set of clauses for which a sequence of connection tableau $T_1, \ldots, T_n$ with T_n closed has been found. We can distinguish two cases:
  - (a) If T_n contains only instances of clauses from the original set S, then T_n is a closed connection tableau for S and S is unsatisfiable by soundness of the Connection Tableau Calculus.
  - (b) If T_n contains some instances of clauses in $\{skip(T_1), \ldots, skip(T_{n-1})\}$ then we can construct a complete and closed tableau from T_n that uses only instances of clauses in S as follows. We start with the last instance of a lemma $skip(T_i)$ used and replace it with the corresponding instance of T_i. All the leaves of this tableau will close in the same way as the lemma did in T_n. We repeat this substitution of $skip(T_i)$ with T_i until all lemmas used in T_n have been replaced. Then, although the closed tableau found is not a connection tableau, by soundness of clausal tableau calculus, we can conclude that S is unsatisfiable.

- *Completeness.* Suppose that S is a minimally unsatisfiable set of clauses, i.e., each $S' \subset S$ is satisfiable, or, in other words, removing any clause from S would make it satisfiable. Let k be the number of lemma literals in S. We proceed by induction in k.
  - $k = 0$. No clause in S has a lemma literal, which means that all clauses are actually Horn clauses. As we have seen before, there is at least one clause with only negative literals if S is unsatisfiable. Then there is a standard closed connection tableau starting from this all-negative clause by completeness of connection tableau calculus.
  - $k \to k+1$. As induction hypothesis IH suppose that for any unsatisfiable set with k lemma literals there is a sequence of tableau constructed using the method and that can be pieced together to form a closed complete tableau for such set.

    Let $C \in S$ be a clause with a lemma literal L and let $D = C \backslash \{L\}$. Now, consider the sets $S_1 = S\backslash\{C\} \cup \{D\}$ and $S_2 = S\backslash\{C\} \cup \{L\}$. We show that S_1 and S_2 are unsatisfiable. Suppose not. If S_1 is satisfiable, then there exists a model M that makes all clauses in $S\backslash\{C\} \cup \{D\}$ true. Then M is also a model for S because $M \models S\backslash\{C\}$ and $M \models D \subset C$, which contradicts that S is unsatisfiable. Analogously, if there exists a model for S_2, $M \models S\backslash\{C\}$ and $M \models L \in C$ implies that S is satisfiable. Therefore, S_1 and S_2 are unsatisfiable.

    We observe that if S has $k + 1$ lemma literals, then S_1 and S_2 have k lemma literals and hence, by IH, there exists two sequences of connection tableau $T_1^1, \ldots, T_n^1$, $T_1^2, \ldots, T_n^2$ constructed using the intermediate lemma method with T_n^1 and T_n^2 closed for S_1 and S_2 respectively. Now take the sequence of tableau for S_1 and add the corresponding instance of the literal L to each instance/renamed instance of the clause C' used, forming instances of C. Now the sequence obtained $T_1'^1, \ldots, T_n'^1$ after those replacements is not finished with a closed tableau $T_n'^1$ but it gives rise to lemma L. We can concatenate now the two sequences together to form the sequence

    $$T_1'^1, \ldots, T_n'^1, T_1^2, \ldots, T_n^2$$

    with T_n^2 being a closed connection tableau. Only clauses from S, lemmas generated within the sequence and well-formed connection tableau are used on it.

□

We formalise the rules for this calculus in Table 3.1. For each clause $C \in S$ such that C has at least one positive literal, a positive literal is chosen non-deterministically and underlined as conclusion literal.

$\mathit{Start}\ (\mathit{St})\ \dfrac{\{\mathit{root}\}, S, \{\}, \{\}}{C, S, \{\mathit{root}\}, \{\}}$ where $C \in S$ is an all-negative clause, $C = \{\neg L_1, \ldots, \neg L_n\}$
$\mathit{Extension}\ (\mathit{Ext})\ \dfrac{C \cup \{\neg K\}, S, \mathit{Branch}, \mathit{Skip}}{(C_2 \backslash \{L_1\})\sigma, S, (\mathit{Branch})\sigma \cup \{\neg K\sigma\}, (\mathit{Skip})\sigma \ \ \vert\vert\ \ C\sigma, S, (\mathit{Branch})\sigma, (\mathit{Skip})\sigma}$ and C_2 is a renamed instance of $C_1 = \{\underline{L_1}, \ldots, L_n\} \in S, \sigma(K) = \sigma(L_1)$
$\mathit{Reduction}\ (\mathit{Red})\ \dfrac{C \cup \{K\}, S, \mathit{Branch} \cup \{L\}, \mathit{Skip}}{C\sigma, S, (\mathit{Branch})\sigma \cup \{L\sigma\}, (\mathit{Skip})\sigma}$ with $\sigma(K) = \sigma(\overline{L})$
$\mathit{Skip}\ (\mathit{Sk})\ \dfrac{C \cup \{K\}, S, \mathit{Branch}, \mathit{Skip}}{C, S, \mathit{Branch}, \mathit{Skip} \cup \{K\}}$ with K a positive literal
$\mathit{Lemma\ \ derivation}\ (\mathit{Lem})\ \dfrac{\{\}, S, \mathit{Branch}, \mathit{Skip}}{\{\mathit{root}\}, S \cup \mathit{Skip}, \{\}, \{\}}$ with $\mathit{Skip} \neq \emptyset$

Table 3.1: Rules of the Connection Tableau Calculus with the Intermediate Lemma Extension

The rules are defined based on the tuple $C, S, \mathit{Branch}, \mathit{Skip}$, where C, S, Branch is the same tuple used in Connection Tableau Calculus and Skip is a set of literals representing all skipped leaves.

We provide also a explanation for them in the context of the graphical tree representation, as we have done for previous variants of Clausal Tableau Calculus in Chapter 2. In order to define the rules for a given a set of clauses S and a marked tableau T for S, we need to refine the definition of marked tableau to allow for the new label *skipped* and introduce the notion of *solved tableau*, to differentiate a tableau in which all leaves are marked and a lemma is generated from them, from a tableau in which all branches are closed.

Definition 13. (Marked, solved and closed tableau). A *marked tableau* is a clausal tableau in which some of the leaf nodes are marked with *closed* or *skipped.* The unmarked leaves are called *subgoals.* A tableau is *solved* if all the leaves are marked. A tableau is *closed* if all its leaves are marked with *closed.*

- *Start*: T is a one-node tree with a root only. Choose an all-negative clause $\neg L_1 \vee \ldots \vee \neg L_n \in S$ and attach the nodes $\neg L_1, \ldots, \neg L_n$ to the root.
- *Lemma derivation rule*: T is a marked solved tableau. Add $\mathit{skip}(T)$ to S, $S = S \cup \mathit{skip}(T)$ and start a new one-node tree with a root only.

- *Extension rule*: select a negative subgoal $\neg K$ and a clause $C = \underline{L_1} \vee \ldots \vee L_n \in S$ with conclusion literal L_1 such that L_1 and K are unifiable with mgu σ, and rename the variables to make them disjoint from the variables occurring in T. Attach the new successor nodes $\neg L_1, \ldots, \neg L_n$ to $\neg K$, mark L_1 with *closed* and apply σ to all the literals in T.
- *Reduction rule*: if a (positive or negative) subgoal K has an ancestor node L unifiable with $\overline{K}$ with mgu σ, then mark K with *closed* and apply σ to all the literals in T.
- *Skip rule*: Select a positive subgoal K and mark it with *skipped.*

The process of forming lemmas and refutations could be controlled in some way, whether fixing the depth to which a tableau has to be developed to derive lemmas, or introducing restrictions in the length, structure or syntax of lemmas that can be produced. Regarding the handling of equations, one possibility would be to restrict intermediate lemmas only for equality literals. This would be indeed a restriction in the syntax of lemmas. We will develop further this topic in Chapter 7.

3.2 SOL Tableau Calculus

In this section we introduce a tableau calculus based on SOL-resolution for *consequence finding*, rather than *refutation* or *proof finding*, which is the goal of a theorem prover. The reason why SOL Tableau Calculus can be of interest from the purposes of this project is that the generation of intermediate lemmas can be seen as a special case of consequence finding. Some very interesting pruning methods have been introduced and proved to be complete (for consequence finding, which is stronger than for refutation finding, as we will see) for the connection tableau version of SOL-calculus ([21]) and they have been implemented quite successfully in a deduction system for consequence finding, SOLAR [17]. We aim to adapt some of them to the framework of connection tableau and intermediate lemma generation.

We will explain briefly the task of consequence finding before introducing the general characteristics of SOL calculus within connection tableau in Section 3.2.1 and we will present some pruning methods that have been successfully implemented in SOLAR in Section 3.2.2. In Section 6.2 it is carefully explicated how we have adapted some of the pruning techniques to the settings of our prover and how we have implemented them.

The task of consequence finding seeks to generate the theorems entailed by a set of given axioms. Many practical reasoning tasks are indeed particular cases of consequence finding, for example, theorem proving is equivalent to show that the empty clause is a consequence of a set of axioms plus the negation of the theorem to be proved. Another example is *abductive reasoning* ([19, 20]), that given a theory B and a set of observations E, aims to find a hypothesis H such that $B \wedge H \models B$. This happens to be equivalent to find $\overline{H}$ such that $B \wedge \overline{E} \models \overline{H}$, which is again a special case of consequence finding.

Many problems can be solved then in the framework of consequence finding, but most of them require that the generated consequences or theorems satisfy certain properties. In other words, not all consequences entailed by the set of axioms are of interest. For example, in the task of theorem proving, the only consequence of interest is the empty clause and in many applications of abductive reasoning, possible hypothesis are restricted to contain only ground literals whose predicates are defined as *abducibles*. Furthermore, in general it is not desirable to produce consequences that are subsumed by other, to avoid producing redundant clauses. For all those reasons, it is necessary to define some language that all clauses in the generated consequences must satisfy. We will call them *characteristic clauses*. This language will be different from one application to another, but K. Iwanuma, K. Inoue and K. Satoh provide in [21] a general schema (*production fields*, explained below) for defining such restrictions over an underlying language.

3.2.1 The SOL Tableau Calculus

We need to introduce some preliminary definitions. Only in this section, and following the author's nomenclature, clauses will be considered *multisets* of literals rather than sets. The definition of subsumption below is then slightly different from the one given in Section 1.3.

Definition 14. (Subsumption for clauses as multisets). We say that a clause C *subsumes* another clause D if there is a substitution σ such that $C\sigma \subseteq D$ *and* C *has no more literals than* D. The classical definition of subsumption only requires the first condition.

To see the difference between the two definitions, let us consider the clauses $C = \{P(x,y), P(y,z), Q(x,z)\}$ and $D = \{P(u,u), Q(u,u)\}$. We have that C subsumes D in the classical sense, for example, if $\sigma = \{x/u, y/u, z/u\}$ we have $C = \{P(u,u), P(u,u), Q(u,u)\} = \{P(u,u), Q(u,u)\} \subseteq D$. However, this is not true with the *multiset* version of the definition because C has more literals than D.

As with the classical definition, we say that C *properly subsumes* D if C subsumes D but D does not subsumes C. For a set of clauses S, μS denotes the set of clauses not properly subsumed by any clause in S.

Definition 15. (Production field). Let $\mathcal{L}$ be the set of all literals of the first-order logic language ($\mathcal{L}^+$ and $\mathcal{L}^-$ denote the sets of all positive and negative literals, respectively). A *production field* $\mathcal{P}$ is a pair $\langle L, Cond\rangle$, where $L \subseteq \mathcal{L}$ and $Cond$ is a certain condition to be satisfied. If $Cond$ is not specified, then $\mathcal{P}$ is denoted just as $\langle L\rangle$. A clause C belongs to $\mathcal{P}$ if every literal in C belongs to $inst(L)$ (the set of instances of L) and $Cond$ is satisfied by C.

Definition 16. (Stable production field). If for any clauses C, D such that C subsumes D, $D \in \mathcal{P}$ only if $C \in \mathcal{P}$, the production field $\mathcal{P}$ is said to be *stable*. For a set of clauses S, $Th_{\mathcal{P}}(S)$ is the set of theorems entailed by the clauses in S that also belong to $\mathcal{P}$.

If a set of literals L contains a non-maximally general literal, then the production field $\langle L\rangle$ is in general not stable (for this general definition of stability, it may be stable when restricting the definition of stability to clauses within a concrete set). A literal is *maximally general* if its arguments are all distinct variables. For example, $P(x,y)$ is maximally general but $Q(a, f(b))$ is not. Production fields that impose a length condition are not stable with the classical definition of subsumption but this is fixed with the definition of subsumption for clauses as multisets.

Stability of production fields is important in practise because it influences the effectivity of some pruning methods (see [18] for more details). Since unstable production fields appear in some applications of consequence finding, Ray and Inoue proposed a transformation in [38] to make an unstable production field into a stable one. For us, a concept analogous to stability of production fields becomes critical in order to guarantee completeness for one of the extensions introduced in Chapter 6. We will return to this subject then, but for the purpose of this section, we consider all production fields to be stable.

Example 10. The following are some examples of production fields:

- $\mathcal{P}_1 = \{R(x)\}$. The clause $R(a) \vee R(f(x))$ belongs to $\mathcal{P}_1$ but the clause $R(a) \vee P(x,b)$ does not.
- $\mathcal{P}_2 = \{\mathcal{L}^-, \text{length is less than } 3\}$. The clauses consisting only of two or less negative literals belong to $\mathcal{P}_2$. For example, $C = \{\neg P(a,b) \vee \neg Q(b)\}$ belongs to $\mathcal{P}_2$ and it subsumed with the classical definition by clause $D = \{\neg P(x,b) \vee \neg Q(y) \vee \neg P(a,y)\}$ (but not subsumed by D with the definition as multisets) that does not satisfy the length condition and does not belong to $\mathcal{P}_2$.
- $\mathcal{P}_3 = \{S(a,x)\}$. Since $S(a,x)$ is not maximally general, $\mathcal{P}_3$ is not stable. For example, $S(a,b)$ belongs to it and the subsuming clause $S(y,z)$ does not.

Definition 17. (Marked tableau, solved tableau).The definitions of clausal and connection tableau are the same as in section 2.2, but we have to adapt the definition of *marked tableau* in a similar way as we did for the intermediate lemma extension. A *marked tableau* is a clausal tableau in which some of the leaf nodes are marked with *closed* or *skipped.* The unmarked leaves are called *subgoals.* A node is *solved* if either itself is a marked leaf node or all the leaf nodes of branches through it are marked. A tableau is *solved* if its root node is solved. The literal of a skipped node is called *skipped literal* or *s-literal* and we denote the set of skipped literals in a tableau T as $\mathit{skip}(T)$. This is a set of literals (not a multiset) and usually is identified with a clause.

Definition 18. (Regular, complement-free, tautology-free tableau). As in section 2.2, a tableau is said to be *regular* if no two nodes in a branch are labeled with the same literal. A tableau is *tautology-free* if all its tableau clauses do not have a pair of complementary literals and is *complement-free* if no two non-leaf nodes on a branch are labeled with complementary literals. Additionally, the definitions of regularity and tautology-freeness can be extended to s-literals: a tableau is *regular for s-literals* if it does not contain any literal L (L does not have to be a leaf, it can be any node) such that $\neg L$ belongs to $\mathit{skip}(T)$ and it is *tautology-free for s-literals* if the clause $\mathit{skip}(T)$ does not contain any pair of complementary literals.

What follows is the description of the rules in SOL calculus and the procedure to derive a clause (find a consequence) from a given set of clauses, as an outline of a more detailed description provided by the authors in [17]. Instead or formalising the rules, we follow the authors' approach and the procedure of constructing a *SOL-deduction* in SOL tableau calculus is described as a whole. Intuitively, it is similar to the procedure of finding a proof with the intermediate lemma extension presented in the previous section. We consider the existence of a selection function ϕ, similarly to Section 2.2, that assigns a subgoal to each non-solved tableau. Given a set of clauses S, a clause $C \in S$ (start clause) and a production field $\mathcal{P}$, a *SOL-deduction of a clause C'* via ϕ consists of a sequence of tableau $T_0, T_1, \ldots, T_n$ such that:

1. T_0 is the tableaux of the start clause C, i.e., it contains only a root node whose immediate successors are the literals of C, all of them unmarked.
2. T_n is solved and $C' = \mathit{skip}(T_n)$
3. For each T_i, $0 \leq i \leq n$, $\mathit{skip}(T_i) \in \mathcal{P}$, T_i is regular, tautology-free and complement-free, and also regular and tautology-free for s-literals.
4. If $K = \phi(T_i)$ is the subgoal selected by ϕ, then T_{i+1} is the resulting tableau of applying one of the following rules to T_i:
   - *Skip:* if $\mathit{skip}(T_i) \cup K \in \mathcal{P}$, then mark K with `skipped`.
   - *Skip-factoring (and skip-merge):* if $\mathit{skip}(T_i)$ contains a literal L and K and L are unifiable with mgu σ, then mark K with *skipped* and apply σ to T_i. If σ is the identity substitution, this rule is called *skip-merge.*
   - *Extension*: select a clause $C = L_1 \vee \ldots \vee L_n \in S$ and rename the variables to make them disjoint from the variables occurring in T_i, and such that L_i and $\overline{K}$ are unifiable with mgu σ for some $1 \leq i \leq n$. Attach new successor nodes to K, mark L_i with *closed* and apply σ to all the literals in T_i.
   - *Reduction:* if K has an ancestor node L unifiable with $\overline{K}$ with mgu σ, then mark K with *closed* and apply σ to all the literals in T_i.

In Figures 3.2 and 3.3 we can see a graphical description of the above rules.

Notice that the last two rules, extension and reduction, are exactly the same defined in Section 2.2 for connection tableau calculus. The new rule *skip* is special for consequence finding and *skip-factoring*

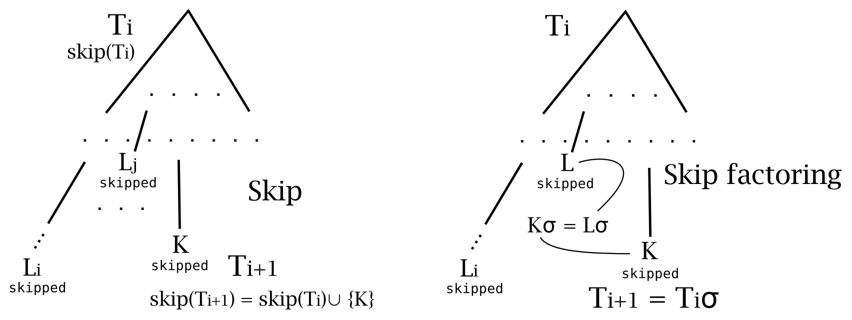


Figure 3.2: Rules *skip* and *skip-factoring*

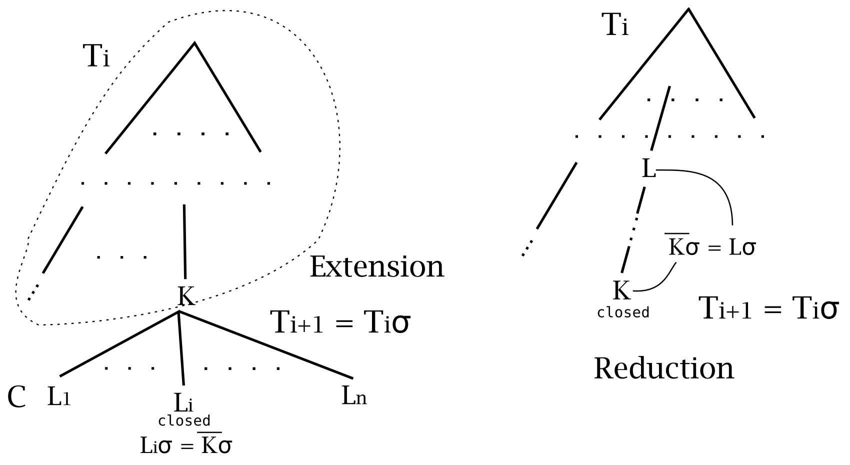


Figure 3.3: Rules *extension* and *reduction*

can be regarded as a variant of the re-use shortcut described in Section 2.2.2, blindly *assuming* the literal that is being skipped to be false (instead of having proved it, as in re-use) and then using it to skip a new subgoal.

Proposition 4. *(Soundness and completeness of the SOL Tableau Calculus)*

The SOL tableau calculus is sound and complete (for any selection function ϕ):

- *If C is derived from S and $\mathcal{P}$ by a SOL-deduction via ϕ, then $C \in Th_{\mathcal{P}}(S)$*
- *If $C \in Th_{\mathcal{P}}(S)$, then there is a SOL-deduction of a clause C' from S and $\mathcal{P}$ via ϕ such that C' subsumes C.*

The final step of the process of consequence finding is to find out all new characteristic clauses by means of some enumeration procedure. Contrary to proof finding, in consequence finding when we find a solved tableau, we have to continue looking for other solved tableau that produce different clauses. As we mentioned before, we are not interested in generating consequences that are subsumed by other because it would mean producing redundant clauses. For that reason, the enumeration procedure should find all *minimal* consequences. We will define precisely the set of clauses that will be generated and show the enumeration procedure for finding all of them.

Definition 19. (SOL-derivatives). The set of *SOL-derivatives* from a given a set of clauses $S \cup \{C\}$ (C is the start clause), a production field $\mathcal{P}$ and a selection function ϕ is the set $\Delta(S, C, \mathcal{P}, \phi) = \{D :$ D is derived using a SOL-deduction from $S \cup C$ and $\mathcal{P}$ via $\phi\}$. The set of production clauses from

$S \cup C$ and $\mathcal{P}$ via ϕ is

$$Prod(S, C, \mathcal{P}, \phi) = \mu\Delta(S, C, \mathcal{P}, \phi)$$

Finally, to generate the previous set, we proceed as follows: create a tableau T_0 with the start clause C only and initialise $Conqs = \{\}$ as a global variable, then call the procedure $EnumConqs(T_0)$ defined below.

Definition 20. (Procedure $EnumConqs(T)$).

```
if $Conqs = \{\Box\}$ then return and end
if $T$ is a solved tableau, then:
    • $C := skip(T)$
    • $Conqs := \mu(Conqs \cup C)$
return and end
$K := \phi(T)$
$R := \{r : r$ is an inference rule applicable to $K\}$
for each $r \in R$ do:
    • $T'$ :=tableau obtained from $T$ by applying $r$ to $K$
    • call $EnumConqs(T')$
end
```

For any $S, C, \mathcal{P}$ and ϕ, if $Conqs$ is a set of consequences obtained by the above procedure then

$$Conqs = Prod(S, C, \mathcal{P}, \phi)$$

Despite all the similarities shared by the described SOL tableau calculus and the connection tableau calculus with the intermediate lemma extension, they are not equivalent when taking $\mathcal{P} = \langle \mathcal{L}^+, true \rangle$. There are some substantial differences between them. The new introduced skip rule is not exactly as marking a positive literal as skipped to be part of a lemma. In SOL tableau calculus case, we can either mark a literal as skipped or use any other applicable rule (it can be part of a consequence or not), whereas in the case of the lemma generation, each positive subgoal has to be skipped if no reduction step is available, for we are not allowed to perform an extension step on it. In addition, although both methods generate a sequence of tableau, the meaning of that sequence changes from one to another. In the context of SOL tableau calculus, all tableau are equally relevant for us to obtain a new consequence from each of them, but in the intermediate lemma extension, the last closed tableau has obviously a different significance. Despite those differences, analysing the SOL tableau calculus is worth for the purpose of this project because we will be able to adapt the idea of production fields and the pruning techniques described in the next section, to our specific context of connection tableau calculus and intermediate lemmas.

3.2.2 Pruning methods

We present in this section two pruning methods or shortcuts. The first one, *skip minimality*, is not exactly a shortcut, but a restriction of the sort of regularity or tautology-freeness. The second one is a shortcut that avoid to further explore an unsolved tableau with a current substitution σ of which we can know that it is not going to succeed.

- *Skip minimality* [17]: A tableau T is *skip-minimal* for a set of clauses S if the clause $skip(T)$ is not properly subsumed by any clause in S. The idea of this method is to prevent from pursuing unsolved tableau (to *prune* them) that will generate only non-minimal consequences with respect to consequences already derived from solved tableau, i.e., to pursue only skip-minimal tableau. This is done simply by modifying the procedure $EnumConqs(T)$ to check the skip-minimality:

  ```
  if $Conqs = \{\square\}$ then return and end
  if $skip(T)$ is not skip-minimal for $Conqs$ then return and end
  ⋮
  ```

  In this way, if a tableau T is not skip-minimal with respect to the consequences already found, then not only T but also its descendants (that is, tableau that can be obtained by T) are pruned.

- *Local failure caching* [17]: The purpose of this technique is to avoid solving a subgoal with the same or a more specific substitution. The local failure caching for consequence finding is formulated in [21], but it is not complete if the production field $\mathcal{P}$ imposes a maximum length condition over the characteristic clauses. It has been reformulated in [17], to consider this case and ensure completeness.

  We need some preliminary definitions before presenting this pruning method.

  Definition 21. (Solution substitution, failure substitution).

  Let T be a tableau and K a selected subgoal in T, let T' be a tableau obtained from T by repeated execution of the $EnumConqs(T)$ procedure. If all branches through K in T' are solved, then the composition $\sigma = \sigma_1 \dots \sigma_n$ of substitutions applied to T on the way to obtain T' is called a *solution substitution* of K at T via T'. If no proof has been found at T' or going further in $EnumConqs(T')$, then the composition σ is called a *failure substitution* for K at T.

  The local failure caching procedure comprises the following steps:

  1. When a subgoal K in T has been solved via T', the solution σ is stored at K. Then:
     (a) If T cannot be completed and the proof procedure backtracks over T', σ is turned into a failure substitution.
     (b) If T has been completed at a previous stage but the proof procedure backtracks to find alternative consequences, then continue but without adding σ to the failure substitutions.
  2. In any alternative solution process of K from T, if a substitution $\tau = \tau_1 \dots \tau_m$ is computed but it is such that one of the failure substitutions stored at K, σ, is more general than τ, then the proof procedure immediately backtracks. By doing so, we avoid solving K again with a more specific substitution. If K where to be solved with τ, the final clause $skip(T)$ obtained would be subsumed by a previous consequence C generated from K using σ, because there would exist θ such that $\tau = \sigma\theta$.
  3. When the procedure backtracks at T, all failure substitution stored at K are deleted. If the proof procedure backtracks at T, that means that no consequence has been found following this path and therefore an alternative path has to be tried but at a node above K.

This method may not be as straightforward as the previous, so we provide an example to illustrate it and explain it more clearly.

Example 11. Let C, $\mathcal{P}$ and S be the following start clause, production field and set of clauses respectively.

$$C = P(x) \vee Q(x,y) \vee R(c), \;\; \mathcal{P} = \langle \mathcal{L}^+, P(x) \rangle, \;\; S = \{\neg P(a), \neg Q(x,y) \vee R(b), \neg R(b)\}$$

Recall that the condition $P(x)$ in $\mathcal{P}$ indicates that only clauses with literals with predicate P of arity 1 can be skipped.

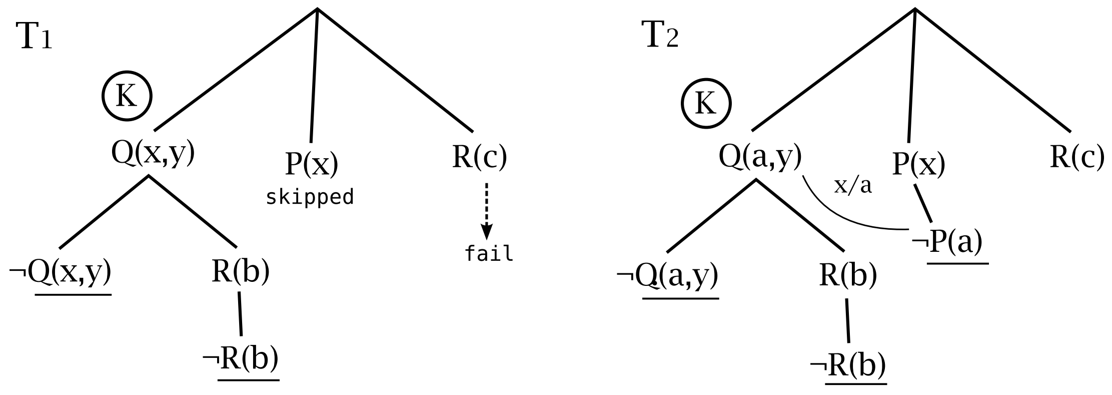


Figure 3.4: Example of local failure caching

Consider tableau T_1 in figure Figure 3.4. We solve the node $Q(x, y)$ using two extension operations and skip the node $P(x)$ but we cannot close $R(c)$, so we store the substitution $\sigma = \{\}$ [1] as a failure substitution to the node $Q(x, y)$ (step 1. (a)). When the procedure backtracks looking for alternatives, the only one is to solve $P(x)$ by means of an extension step instead of skipping it (see tableau T_2 in Figure 3.4). Then the substitution $\tau = \tau_1 = \{x/a\}$ is computed and as σ is more general than τ_1, the proof procedure immediately backtracks (step 2), without checking whether $R(c)$ can be closed or not.

Local failure caching as stated turns out to be incomplete if the condition in $\mathcal{P}$ is a maximum length condition. This can be shown with an example.

Example 12. As before, consider the following start clause C, production field $\mathcal{P}$ and set of clauses S.

$$C = P(x) \vee Q(x) \vee R(x), \ \ \mathcal{P} = \langle \mathcal{L}^+, \text{ length is less than 3 } \rangle, \ \ S = \{\neg Q(x) \vee S(x), \neg S(y)\}$$

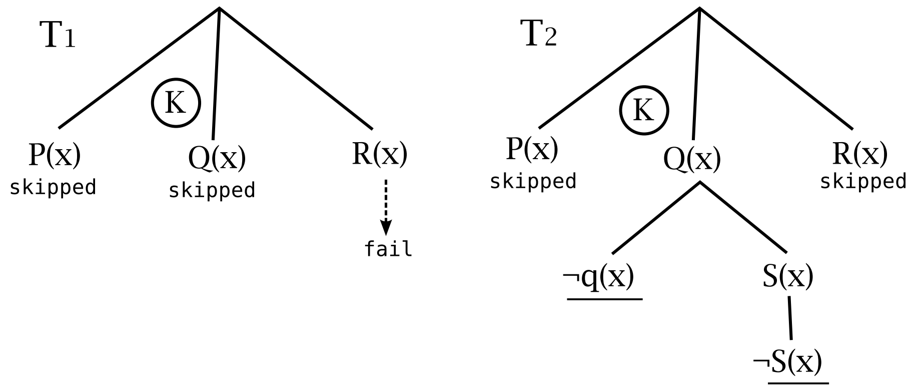


Figure 3.5: Example of local failure caching with length condition

We can apply skip to the first nodes $P(x)$ and $Q(x)$ of C (see tableau T_1 in Figure 3.5) but then we cannot apply skip to $R(x)$ because of the length condition (we would obtain length

[1]As we have done before, we omit the renaming and substitution for simplicity. The substitution $\sigma = \{\}$ is in fact $\sigma = \{x/x_1, y/y_1\}$ where x_1, y_1 are fresh variables introduced to rename clause $\neg Q(x, y) \vee R(b)$

skip(T_1) with length not less than 3). Therefore the procedure fails and the failure substitution $\sigma = \{\}$ is stored at the node K $(Q(x))$. The empty substitution is the most general, so the procedure backtracks at $Q(x)$ for any alternative rule, without finding the solution in tableau T_2 in Figure 3.5.

In order to avoid the incompleteness in those cases, the notion of failure substitution and generality between substitutions has to be extended:

Definition 22. (Extended solution substitution, extended failure substitution).

Let T be a tableau and K a selected subgoal in T, let T' be a tableau obtained from T by repeated execution of the *EnumConqs*(T) procedure. If all branches through K in T' are solved, then the pair $\langle \sigma, s \rangle$ where σ is the composition $\sigma = \sigma_1 \dots \sigma_n$ of substitutions applied to T on the way to obtain T' and $s = skip(T')$ is called an *extended solution substitution* of K at T via T'. If no proof has been found at T' or going further in *EnumConqs*(T'), then the pair $\langle \sigma, s \rangle$ is called an *extended failure substitution* for K at T. For two extended failure substitutions, $\langle \sigma_1, s_1 \rangle$ is more general than $\langle \sigma_2, s_2 \rangle$ if σ_1 is more general than σ_2 and s_1 subsumes s_2.

Example 13. We consider again the previous example to show how this extension works. When $r(x)$ cannot be closed in tableau T_1 the extended failure substitution $\langle \{\}, \{P(x), Q(x)\} \rangle$ is stored at $Q(x)$ and the procedure backtracks, solving now $Q(x)$ performing two extension steps (see tableau T_2 in Figure 3.5). The extended solution substitution $\langle \{\}, \{P(x)\} \rangle$ is computed at $Q(x)$ and compared with $\langle \{\}, \{P(x), Q(x)\} \rangle$, which is not more general than $\langle \{\}, \{P(x)\} \rangle$ because $\{P(x), Q(x)\}$ does not subsume $\{P(x)\}$. Hence, the proof procedure is not automatically backtracked, $R(x)$ is skipped and we can get the consequence $R(x) \vee P(x)$ as *skip*(T_2).

In the cases in which $\mathcal{P}$ does not have a maximum length condition, this extended definition can be ignored and the simpler one used.

Our motivation for presenting this pruning techniques developed for the SOL Tableau Calculus is based on the similarities shared by the lemma generation and consequence finding tasks. As part of this project, we attempt to adapt and incorporate skip minimality and local failure caching into a prover based on Connection Tableau Calculus with the Intermediate Lemma Extension. This attempt is thoroughly described in Section 6.2.

3.3 Restricting backtracking in connection calculi

In this section we discuss a very interesting pruning technique in connection calculi introduced by Jens Otten in [32] for the connection method that they implemented in leanCoP (see Section 2.3.4), and incorporated into the version 2.0 of this prover. We adapt slightly the concepts described in [32] to explain them in the context of the connection tableau calculus.

When searching for a closed tableaux for a given set of formulas, backtracking has to be applied if there is more than one rule that can be applied to a node. This is true in general for all calculi that are not proof confluent, in contrast to saturation-based calculi such as resolution. Once a rule is chosen, if it does not lead to a closed tableau, the algorithm has to backtrack to this point in the search and choose a different rule. In connection tableau calculus, the search requires backtracking in the following cases:

- For different clauses in the tableau start rule.
- For different ancestors in the tableau reduction rule.
- For different clauses and different successors in the tableau extension rule.

- More than one of the rules are applicable at the same time.

Moreover, since the substitution σ constructed during the search is fixed for the entire derivation, it is not only relevant *whether* a branch is closed, but also *how* is closed. The application of a different rule at a node might result in a different substitution in another branch, so it may be necessary to perform backtracking even for branches that have already been closed.

For an unsatisfiable set clauses there may exist several closed connection tableau, and even though in many applications like Logic Programming finding all of them is desirable, in theorem proving we are interested in just one of them. It is reasonable then to raise the question of whether all backtracking carried out during a search in connection calculus is equally relevant. This is the motivation of next section.

3.3.1 Analysis of backtracking in connection tableau

After some previous definitions, we outline here the analysis about the amount of backtracking needed for finding proofs in connection calculus carried out in [32], over problems from the TPTP Library ([51] and Section 8.1).

Definition 23. (Principal literal, solved literal). When the reduction or extension rule (described in Section 2.2.1) is applied to a subgoal K, we called K the *principal literal* of the step. A reduction step *solves* a literal K iff K is the principal literal of the step. An extension step *solves* a literal K iff K is the principal literal of the proof step and there exist a closed sub-tableau for each of the subgoals left by the extension step.

A solved literal corresponds to a closed branch and within a derivation, the application of a rule deletes the principal literal of the step and solves it if any new branches opened by the step can be solved as well.

This definitions are illustrated in Example 14

Example 14. Consider the tableau in Figure 3.6. $\neg S(x)$ is the principal literal of the extension step at node K_1, $Q(y)$ is the principal and solved literal of the extension step at node K_2 and finally $S(z)$ is the principal and solved literal of the reduction step at node K_3.

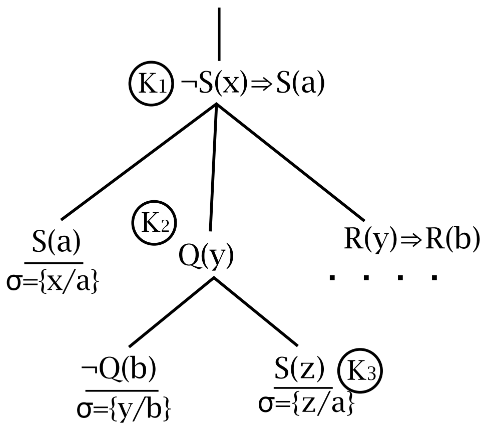


Figure 3.6: Example of principal and solved literals in reduction and extension steps

For the analysis in [32], the 17 problems in the AGT domain of the TPTP Library for which leanCoP is able to find a proof are used. Each connection proof found is studied in terms of the possible choices of rules/clauses in every step and the choice that contributes to the final proof. The author remarks two observations from the analysis:

- There are between 43 and 49 alternatives for the choice of a start clause and in 15 of the 17 proofs, the *first* chosen clause results in a successful proof search.
- There are 81 reduction or extension steps in the 17 proofs. For 79 of them, the *first* applicable rule that solves a literal (that is obviously not the first choice in general) is the one used in the final proof.

Those observations suggest a distinction between backtracking that happens before a literal is first solved and backtracking that occurs afterwards, presented in the following definition.

Definition 24. (Essential and non-essential backtracking/step). Let $R_1, \ldots, R_n$ be instances of extension/reduction rules whose principal literal is L. If L can be solved by applying the rule R_i but not the rules $R_1, \ldots, R_{i-1}$, then backtracking over the rules $R_1, \ldots, R_{i-1}$ is called *essential backtracking* and backtracking over the rules $R_{i+1} \ldots R_n$ is called *non-essential backtracking.* The application of R_i is called *essential step* and the application of any of the rules $R_{i+1} \ldots R_n$ is called *non-essential step.*

See Example 15 for clarification.

Example 15. Consider the set of clauses $\{\neg S(x), S(a) \vee Q(y) \vee R(y), \neg Q(b) \vee S(z), \neg Q(c), \neg R(c)\}$ and the tableau in Figure 3.7 after 3 extension steps. At node K_3 we have two alternatives: performing either a reduction step with $\sigma = \{z/a\}$ or an extension step using a copy of the clause $\neg S(x)$. Since the application of the reduction step already solves the rule, this is an essential step and backtracking over the second choice is non-essential. Likewise at node K_2, we can apply the extension rule either with clause $\neg Q(b) \vee S(z)$ (depicted) or with clause $\neg Q(c)$. As the first one already solves the literal $Q(y)$ and it is an essential step, backtracking to try clause $\neg Q(c)$ is a non-essential backtracking step. In contrast, at node K_1, we have the alternatives of choosing clause $S(a) \vee Q(y) \vee R(y)$ or clause $\neg Q(b) \vee S(z)$. As we can see, the application of the first one fails because we are not able to solve the subgoal $R(y) \Rightarrow R(b)$. Then backtracking over the next choice, $\neg Q(b) \vee S(z)$ is essential backtracking.

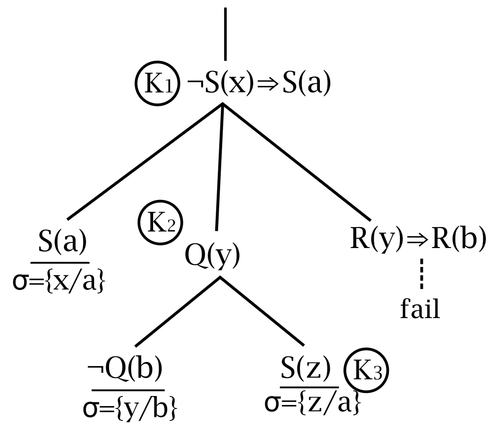


Figure 3.7: Example of essential and non-essential backtracking/step

3.3.2 Restricted backtracking

The main idea behind restricting backtracking in connection calculi is to allow only essential backtracking in reduction/extension steps, and in this way, backtracking is not carried out once a literal has been solved. Furthermore, the start step can be restricted to the first chosen start clause and any additional steps introduced by shortcuts such as factoring or re-use (Section 2.2.2) are restricted in the same way.

Restricted backtracking and restricted start step preserve correctness of the calculus but do not preserve completeness.

Example 16. It is easy to see that completeness is lost with an example. Consider the set of clauses of example 15. With restricted backtracking, we are only allowed to backtrack over the alternatives at node K_1, as we explained before. However, the only choice left, an extension step with clause $\neg Q(b) \vee S(z)$ does not lead to a closed tableau, as observed in tableau T_1 Figure 3.8. If we are restricting also the start step, at this point we have no alternatives left. Because backtracking at node K_2 in Figure 3.7 was non-essential and hence not allowed, we can not find the closed tableau T_2 in Figure 3.8.

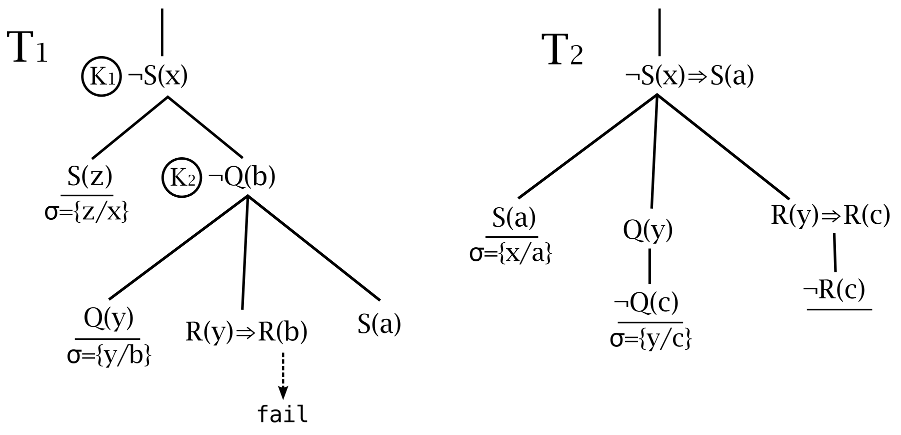


Figure 3.8: Completeness not preserved by restricted backtracking

Although the described restriction seems to be very strict, of the 1256 FOF TPTP (release v.3.7.0) problems that leanCoP is able to solve, for 882 the proof found contains only essential steps. Moreover, of the 374 problems in which a proof is found using non-essential steps, a different proof (of a different length and/or using different clauses and rules) with only essential steps can be also constructed for 218 using a higher time limit. If we refer to J. Otten's analysis, the version of leanCoP with restricted backtracking is able to solve 330 new problems for which the *regular* leanCoP fails to find a proof.

leanCoP's authors have tested the performance of both versions, regular and restricted. The restricted version solves 1560 FOF problems using an average time of 2.6 seconds, whereas the regular version solves 1256 with an average time of 3.6 seconds. Restricting backtracking seems to be specially helpful in the case of problems with equality and with a higher difficulty rate. The best version of leanCoP is one that combines restricted and unrestricted backtracking by means of a strategy scheduling. See [32] for more details and complete performance results.

3.3.3 Restricting backtracking with the intermediate lemma extension

In light of the results obtained for leanCoP by incorporating this restriction in spite of loosing completeness, it would be a good improvement to the connection tableau calculus with the intermediate lemma extension to introduce a similar restriction in the proof search. However, a simple adaptation of the definitions and restrictions to our framework with only minor changes is not apparently possible. The reason is that the intermediate lemma extension introduces its own constraints in the calculus, which in conjunction with restricted backtracking lead to a much higher *level of incompleteness.* Many proofs that can be constructed with only essential backtracking in the original tableau calculus are impossible to find when adding the intermediate lemma extension's constraints. Example 17 illustrates this problem.

Example 17. Consider the set of clauses $S = \{\neg P(a), \underline{Q(e)}, \underline{P(y)} \vee \neg Q(z) \vee P(z), \underline{Q(a)}\}$, where we have underlined the positive literals chosen as conclusion literals for the intermediate lemma extension and suppose that we attempt to find a closed tableau only allowing essential backtracking. We only have one possible start clause, $\neg P(a)$, and then only the choice of applying an extension step using clause $\underline{P(y)} \vee \neg Q(z) \vee P(z)$ at K_1. After that, at node K_2 we have two alternatives for the next extension step, $\underline{Q(e)}$ and $\underline{Q(a)}$. If we choose the first one, this is an essential step that solves the literal $Q(z)$ so we are not able to backtrack and try $\underline{Q(a)}$. The substitution $\sigma = \{z/a\}$ causes that the literal $P(z) \rightarrow P(a)$ can not be closed, as observed in tableau $T1$ in Figure 3.9. Since we are not able to backtrack at K_2 because it would be non-essential, and above this there is no more alternatives for backtracking, the proof search fails.

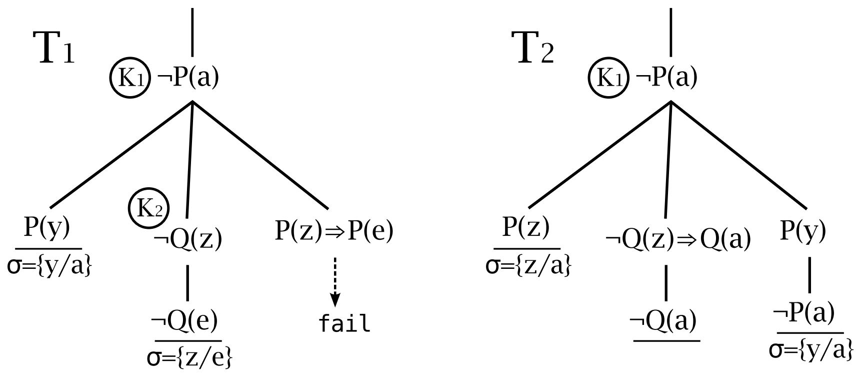


Figure 3.9: Restricting backtracking with the intermediate lemma extension

However, if we consider the same set without any restriction, $S = \{\neg P(a), Q(e), P(y) \vee \neg Q(z) \vee P(z), Q(a)\}$, after failing at K_3, we have one alternative left at K_1, using the clause $P(y) \vee \neg Q(z) \vee P(z)$ with successor $P(z)$ for the extension step. This leads to the closed tableau T_2 in Figure 3.9. Notice that backtracking at K_1 is essential, and the extension step performed at that node with successor $P(z)$ is essential as well.

The features of the intermediate lemma extension introduce restrictions over the backtracking that preserve completeness. Nevertheless, when combined with the limitations of permitting only essential backtracking, the overall restriction over the proof search becomes too strong. The key is that by fixing the conclusion literal on each clause, the flexibility of using the same clause in different ways is lost.

Chapter 4

Equality Reasoning in Tableau Calculus

Contrary to many kinds of relations, such as *less than* or *depends on*, equality is an universal relation like logic connectives. Equality is part of practically every mathematical theory of interest and therefore, incorporating it into theorem provers, as a part of formal logic is desirable. Adding equality to a theorem prover make it to become quite more complex. The number of paths to explore increases dramatically, thus the employment of heuristics or techniques to deal with it. In this section we will describe the properties of equality, present the equality axioms and introduce a simple way to introduce rules for equality in tableau calculi. Some ideas about how to handle equality in tableau more efficiently will be discussed in subsequent sections.

It is usual to use the symbol $\approx$ to denote the equality relation and to avoid confusion with $=$ used at a top level to describe the logic rather than as part of the logic itself. However, we will use $s = t$ to express the equality relation between two terms s and t, providing clarification if needed. Sometimes we will abbreviate $\neg(t = s)$ by $t \neq s$.

First, we should consider informally some fundamental properties of equality. Clearly, it is an equivalence relation: reflexive, symmetric and transitive. Furthermore, equality obeys a *substitution* property, in other words, equals can be substituted or replaced with equals. Transitivity and symmetry can be deduced from reflexivity and substitution alone (see below). These important properties can be expressed by sentences of first-order logic and such sentences are often called *equality axioms*.

Definition 25. (Equality axioms).

1. $\forall x\ (x = x)$ (Reflexivity)
2. $\forall x, y\ (x = y \rightarrow y = x)$ (Symmetry)
3. $\forall x, y, z\ (x = y \wedge y = z \rightarrow x = z)$ (Transitivity)
4. $\forall x_i, y_i, 1 \leq i \leq n\ (x_i = y_i \rightarrow f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n))$
5. $\forall x_i, y_i, 1 \leq i \leq n\ (x_i = y_i \wedge P(x_1, \ldots, x_i, \ldots, x_n) \rightarrow P(x_1, \ldots, y_i, \ldots, x_n))$

f and P are respectively a n-arity function symbol and a n-arity predicate symbol. The last two axioms are substitution schema. There is one axiom for each argument position for each function and predicate symbol (that is, n axioms for an n-ary function/predicate symbol). An alternative and equivalent (see proof below) version of those two axioms adds only one axioms for each predicate symbol:

4'. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$

5'. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1, \ldots, x_n) \rightarrow P(y_1, \ldots, y_n))$

In this report we use both versions, depending on the context.

As we said, symmetry and transitivity can be deduced from reflexivity and substitution, if we consider the substitution axioms for the *equality* predicate, that is,

(Eq1) $\forall x, y(x = y \wedge x = z \rightarrow y = z)$

(Eq2) $\forall x, y(x = y \wedge z = x \rightarrow z = y)$

or in the second version,

(Eq3) $\forall x_1, x_2, y_1, y_2(x_1 = y_1 \wedge x_2 = y_2 \wedge x_1 = x_2 \rightarrow y_1 = y_2)$

For symmetry assume $x = y$ and $x = x$. Using (Eq1) $x = y \wedge x = x \rightarrow y = x$ we can conclude $y = x$ and hence symmetry. For transitivity, assume $x = y$, $y = z$ and $x = x$. Using (Eq3) $x = x \wedge y = z \wedge x = y \rightarrow x = z$ we have $x = z$ and hence transitivity.

We can write the equality axioms as clauses, which are more convenient for our purposes (recall that all variables are implicitly universally quantified).

1. $x = x$
2. $\neg(x = y) \vee y = x$
3. $\neg(x = y) \vee \neg(y = z) \vee x = z$
4. $\neg(x_i = y_i) \vee f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n))$
5. $\neg(x_i = y_i) \vee \neg P(x_1, \ldots, x_i, \ldots, x_n) \vee P(x_1, \ldots, y_i, \ldots, x_n)$

And the alternative version for the substitution schema:

4'. $\neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n) \vee f(x_1, \ldots x_n) = f(y_1, \ldots, y_n)$

5'. $\neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n) \vee \neg P(x_1, \ldots, x_n) \vee P(y_1, \ldots, y_n)$

Proposition 5. *(Equivalence of the substitution equality axioms). The two version of the substitution equality axioms are equivalent.*

Proof. (Equivalence of the substitution equality axioms).

For simplicity our proof uses the first-order logic formulation, instead of the clausal form.

Function symbol substitution schema.

$\Rightarrow$ Assume that for each $i, 1 \leq i \leq n$, we have $\forall x_i, y_i\ (x_i = y_i \rightarrow f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n))$ and show $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$. Suppose that $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n)$ holds. We show $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ by successively applying the n substitution axioms for f as follows:

1. $x_1 = y_1$ and $\forall x_1, y_1\ (x_1 = y_1 \rightarrow f(x_1, \ldots, x_i, \ldots, x_n) = f(y_1, \ldots, x_i, \ldots, x_n))$ imply $f(x_1, \ldots, x_i, \ldots, x_n) = f(y_1, \ldots, x_i, \ldots, x_n)$.
2. Step 1, $x_2 = y_2$ and $\forall x_2, y_2\ (x_2 = y_2 \rightarrow f(y_1, x_2, \ldots, x_i, \ldots, x_n) = f(y_1, y_2 \ldots, x_i, \ldots, x_n))$ imply $f(x_1, x_2, \ldots, x_i, \ldots, x_n) = f(y_1, y_2, \ldots, x_i, \ldots, x_n)$.

   $\vdots$

i. Step $i-1$, $x_i = y_i$ and $\forall x_i, y_i\ (x_i = y_i \rightarrow f(y_1, \ldots, y_{i-1}, x_i, \ldots, x_n) = f(y_1, \ldots, y_i, \ldots, x_n))$ imply $f(x_1, \ldots, x_{i-1}, x_i, \ldots, x_n) = f(y_1, \ldots, y_{i-1}, y_i, \ldots, x_n)$.

   $\vdots$

n. Step $n-1$, $x_n = y_n$ and $\forall x_n, y_n\ (x_n = y_n \rightarrow f(y_1, \ldots, y_i, \ldots, x_n) = f(y_1, \ldots, y_i, \ldots, y_n))$ imply $f(x_1, \ldots, x_i, \ldots, x_n) = f(y_1, \ldots, y_i, \ldots, y_n)$.

And thus we conclude $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1, \ldots, x_i, \ldots, x_n) = f(y_1, \ldots, y_i, \ldots, y_n))$

$\Leftarrow$ Assume that $\forall x_1, \ldots, x_n, y_1, \ldots, y_n 1\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$ and show that for each $i, 1 \leq i \leq n$, we have $\forall x_i, y_i\ (x_i = y_i \rightarrow f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n))$.

We use the reflexivity axiom, $\forall x(x = x)$ to prove each one of the n substitution axioms as follows:

1. We assume $x_1 = y_1$, that together with $x_2 = x_2, \ldots, x_n = x_n$ and $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$ imply $f(x_1, \ldots, x_i, \ldots, x_n) = f(y_1, \ldots, x_i, \ldots, x_n)$. Thus we conclude $\forall x_1, y_1\ (x_1 = y_1 \rightarrow f(x_1, \ldots, x_i, \ldots, x_n) = f(y_1, \ldots, x_i, \ldots, x_n))$.

   $\vdots$

i. We assume $x_i = y_i$, that together with $x_1 = x_1, \ldots, x_{i-1} = x_{i-1}, x_{i+1} = x_{i+1}, \ldots x_n = x_n$ and $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$ imply $f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n)$. Thus we conclude $\forall x_i, y_i\ (x_i = y_i \rightarrow f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n))$.

   $\vdots$

n. We assume $x_n = y_n$, that together with $x_1 = x_1, \ldots x_{n-1} = x_{n-1}$ and $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$ imply $f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, x_i, \ldots, y_n)$. Thus we conclude $\forall x_n, y_n\ (x_n = y_n \rightarrow f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, x_i, \ldots, y_n))$.

The reasoning to show the equivalence of the predicate symbol substitution axioms is very similar to the previous one.

$\Rightarrow$ Assume that for each $i, 1 \leq i \leq n$, we have $\forall x_i, y_i\ (x_i = y_i \wedge P(x_1, \ldots, x_i, \ldots, x_n) \rightarrow P(x_1, \ldots, y_i, \ldots, x_n)$ and show $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1, \ldots, x_n) \rightarrow P(y_1, \ldots, y_n))$.

Suppose that $\forall x_1, \ldots, x_n, y_1, \ldots, y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1, \ldots, x_n))$ holds. We show $P(y_1 \ldots, y_n)$ by successively applying the n substitution axioms for P as follows:

1. $x_1 = y_1 \wedge P(x_1, \ldots, x_n)$ and $\forall x_1, y_1\ (x_1 = y_1 \wedge P(x_1, \ldots, x_n) \rightarrow P(y_1, \ldots, x_n))$ imply $P(y_1, \ldots, x_n)$.
2. Step 1, $x_2 = y_2 \wedge P(x_1, \ldots, x_n)$ and $\forall x_2, y_2\ (x_2 = y_2 \wedge P(y_1, \ldots, x_n) \rightarrow P(x_1, y_2, \ldots, x_n))$ imply $P(y_1, y_2 \ldots, x_n)$.

   $\vdots$

i. Step $i-1$, $x_i = y_i \wedge P(x_1,\ldots,x_n)$ and $\forall x_i, y_i\ (x_i = y_i \wedge P(y_1,\ldots,x_i,\ldots,x_n) \rightarrow P(y_1,\ldots,y_i,\ldots,x_n))$ imply $P(y_1,\ldots,y_i,\ldots,x_n)$.

$\vdots$

n. Step $n-1$, $x_n = y_n \wedge P(x_1,\ldots,x_n)$ and $\forall x_n, y_n\ (x_n = y_n \wedge P(y_1,\ldots,x_n) \rightarrow P(y_1,\ldots,y_n))$ imply $P(y_1,\ldots,y_i,\ldots,y_n)$.

And thus we conclude
$\forall x_1,\ldots,x_n,y_1,\ldots,y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1,\ldots,x_n) \rightarrow P(y_1,\ldots,y_n))$

$\Leftarrow$ Assume that $\forall x_1,\ldots,x_n,y_1,\ldots,y_n 1\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1,\ldots,x_n) \rightarrow P(y_1,\ldots,y_n))$ and show that for each $i, 1 \leq i \leq n$, we have
$\forall x_i, y_i\ (x_i = y_i \wedge P(x_1,\ldots,x_n) \rightarrow P(x_1,\ldots,y_i,\ldots,x_n))$.

We use the reflexivity axiom, $\forall x(x = x)$ to prove each one of the n substitution axioms as follows:

1. We assume $x_1 = y_1 \wedge P(x_1,\ldots,x_n)$, that together with $x_2 = x_2,\ldots,x_n = x_n$ and $\forall x_1,\ldots,x_n,y_1,\ldots,y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1,\ldots,x_n) \rightarrow P(y_1,\ldots,y_n))$ imply $P(y_1,x_2,\ldots,x_n)$. Thus we conclude $\forall x_1, y_1\ (x_1 = y_1 \wedge P(x_1,\ldots,x_n) \rightarrow P(y_1,\ldots,x_n))$.

$\vdots$

i. We assume $x_i = y_i \wedge P(x_1,\ldots,x_n)$, that together with $x_1 = x_1,\ldots,x_{i-1} = x_{i-1}, x_{i+1} = x_{i+1},\ldots,x_n = x_n$ and $\forall x_1,\ldots,x_n,y_1,\ldots,y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1,\ldots,x_n) \rightarrow P(y_1,\ldots,y_n))$ imply $P(x_1,\ldots,y_i,\ldots,x_n)$. Thus we conclude
$\forall x_i, y_i\ (x_i = y_i \wedge P(x_1,\ldots,x_n) \rightarrow P(x_1,\ldots,y_i,\ldots,x_n))$.

$\vdots$

n. We assume $x_n = y_n \wedge P(x_1,\ldots,x_n)$, that together with $x_1 = x_1,\ldots x_{n-1} = x_{n-1}$ and $\forall x_1,\ldots,x_n,y_1,\ldots,y_n 1\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1,\ldots,x_n) \rightarrow P(y_1,\ldots,y_n))$ imply $P(x_1,\ldots,y_n)$. Thus we conclude $\forall x_n, y_n\ (x_n = y_n \wedge P(x_1,\ldots,x_n) \rightarrow P(x_1,\ldots,y_n))$.

□

The semantics of equality can be obtained straightforwardly from the semantics of first-order logic, simply restricting to those model in which the equality symbol behaves as such, satisfying the previous axioms. A detailed and formal description of the semantics and the proof of the Model Existence Theorem can be found for example in [14]. For practical purposes, if S if a set of clauses expressed in first-order logic with equality and we want to find a refutation for it, we will consider instead the set $S \cup Axioms$, where *Axioms* is a set containing the previous axioms for equality.

Integration of equality is a major problem of connection calculi and connection tableau in particular. Paramodulation, that is a successful technique for dealing with equality in saturation-based theorem proving, is not complete for the goal-oriented approach of connection calculi. Equality computing is in general difficult in a top-down theorem prover framework as connection tableau and not much progress has been achieved in this area.

The integration of equality just by adding the equality axioms to the input set, as explained before, has the disadvantage that in some cases, several hundreds of equality axioms are needed. This happens because apart from the three basic axioms, n substitution axioms are generated for each predicate and function symbol of arity n. If using the second version of the substitution axioms, only one is needed but even so, the search space becomes much larger. For that reason, it is desirable to develop some built-in technique for equality reasoning in the connection tableau calculus. There are some techniques of this kind and it is an important goal of this project to investigate them. The next sections contain descriptions of possible methods of integrating built-in mechanisms for equality reasoning in tableau calculus. The last one, Section 4.3 is perhaps the most relevant, as it is described the method that we have finally designed and implemented as part of our project, adapting several ideas and features described in the rest of this chapter.

4.1 Simple integration of equality in tableau

A simple way of dealing with equality in connection tableau is to formulate additional tableau rules that capture the essential properties of equality. Let t and u be distinct ground terms and $C(x)$ a literal that contains free occurrences of x. We will write $C(t)$ as shortcut for $C(x)\{x/t\}$. Then consider that $C(t)$ differs from $C(u)$ in that $C(u)$ has occurrences of u at some of the places that $C(t)$ has occurrences of t but it not necessarily the case that *all* occurrences of t have been replaced by u. For example, if $C(x) = P(x, a)$, then $C(a) = P(a, a)$ and $C(b) = P(b, a)$. Having clarified the notation, consider the following two rules as example:

- *Tableau Replacement Rule*: if $t = u$ and a literal $C(t)$ appear then $C(u)$ can be added to the end producing another tableau. That is,

$$\frac{\begin{array}{c} t = u \\ C(t) \end{array}}{C(u)}$$

- *Tableau Reflexivity Rule*: we can always add the formula $x = x$, where x is a free variable to the end of a branch, and for any n-place function symbol, we can add $f(x_1, \dots, x_n) = f(x_1, \dots, x_n)$

$$\frac{}{x = x} \qquad\qquad \frac{}{f(x_1, \dots, x_n) = f(x_1, \dots, x_n)}$$

The problem is that there are many different possibilities for closure and if we impose restrictions, completeness is not always ensured. For example, in connection tableau we could restrict the use of an equation to the cases in which it already occurs in a branch. The clauses $S = \{P(a), \neg P(b), a = b\}$ show that this restriction is not complete because we cannot find a closed tableau unless we choose $a = b$ as top clause. Also, most of the methods based on ordered equality handling are not compatible with connection conditions on tableaux, so we will have to search for some way to relax or redefine restrictions over tableau to deal with equality.

To finish this section, we provide an example illustrating the use of the above rules to find a refutation for a set of clauses with equality.

Example 18. Let S be the following set of clauses:

$$S = \{P(x) \vee Q(x), \neg P(a) \vee R(x), \neg Q(a) \vee a = b, \neg R(x), \neg Q(b)\}$$

Figure 4.1 depicts a closed connection tableau for S, with start clause $\neg R(x)$ and one application of the tableau replacement rule at the end. This tableau would be a connected one if we did not consider the application of the replacement rule, since the equality literal $a = b$ does not have its negation as immediate successor.

In Section 4.2 the problem of equality reasoning is tackled in a different way: considering another tableau calculus.

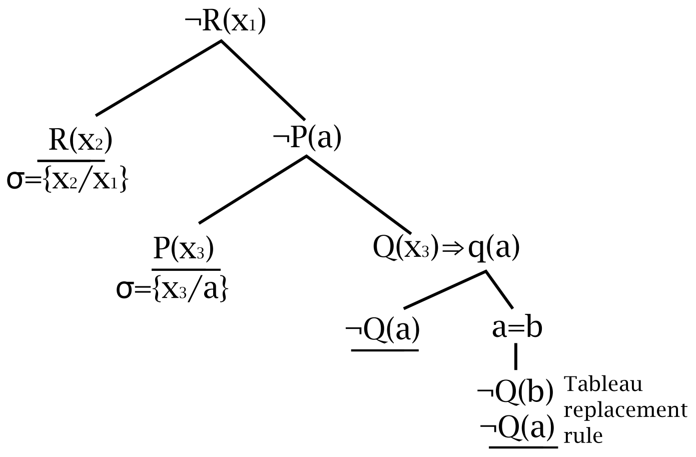


Figure 4.1: Example of equality in tableau

4.2 Equality reasoning in the Disconnection Tableau Calculus

As we briefly commented in the previous section, some constraints like connectedness or regularity are not compatible with restrictions for closure of equality literals. For that reason, in this section we present an alternative tableau calculus that does not impose connectedness or regularity conditions, the *Disconnection Tableau Calculus*. Next two different approaches to integrate equality in this new context are explained. The first one is based on the paramodulation rule adapted for this calculus and the second is a more goal oriented approach based on a variant of paramodulation.

4.2.1 The Disconnection Tableau Calculus

The Disconnection Tableau Calculus [28] is an integration of Plaisted's clause linking method [42] into a tableau structure. The static objects in this framework are clausal tableaux and the calculus consists of a single complex inference rule: the *linking rule*. Before going into more detail, we give a few basic definitions.

Definition 26. (Link, linking instance). Given two literals L and K in two variable-disjoint clauses C and D, such that L and $\overline{K}$ are unifiable with mgu σ, the set $\ell = \{L, K\}$ is called a *link* or *connection* (between the clauses C and D). The clauses $C\sigma$ and $D\sigma$ are named *linking instances* of C (respectively of D) w.r.t. ℓ.

Definition 27. (Path, branch). A *path* is a mapping π from the clauses in S to literals in S such that $\pi(C) \in C$, that is, it assigns to every clause $C \in S$ a literal that occurs in C. A tableau is started from a clause set S by choosing a path through S. A *branch* B in a tableau is a maximal sequence of nodes $B = N_1, N_2, N_3, \ldots$ such that N_1 is an immediate successor of the root and every N_i is an immediate successor of N_i. A branch defines a path through the clauses *on* the branch.

The initial path remain fixed throughout the entire tableau construction.

Next we outline the main features of the basic disconnection tableau calculus and illustrate it with an example.

- *Linking rule*: given a tableau T with branch B containing two literals K and L in tableau clauses C and D respectively, if there is a $\ell = \{L, K\}$ for L and a renamed instance of K, then expand the branch by successively renaming the linking instances of C and D ($C\sigma$ and $D\sigma$).

Example 19. The application of this rule is illustrated in Figure 4.2. The tableau branch on the left has two literals $K = P(x, b)$ in clause C and $L = \neg P(a, y)$ in clause D, such that $\overline{K}$ and L are unifiable with $\sigma = \{x/a, y/b\}$. Two renamed instances of $C\sigma$ and $D\sigma$ are attached then at the end of the branch, expanding it, as we see on the left tableau. We also mark the node $\neg P(a, b)$ as closed.

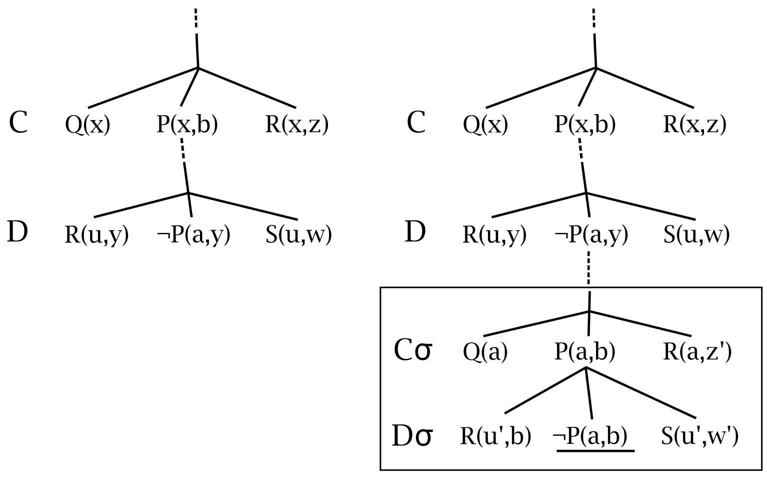


Figure 4.2: Example of application of the linking rule

As the two clauses are linked by $\ell = \{P(x, b), P(a, y)\}$ and the linking instances attached to the end of the branch, the link between them cannot be used any more on future expansions of the branch B, so applying the rule is like "disconnecting" the clauses, and hence the name of the calculus.

Definition 28. ($\forall-$closed branch, $\forall-$closed tableau). The notion of *closure for a branch* is the same as employed in the clause linking method. A branch of a tableau is $\forall-$*closed* if it contains two literals K and L such that $K\sigma = \overline{L\sigma}$ where σ is a substitution that maps all variables in the tableau to a new constant.

Example 20. Figure 4.3 depicts a $\forall-$closed disconnection tableau for the set of clauses

$$S = \{P(x, z) \vee \neg P(x, y) \vee \neg P(x, z), P(b, c), P(a, b), \neg P(a, c)\}$$

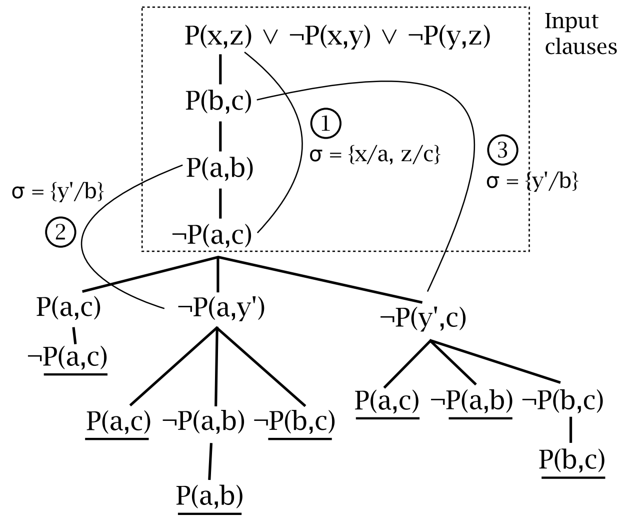


Figure 4.3: Example of a $\forall-$closed disconnection tableau

Links between literals are indicated with arcs numbered, together with the substitution σ used.

Proposition 6. *(Soundness and completeness of the Disconnection Tableau Calculus). The Disconnection Tableau Calculus is sound and complete.*

The proof can be found in [28].

What we have presented here is just an introduction of the basic calculus. There are several pruning techniques and refinements like regularity, folding-up or clause subsumption that can be incorporated into it. They are out of the scope of this report and can be consulted in [28]. The basic concepts introduced are enough for our purposes. It is possible to incorporate equality reasoning in this calculus by explicit use of the equality axioms presented at the beginning of this chapter (i.e., all equality axioms are automatically included in the input set of clauses) but in the next two sections we introduce two methods to build equality reasoning mechanism in this calculus, one of which will be the basis of the technique adopted in our prover.

4.2.2 Paramodulation in Disconnection Tableau Calculus

Definition 29. (Paramodulation). *Paramodulation* is a generalisation of equality substitution. Let r, s, t be terms. If $s = t$ and s (t) occurs in some clause C, then t (s) can replace s in any of the occurrences. In the general case, consider two clauses $C = L(t) \vee C'$ (we use $L(t)$ to denote that t occurs in L) and $D = r = s \vee D'$, where r and t are unifiable with mgu σ, $r\sigma = t\sigma$. Then the clause $(C' \vee D' \vee L(s\sigma))\sigma$ is called a *paramodulant.* It can be obtained following the steps:

1. Unify r with t with unifier σ.
2. Apply σ to C and D, getting $C\sigma$ and $D\sigma$.
3. Replace t in $C\sigma$ (that is, in $L(t)$) by $s\sigma$.
4. The disjunction of $C\sigma$ and $D\sigma$ after replacement and without the equation is the resulting paramodulant.

One o more (or all) occurrences of t can be replaced.

Example 21. To illustrate this, consider $C = P(f(a)) \vee Q(b)$ and $D = f(x) = b \vee P(x)$.

1. $f(x)$ is unified with $f(a)$, with $\sigma = \{x/a\}$
2. $C\sigma = D = P(f(a)) \vee Q(b)$ and $D\sigma = f(a) = b \vee P(a)$.
3. $f(a)$ is replaced in C by b: $P(b) \vee Q(b)$
4. The result is $P(b) \vee Q(b) \vee P(a)$

Paramodulation is adapted to the disconnection calculus as the *eq-linking rule.*

- *Eq-linking rule*: given a tableau branch B that contains an equation $s = r$ and a literal $L(t)$ in tableau clauses C and D respectively, if there exists an unifier σ for r and t, then successively expand the branch with $C\sigma$ and $s \neq r \vee paramodulant(C, D)$.

  Notice that there is a source of indeterminism in the application of this rule, because in general for two clauses C and D, $paramodulant(C, D)$ is not necessarily unique.

Two independent clauses are introduced and the negation of the equation is added to ensure the soundness of inferences ([9] for more details). Additionally, if the reflexivity axiom $x = x$ has not been added to the initial path, another rule is needed to allow closure of branches with $s \neq t$.

- *Reflexivity linking*: given a tableau branch B with $s \neq t$ in clause C such that s and t are unifiable with σ, then expand the branch with a renamed instance of $C\sigma$.

And now we can define a $\forall-$closure rule for branches on which $t \neq t$ appears.

The application of above rules as they are stated, without any restriction, leads to generation of many redundant clauses. It is usual when dealing with equality to introduce some ordering in the Herbrand universe to control the symmetry property of equality, to replace terms with smaller terms and not the opposite. Then overlapping is allowed with the maximal side of the equation and in case the overlapped literal is itself an equality literal, into maximal sides only (the ordering is partial so in case two sides cannot be ordered, both are considered maximal).

Example 22. Figure 4.4 represents an example of a refutation with ordered eq-linking for the input clauses

$$S = \{f(x) = g(x) \vee R(f(x)), P(f(x)) \vee R(f(x)), \neg P(g(b)), \neg R(f(b))\}$$

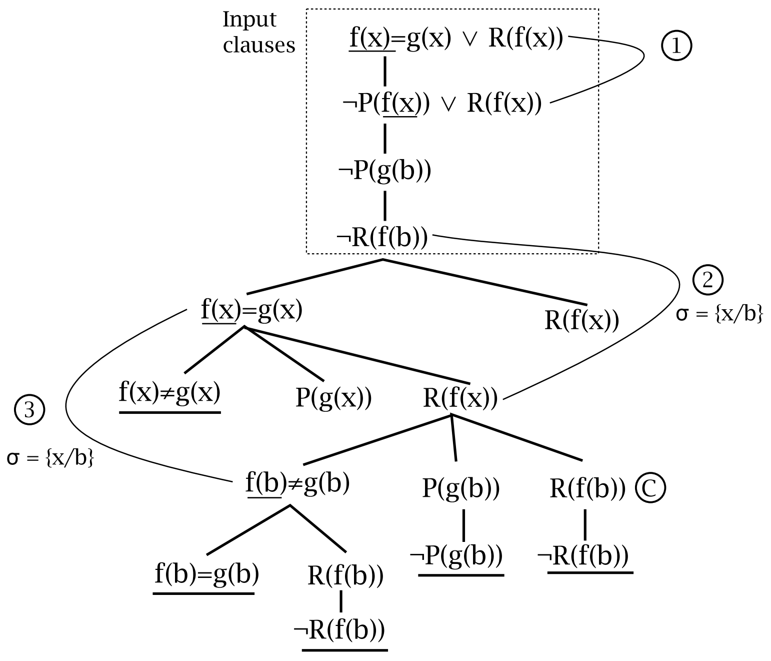


Figure 4.4: Example of a disconnection proof with ordered eq-linking

We take $f > g$ for term ordering and the selected sides of the overlapping equations and overlapped terms are underlined in the tableau. For simplicity, only the links that do not involve unit clauses are indicated and renaming of clauses are omitted. When applying link (1), $P(g(x))$ is valid only under the assumption $f(x) = g(x)$ and this is expressed by adding $f(x) \neq g(x)$ to the clause. When this clause is used for the linking step (2), a new condition is required for $f(b) \neq g(b)$ but we have to prove it explicitly. A eq-linking step, numbered with (3) is performed. Notice that only one of the clauses resulting from that eq-linking step is written, $f(b) = g(b) \vee R(f(b))$. The other one, $f(b) \neq g(b) \vee g(b) \neq g(b) \vee P(g(b)) \vee R(f(b))$ is variant-subsumed by clause C, but $f(b) = g(b)$ is already $\forall-$closed by its path. Finally, we leave open two nodes $P(g(x))$ and $R(f(x))$, but their proofs are identical to the subproof below link application (2).

Eq-linking in tableaux is complete ([28]) but turns out to be incompatible with the regularity condition. For more details on this see [29].

4.2.3 A variant of lazy paramodulation

In this section we describe briefly an attempt to develop a goal-oriented technique for equality handling, sacrificing compatibility with term ordering.

Paramodulation, and hence eq-linking, introduces two forms of indeterminism: one is the number of equations that can be applied to a term and the other is the number of occurrences of a term that we can replace. The first cannot be avoided but *lazy paramodulation* provides a way to get rid of the second. A restriction is introduced, to allow replacement in root terms only. As the authors did in [29], for the sake of simplicity only the pure equality case (when the equality sign is the only *predicate* symbol) is considered in this section. The extension to the general case is straightforward and we comment it briefly. It will be more clear in Section 4.3.

This method of *lazy root paramodulation*, as the authors have termed it, is centred around the concepts of *disagreement set* and *disagreement substitution* and it is closely related with Digricoli's RUE resolution [13]. Some differences and similarities between them will be further explored in Section 4.3, but here just the main features of lazy root paramodulation are outlined and illustrated.

Definition 30. (Disagreement set, disagreement substitution). Given two terms of the form $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$, their *disagreement set* is the clause $s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n = \{s_1 \neq t_1, \ldots, s_n \neq t_n\}$ and their *disagreement substitution* is the empty set. The *disagreement set* of two terms x and t where x is a variable that does not occur in t is the empty set and their *disagreement substitution* is $\sigma = \{x/t\}$. Otherwise it is not defined.

The previous definition leads to two new rules for the disconnection calculus:

- *Disagreement linking*: given a tableau branch B with literals $K = s \neq s'$ and $L = t = t'$ in tableau clauses C and D, respectively, such that the disagreement set of s and t is $\mathcal{D}$ with a disagreement substitution σ, then successively expand the branch with renamed instances of $C\sigma$ and $D'\sigma$ where $D' = \mathcal{D} \vee (D \setminus \{L\}) \vee s' \neq t' \vee s = s'$. To ensure completeness, another rule is needed:
- *Decomposition*: given a tableau branch B containing a literal $L = s \neq t$ in tableau clause D such that s and t have a disagreement set $\mathcal{D}$ and a disagreement substitution σ, then successively expand the branch with renamed instances of the clause $D'\sigma$ where $D' = \mathcal{D} \vee (D \setminus \{L\})$

In the general case of P being any n-ary predicate symbol we consider $K = \neg P(s_1, \ldots, s_n)$ and $L = P(t_1, \ldots, t_n)$ literals in clauses C and D such that each pair s_i, t_i has a disagreement set $\mathcal{D}_\rangle$ with disagreement substitution σ_i. Then the branch is expanded with $C\sigma$ and $D'\sigma$ where $D' = P(s_1, \ldots, s_n) \vee \mathcal{D} \vee (D \setminus \{L\})$ and

$$\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_i \quad \sigma = \bigcup_{i=1}^{n} \sigma_i$$

Example 23. In Figure 4.5 we can find an example of a proof that uses disagreement linking, for the set of clauses

$$S = \{\underline{h(a)} \neq c, \underline{h(e)} = d, h(f) = \underline{h(b)}, a = b, c = d, e = f\}$$

All redundant clauses and subgoal have been omitted for simplicity. As before, selected sides of equations have been underlined and the disagreement sets D and substitutions σ are indicated together

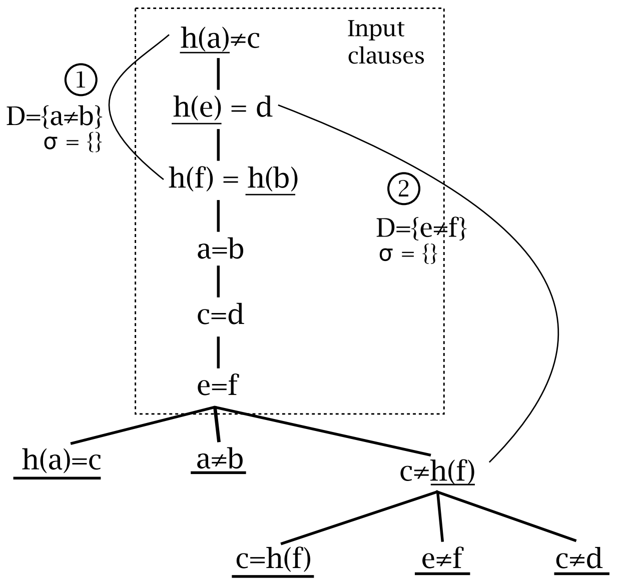


Figure 4.5: Example of a disconnection proof with disagreement linking

with the disagreement link number. Let us explain in more detail the link (1). We have literals $K = h(a) \neq c$ and $L = h(f) = h(b)$. The disagreement set of $h(a)$ and $h(b)$ (those are the selected sides) is $\{a \neq b\}$ and their disagreement substitution is $\sigma = \emptyset$. We expand then the branch with clauses $h(a) \neq c$ (omitted for clarity, as it is the same as K) and $a \neq b \vee c \neq h(f) \vee h(a) = c$

Similarly to paramodulation, the reflexivity axiom $x = x$ needs to be added to the input set to achieve completeness.

R. Letz and G. Stenz have implemented both methods, paramodulation and lazy paramodulation (in a more complex and sophisticate way than we have described) in a theorem prover called DCTP [29].

4.3 Built-in equality mechanism in Connection Tableau Calculus

The latter method of lazy paramodulation represents a very convenient method for implicitly dealing with equality as alternative to include the equality axioms in the input set. However, for our purposes it has the main disadvantage of being based in a different calculus. Although we could *forget* the connection tableau calculus and the intermediate lemma extension and implement a completely different version of the theorem prover based on the disconnection calculus, what we really want is to have such built-in mechanisms integrated with the intermediate lemma extension in our current framework.

We have already mentioned that the approach of lazy paramodulation is closely related to RUE resolution. In next section, when we study RUE resolution more deeply we will see that what the authors have done is to adapt the RUE rules to a tableau setting. Following Letz and Stenz's idea, we have adopted the mechanism of RUE rules as alternative to explicit use of the axioms and have adjusted it to the connection tableau with the intermediate lemma extension framework. Details concerning this matter can be found in Section 4.3.2.

4.3.1 RUE resolution

RUE-resolution [13] is an alternative to paramodulation that implicitly includes the equality axioms into the deduction. *RUE* denotes "resolution by unification and equality". In some way, it imposes a higher preference on the non-equality literals in the predicate substitution axioms such that they must be resolved first (see example) The authors have adapted the entire theory of unification resolution to equality, adding two new rules of inference, presented below.

- *RUE rule*: given the clauses $C = P(s_1, \ldots, s_n) \vee A$ and $D = \neg P(t_1, \ldots, t_n) \vee B$, the *RUE-resolvent* of C and D is $A\sigma \vee B\sigma \vee \mathcal{D}$ where $\mathcal{D}$ is *a disagreement set* of $P(s_1, \ldots, s_n)\sigma$ and $\neg P(t_1, \ldots, t_n)\sigma$.
- *NRF rule*: let C be the clause $t_1 \neq t_2 \vee A$. The *negative reflective function (NRF)-resolvent* of C is $A\sigma\mathcal{D}$ where $\mathcal{D}$ is *a disagreement set* of $t_1\sigma$ and $t_2\sigma$.

Both rules are based on the concept of *disagreement set.* Although very similar, the definition is not the same as the one stated in Section 4.2.3, which actually is a simplified version of the original definition given next.

Definition 31. (Disagreement set of a pair of terms).

- If s and t are non-identical terms, the set $\{s \neq t\}$ is the *origin disagreement set* of s and t.
- If s and t have the form $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$ then the set $\{s_1 \neq t_1, \ldots, s_n \neq t_n\}$ is the *top-most disagreement set* of s and t.
- If $\mathcal{D}$ is a disagreement set of s and t, then the set $\mathcal{D}'$ formed by replacing any member $s_i \neq t_i$ of $\mathcal{D}$ by the elements of the disagreement set of s_i and t_i is also a *disagreement set* of s and t.
- If s and t are identical terms, the empty set is the only disagreement set.

Example 24. The pair of terms $f(a, h(b, g(c)))$ and $f(b, h(c, g(d)))$ has the following disagreement sets:

- $\mathcal{D}_1 = \{f(a, h(b, g(c))) \neq f(b, h(c, g(d)))\}$ (origin disagreement set)
- $\mathcal{D}_2 = \{a \neq b, h(b, g(c)) \neq h(c, g(d))\}$ (top-most disagreement set)
- $\mathcal{D}_3 = \{a \neq b, b \neq c, g(c) \neq g(d)\}$
- $\mathcal{D}_4 = \{a \neq b, b \neq c, c \neq d\}$

Definition 32. (Disagreement set of complementary literals). A *disagreement set* of the literals $P(s_1, \ldots, s_n)$ and $\neg P(t_1, \ldots, t_n)$ is defined as the union

$$\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_i$$

where $\mathcal{D}_i$ is a disagreement set of the corresponding arguments s_i and t_i. The *top-most disagreement set* of $P(s_1, \ldots, s_n)$ and $\neg P(t_1, \ldots, t_n)$ is the set of pairs of corresponding arguments that are not identical.

The definition of disagreement set used for the disagreement linking and decomposition rules in the disconnection tableau calculus considers only the top-most disagreement set of two terms at the *rule level* and that is why they refer to *the* disagreement set as an unique set, whereas in RUE resolution the rules are stated according to *a* not unique disagreement set. In turn, all disagreement sets except for the origin with equality literals are considered. Apart from that difference, the disagreement linking rule and the decomposition rule are the tableau version of the RUE and CNF rules, respectively, for the general case of P being an n-ary predicate symbol different from equality. All the equality axioms are effectively contained in the RUE and CNF rules except for one, symmetry. For the latter, the following modification to the RUE rule must be applied, only for equality literals:

- *RUE rule for complimentary equality literals*: given the clauses $C = s_1 = s_2 \vee A$ and $D = t_1 \neq t_2 \vee B$, form two resolvents,

  $$\mathcal{D}_{1,1} \vee \mathcal{D}_{2,2} \vee A\sigma \vee B\sigma$$

  $$\mathcal{D}_{2,1} \vee \mathcal{D}_{1,2} \vee A\sigma \vee B\sigma$$

  where $\mathcal{D}_{i,j}$ is a disagreement set of $s_i\sigma, t_j\sigma$. The first one correspond to the normal RUE resolvent defined above and the second is obtained by interchanging the arguments of equality in *one* of the input equality literals, to effectively simulate symmetry.

The substitutions σ applied in the defined rules are expected to unify the involved literals as much as possible and thus reduce the number of inequalities in $\mathcal{D}$. This is implicitly done in the disagreement linking and decomposition rules for tableau when forcing $\sigma = \{x/t\}$ to be the disagreement substitution of a variable x and a term t.

4.3.2 RUE rules for Connection Tableau Calculus

We add two new rules to the Connection Tableau Calculus with the intermediate lemma extension, one corresponding to the RUE resolvent or disagreement linking rule and another corresponding to the NRF resolvent or the decomposition rule. The definition of disagreement set on which the new rules are based contain all the cases considered in the original definition above except for the origin disagreement set, together with Letz and Stenz's concept of disagreement substitution. Also, we modified the extension and reduction rules to consider symmetry for equality literals. We present here the resulting whole set of rules for the calculus and illustrate them with an example.

Given a set of clauses S and a marked tableau T for S, we define:

- *Tableau start*: T is a one-node tree with a root only. Choose an all-negative clause $\neg L_1 \vee \ldots \vee \neg L_n \in S$ and attach the nodes $\neg L_1, \ldots, \neg L_n$ to the root.
- *Lemma derivation*: T is a marked solved tableau. Add *skip*(T) to S, $S = S \cup \mathit{skip}(T)$ and start a new one-node tree with a root only.
- *Extension rule*: select a negative subgoal $\neg K$ and a clause $C = \underline{L_1} \vee \ldots \vee L_n \in S$ with conclusion literal L_1 such that L_1 and K are unifiable with mgu σ, and rename the variables to make them disjoint from the variables occurring in T. Attach the new successor nodes $\neg L_1, \ldots, \neg L_n$ to $\neg K$, mark L_1 with *closed* and apply σ to all the literals in T.
  - To include symmetry, if $\neg K = \neg(t_1 = t_2)$ is a negative equality literal, we select a clause C with conclusion literal $L_1 = (s_1 = s_2)$ such that $(t_1 = t_2)$ and either $(s_1 = s_2)$ or $(s_2 = s_1)$ are unifiable with mgu σ.
- *Reduction rule*: if a (positive or negative) subgoal K has an ancestor node L unifiable with $\overline{K}$ with mgu σ, then mark K with *closed* and apply σ to all the literals in T.

- To include symmetry, if $K = \neg(t_1 = t_2)$ (resp. $K = (t_1 = t_2)$) is a negative (resp. positive) equality literal with an ancestor $L = (s_1 = s_2)$ (resp. $L = \neg(s_1 = s_2)$) such that K and either $\neg(s_1 = s_2)$ or $\neg(s_2 = s_1)$ (resp. either $\neg(s_1 = s_2)$ or $(s_2 = s_1)$) are unifiable with mgu σ, then mark K with *closed* and apply σ to all the literals in T.

- *Skip*: Select a positive subgoal K and mark it with *skipped.*
- *Disagreement extension rule*: Select a negative subgoal $\neg K$.
  - If $\neg K = \neg P(t_1, \dots, t_n)$ is not an equality literal, choose a clause $C = \underline{L_1} \vee \dots \vee L_n \in S$ with conclusion literal $L_1 = P(s_1, \dots, s_n)$ such that the top-most disagreement set of K and L_1 is $\mathcal{D}$ and the disagreement substitution is σ. Rename the variables to make them disjoint from the variables occurring in T, attach new successor nodes with literals $L_1, \dots, L_n$ and inequalities in $\mathcal{D}$, mark L_1 with *closed* and apply σ to all the literals in T.
  - If $\neg K = \neg(t_1 = t_2)$ is an equality literal, choose a clause $C = \underline{L_1} \vee \dots \vee L_n \in S$ with conclusion literal $L_1 = s_1 = s_2$ such that the top-most disagreement set of t_i, s_j is $\mathcal{D}$ with disagreement substitution σ. Rename the variables to make them disjoint from the variables occurring in T, attach new successor nodes with literals $L_1, \dots, L_n$, the inequalities in $\mathcal{D}$ and an additional node $\neg(t_k = s_l)$ where $k \neq i, l \neq j$. Finally, mark L_1 with *closed* and apply σ to all the literals in T.
- *Decomposition rule:* select a negative subgoal $K = \neg(s = t)$ such that the top-most disagreement set of s and t is $\mathcal{D}$ with disagreement substitution σ. Attach new successor nodes with $s = t$ and the inequalities in $\mathcal{D}$, mark $s = t$ as *closed* and apply σ to all the literals in T.

We must make the following observations about the rules:

1. The disagreement extension rule and decomposition rule are only applicable if the input set contains equality. If this is not the case, those two rules are not considered and the set of rules corresponds to the Connection Tableau Calculus with the Intermediate Lemma Extension previously presented.
2. Although we choose always the top-most disagreement set *lower* or *deeper* level disagreement sets are considered in successive applications of the rules if needed. We take then the same approach as in lazy paramodulation in the disconnection calculus.
3. None of the equality axioms need to be added to the input set, as they are effectively simulated by the set of rules.
4. Neither ordering between the terms nor orientation of equations is considered.
5. The rules are not applied in a non-deterministic way but following an order. First we try to close a negative literal (equality or non-equality) by means of the normal extension steps. If this is not possible, we turn to the disagreement extension rule for non-equality literals or to the decomposition rule for equality literals. The last rule tried is then disagreement extension for equality literals. With respect to positive literals, reduction is tried before skip.

Next we provide two examples to illustrate the connection tableau calculus extended with the above rules.

Example 25. Consider the set of clauses with equality

$$S = \{x = a \vee x = b, \neg(g(x) = x), R(a, b), R(g(a), g(b)), \neg R(a, g(a)) \vee \neg R(b, g(b))\}$$

We show a closed connection tableau for S in Figure 4.6.

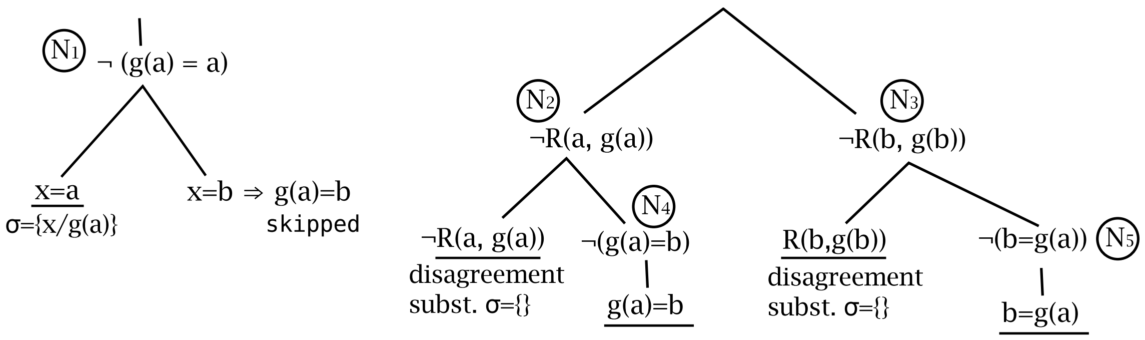


Figure 4.6: Closed connection tableau using disagreement extension rules

At node N_1 we apply the normal extension rule with clause $x = a \vee x = b$ and $\sigma = \{x/g(a)\}$, generating the lemma $g(a) = b$. We start a new tableau with clause $\neg R(a, g(a)) \vee \neg R(b, g(b))$. At node N_2 we perform a disagreement extension step with clause $R(a, b)$. The top-most disagreement set of the literals $\neg R(a, g(a))$ and $R(a, b)$ is $\mathcal{D} = \{b \neq g(a)\}$ after unification with $\sigma = \{\}$. $\mathcal{D}$ is attached to the right of the closure, at node N_4 and closed with an extension step using the previously generated lemma. We proceed exactly in the same way at node N_3, with a disagreement extension operation and a top-most disagreement set $\mathcal{D} = \{b \neq g(a)\}$ with $\sigma = \{\}$ between the literals $R(b, g(b)$ and $R(g(a), g(b))$ attached to node N_5. At N_5 an extension step that uses the symmetry property of equality is applied with the lemma $g(a) = b$ and the whole tableau closed.

This proof is found by our prover with the built-in equality mechanism in less than 0.01 seconds, whereas the prover without RUE rules needs 7.3 seconds to find a much longer proof for $S \cup Axioms$

Example 26. Consider the same set of clauses with only equality literals as in Example 23.

$$S = \{\neg(h(a) = c), d = h(e), h(b) = h(f), a = b, c = d, e = f\}$$

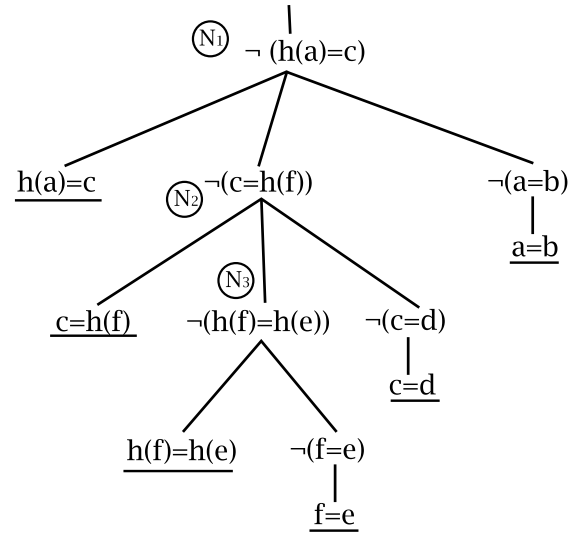


Figure 4.7: Closed connection tableau using disagreement extension and decomposition rules

Our only possible start clause is $\neg(h(a) = c)$. We start with a disagreement extension step at node N_1 using clause $h(b) = h(f)$. The top-most disagreement set of $h(a)$ and $h(b)$ is $\mathcal{D} = \{a \neq b\}$ and we need to add also the inequality $c \neq h(f)$. At node N_2 we apply again the disagreement extension rule with clause $d = h(e)$, adding to the right the disagreement set $\mathcal{D} = \{c \neq d\}$ and the inequality $h(f) \neq h(e)$ at node N_3. To expand the latter we apply a decomposition step since there is a top-most disagreement set for $h(f)$ and $h(e)$, $\mathcal{D} = \{f \neq e\}$.The rest of nodes are closed with normal extension steps using clauses in S, sometimes making use of the symmetry property for equality.

In this case, our prover without built-in equality mechanism finds a proof for $S \cup Axioms$ in approximately the same time as the prover with disagreement linking and decomposition rules (less than 0.01 seconds) but the proof found is still longer.

Proposition 7. *(Soundness of the Connection Tableau Calculus with the Intermediate Lemma Extension and RUE resolution rules for equality).*

The Connection Tableau Calculus with the Intermediate Lemma Extension and RUE resolution rules for equality is sound.

Given a set S that may contain equality literals but not the equality axioms, if a sequence of connection tableau $T_1, \ldots, T_n$ with T_n closed can be constructed from S by application of the rules of the calculus, then S is unsatisfiable.

Proof. (Soundness of the Connection Tableau Calculus with the Intermediate Lemma Extension and RUE resolution rules for equality).

To prove soundness for this calculus, we rely on soundness of the Connection Tableau Calculus with the Intermediate Lemma Extension, proved in Section 3.1.

Let S be a set of clauses with equality that does not contain the equality axioms. Suppose that a sequence of connection tableau $T_1, \ldots, T_n$ with T_n closed is constructed for S using the rules of the calculus. If no disagreement extension step, decomposition step or the symmetry case for the extension step have been applied in $T_1, \ldots, T_n$, then the sequence $T_1, \ldots, T_n$ could have been constructed with the connection tableau calculus with the intermediate lemma extension and by soundness of this calculus we can conclude that S is unsatisfiable. Suppose then that some of those rules have been applied in the sequence of tableaux. We show that every step involving disagreement extension, decomposition or symmetric extension can be replaced by a succession of steps using only extension and reduction steps and the equality axioms, and therefore could have been performed in the context of the calculus without built-in equality rules.

- *Symmetry case for the extension/reduction rule.* we have performed an extension step with subgoal $K = \neg(t_1 = t_2)$, a selected clause $C = (s_1 = s_2) \vee L_2 \ldots \vee L_n$ and an unifier σ such that $(t_2 = t_1)\sigma = (s_1 = s_2)$. This extension step can be replaced by two extension steps, the first one involving the equality axiom $\neg(\underline{x = y}) \vee y = x$, the subgoal K and unifier $\sigma_1 = \{x/t_1, y/t2\}$ and the second one involving $C = \overline{(s_1 = s_2)} \vee L_2 \ldots \vee L_n$, next subgoal $\neg(t_2 = t_1)$ and unifier σ. Figure 4.8 shows both steps in the right tableau. For the reduction is analogous but closing the subgoal $\neg(t_2 = t_1)$ after performing the extension step with the symmetry axiom.

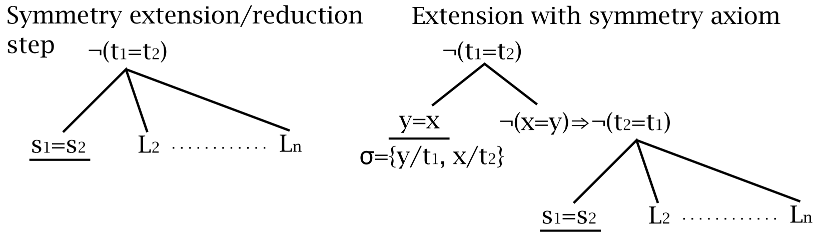


Figure 4.8: Transformation of the symmetry case for extension/reduction step

- *Disagreement extension step with non-equality subgoal.* we have performed a disagreement extension step at a subgoal $K = \neg P(t_1, \ldots, t_n)$, with a clause $C = P(s_1, \ldots, s_n) \vee L_2 \ldots \vee L_n$ such that the top-most disagreement set of K and L_1 is $\mathcal{D}$ and the disagreement substitution is σ. This step can be replaced again by two extension steps, the first one using the substitution axiom (the second version given at the beginning of the chapter) for P, $P(x_1, \ldots, x_n) \vee \neg P(y_1, \ldots, y_n) \vee \neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n)$ at subgoal $K = \neg P(t_1, \ldots, t_n)$ with substitution $\sigma_1 = \{x_1/t_1, \ldots, x_n/t_n\}$ and the second using the clause $C = P(s_1, \ldots, s_n) \vee L_2 \ldots \vee L_n$ at subgoal $\neg P(y_1, \ldots, y_n)$ with $\sigma_2 = \{y_1/s_1, \ldots, y_n/s_n\}$. As for the symmetry case, we show the equivalence of both versions in Figure 4.9.

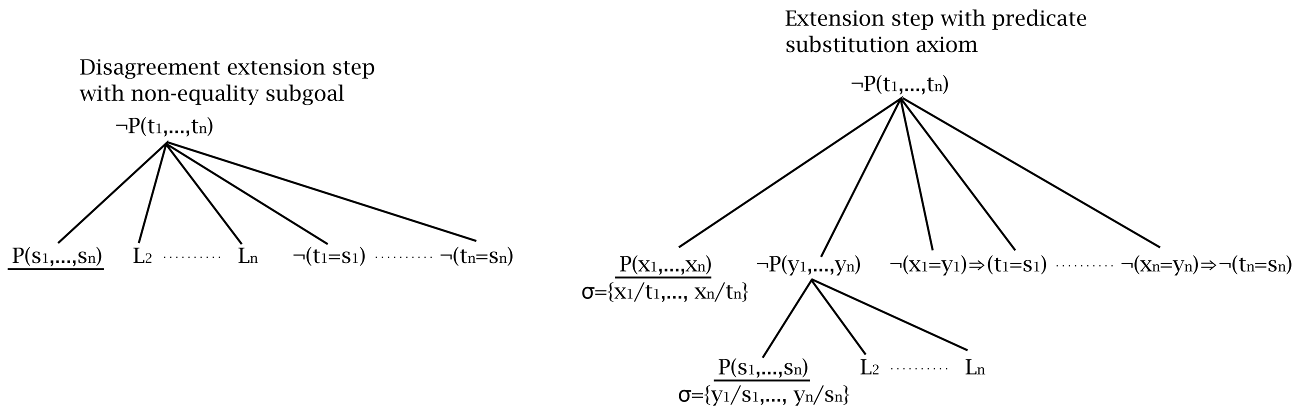
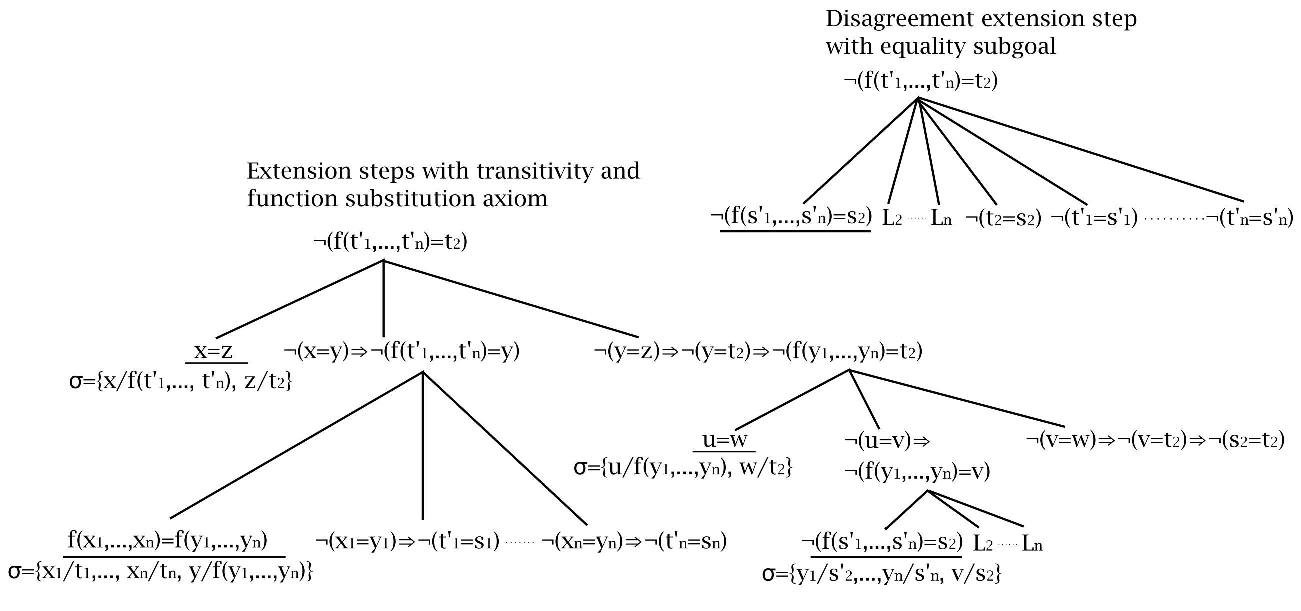


Figure 4.9: Transformation of the disagreement extension step with non-equality subgoal

- *Disagreement extension step with equality subgoal.* we have performed a disagreement extension step at a subgoal $K = \neg(t_1 = t_2)$, with a clause $C = (s_1 = s_2) \vee L_2 \ldots \vee L_n$, and suppose that the top-most disagreement set of t_1, s_1 is $\mathcal{D}$ with disagreement substitution σ. The other cases for different values of i, j are analogous to this, as we can always use the reflexivity axiom to invert the order of equations.

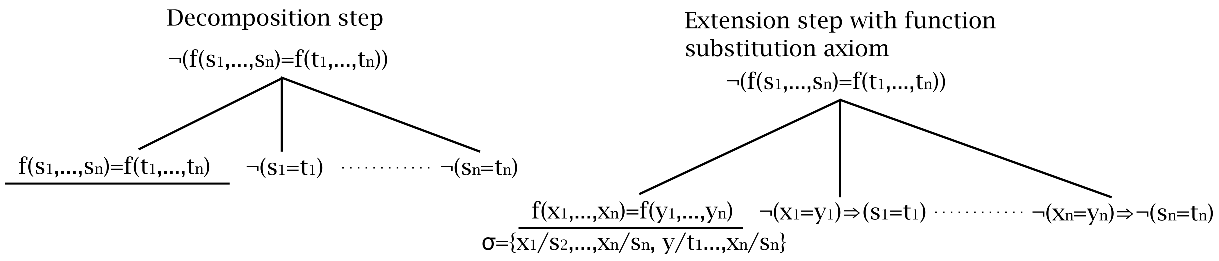


Disagreement extension step with equality subgoal
¬(f(t'1,...,t'n)=t2)
¬(f(s'1,...,s'n)=s2)
L2
Ln
¬(t2=s2)
¬(t'1=s'1)
¬(t'n=s'n)
Extension steps with transitivity and function substitution axiom
¬(f(t'1,...,t'n)=t2)
x=z
σ={x/f(t'1,..., t'n), z/t2}
¬(x=y)⇒¬(f(t'1,...,t'n)=y)
¬(y=z)⇒¬(y=t2)⇒¬(f(y1,...,yn)=t2)
f(x1,...,xn)=f(y1,...,yn)
σ={x1/t1,..., xn/tn, y/f(y1,...,yn)}
¬(x1=y1)⇒¬(t'1=s1)
¬(xn=yn)⇒¬(t'n=sn)
u=w
σ={u/f(y1,...,yn), w/t2}
¬(u=v)⇒
¬(f(y1,...,yn)=v)
¬(v=w)⇒¬(v=t2)⇒¬(s2=t2)
¬(f(s'1,...,s'n)=s2)
σ={y1/s'2,...,yn/s'n, v/s2}
L2
Ln

Figure 4.10: Transformation of the disagreement extension step with equality subgoal

We have that t_1 and s_1 are of the form $f(t'_1, \ldots, t'_n)$ and $f(s'_1, \ldots, s'_n)$, otherwise they would not have a top-most disagreement set. In this case, the step can be replaced by 4 extension steps. The first one uses the transitivity axiom, $x = z \vee \neg(x = y) \vee \neg(y = z)$ at subgoal $K = \neg(f(t'_1, \ldots, t'_n) = t_2)$ with substitution $\sigma_1 = \{x/t_1, z/t_2\}$. The second one uses the function substitution axiom for f (the second version of it), $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \vee \neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n)$ at subgoal $\neg(f(t'_1, \ldots, t'_n) = y)$ with substitution $\sigma_2 = \{x_1/t'_1, \ldots, x_n/t'_n, y/f(y_1, \ldots, y_n)\}$. The third steps uses again the transitivity axiom, $u = w \vee \neg(u = v) \vee \neg(v = w)$ at subgoal $\neg(f(y_1, \ldots, y_n) = t_2)$ with $\sigma_3 = \{u/f(y_1, \ldots, y_n), w/t_2\}$. Finally, the fourth step uses the clause $C = (f(s'_1, \ldots, s'_n) = s_2) \vee L_2 \ldots \vee L_n$ at subgoal $\neg(f(y_1, \ldots, y_n) = v)$ with $\sigma_4 = \{y_1/s'_1, \vee, y_n/s'_n, v/s_2\}$ As for the symmetry case, in Figure 4.10 we show the equivalence between the two versions of the calculus.

- *Decomposition step.* we have performed a decomposition step at subgoal $K = \neg(s = t)$ such that the top-most disagreement set of s and t is $\mathcal{D}$ with disagreement substitution σ. Then s and t are of the form $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$, otherwise they would not have a top-most disagreement set. We replace this step with one extension step using the equality substitution axiom for f (the second version of it), $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \vee \neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n)$ with $\sigma = \{x_1/s_1, \ldots, x_n/s_n, y_1/t_1, \ldots, y_n/t_n\}$, as illustrated in Figure 4.11.



Decomposition step
¬(f(s1,...,sn)=f(t1,...,tn))
f(s1,...,sn)=f(t1,...,tn)
¬(s1=t1)
¬(sn=tn)
Extension step with function substitution axiom
¬(f(s1,...,sn)=f(t1,...,tn))
f(x1,...,xn)=f(y1,...,yn)
¬(x1=y1)⇒(s1=t1)
¬(xn=yn)⇒¬(sn=tn)
σ={x1/s2,...,xn/sn, y/t1...,xn/sn}

Figure 4.11: Transformation of the decomposition step

The order of the subgoals changes from one proof to another but the calculus is strong independent of the selection function used, as mentioned in Section 2.2.1.

After having applied all those transformation to the steps of the proof $T_1, \ldots, T_n$, we have a sequence of tableau with T_n closed constructed with the rules of the connection tableau calculus with the intermediate lemma extension and hence, we conclude that S is unsatisfiable. □

Proposition 8. *(Completeness of the Connection Tableau Calculus with the Intermediate Lemma Extension and RUE resolution rules for equality).*

The Connection Tableau Calculus with the Intermediate Lemma Extension and RUE resolution rules for equality is complete.

Given a set S that may contain equality literals but not the equality axioms, if S is unsatisfiable, then there exists a sequence of connection tableau $T_1, \ldots, T_n$ constructed using the rules of the calculus and such that T_n is closed.

Proof. (Completeness of the Connection Tableau Calculus with the Intermediate Lemma Extension and RUE resolution rules for equality).

As we did for soundness, to prove completeness for this calculus, we rely on completeness for the Connection Tableau Calculus with the Intermediate Lemma Extension, proved in Section 3.1.

Let S be a set of clauses with equality that does not contain the equality axioms. Suppose that S is unsatisfiable. Then, by completeness of the connection tableau calculus with the intermediate lemma extension, there exists a sequence of connection tableau $T_1, \ldots, T_n$ with T_n closed constructed for $S \cup \{Axioms\}$ using the rules of the calculus. In a similar way as we did for soundness, we show that this sequence can be transformed into another that do not uses any equality axioms but in which disagreement extension and decomposition steps or the symmetry version for the extension rule may have been applied. In order to do that, we need to transform only the extension steps that involve any of the 5 equality axioms contained in *Axioms*.

- *Reflexivity:* we have performed an extension step at subgoal $\neg(t = t)$ using clause $x = x$. In this case, we replace the step using the decomposition rule. The disagreement set of t and t is $\mathcal{D} = \{\}$ with $\sigma = \{\}$, as shown in Figure 4.12.

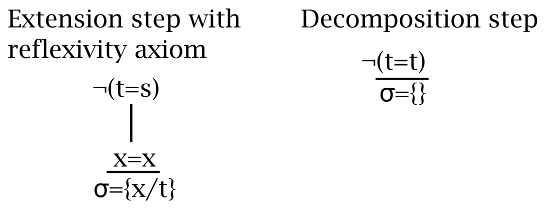


Figure 4.12: Transformation of extension step with reflexivity axiom

- *Symmetry:* we have performed an extension step at subgoal $\neg(t = s)$ with clause $x = y \vee \neg(y = x)$. We replace this step with an extension step applying the symmetry case with the nodes under $\neg(s = t)$, as we can see in Figure 4.13. Notice that these nodes must exists, since no reduction step can be applied at a negative subgoal and because the tableau is solved, an extension step must have been performed.

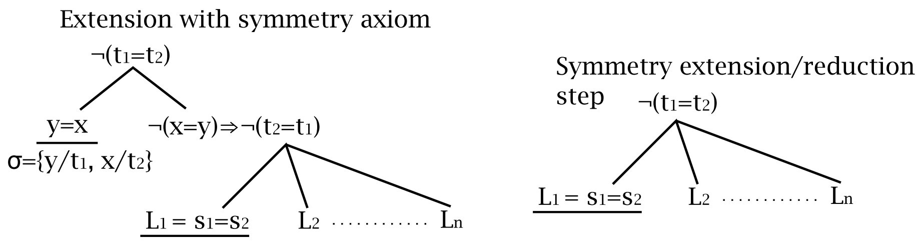


Figure 4.13: Transformation of extension step with symmetry axiom

- *Transitivity:* we have performed an extension step at subgoal $\neg(t = s)$ with clause $x = z \vee \neg(x = y) \vee \neg(y = z)$. We can substitute this step with two disagreement extension steps. The first one uses $L_1 = t_1 = s_1$ under $\neg(t = y)$, at the subgoal $\neg(t = s)$ with an empty disagreement set between t and t_1 (since they are unifiable) and the same substitution $\sigma_1 \cup \{y/t_2\}$ used at the extension step. The second step is performed at subgoal $\neg(s_1 = s)$ using $K_1 = (s_2 = t_2)$, with an empty disagreement set between s_1 and s_2 as they are unifiable, and disagreement substitution σ_2, used at the extension step. As for the case of symmetry, notice that L_1 and K_1 must exist, with the appropriate substitutions. This is so because the tableau in which the transitivity axiom is used for an extension step is closed with the rules of the connection tableau calculus with the intermediate lemma extension, and since $\neg(t = y)$ and $\neg(s_1 = s)$ are negative subgoals, suitable clauses $L_1 \vee C$ and $K_1 \vee D$ must have been used for extension steps at them. This is clarified in Figure 4.14.

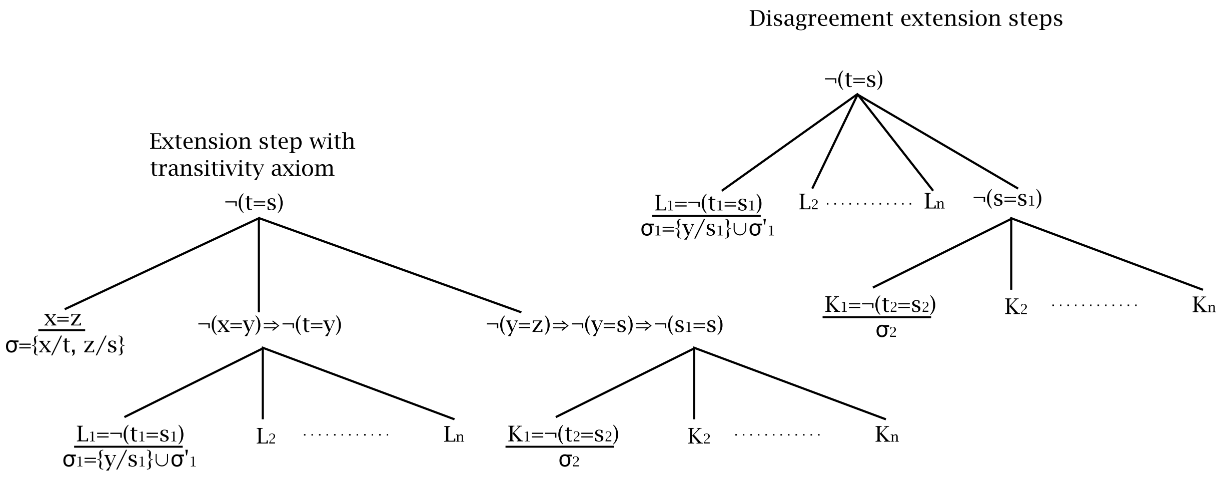


Figure 4.14: Transformation of extension step with transitivity axiom

- *Function substitution:* we have performed an extension step at subgoal $\neg(f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n))$ using $f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n) \vee \neg(x_1 = y_1)\ldots \vee \neg(x_n = y_n)$. We replace this step by a decomposition step with disagreement set $\mathcal{D} = \{\neg(t_1 = s_1),\ldots,\neg(t_n = s_n)\}$, as it can be observed in Figure 4.15.

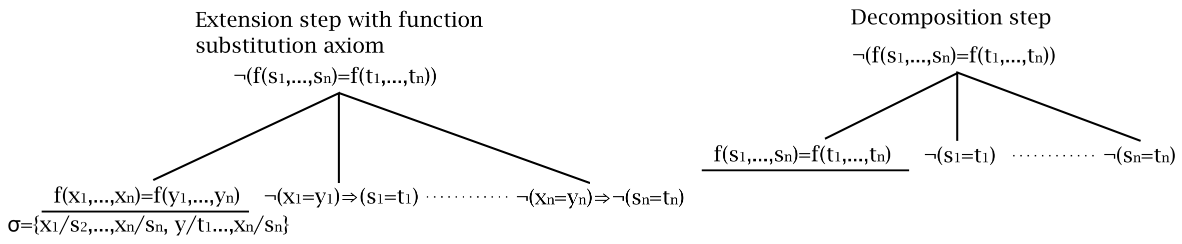

Extension step with function substitution axiom
¬(f(s1,...,sn)=f(t1,...,tn))
f(x1,...,xn)=f(y1,...,yn)
σ={x1/s2,...,xn/sn, y/t1...,xn/sn}
¬(x1=y1)⇒(s1=t1)
¬(xn=yn)⇒¬(sn=tn)
Decomposition step
¬(f(s1,...,sn)=f(t1,...,tn))
f(s1,...,sn)=f(t1,...,tn)
¬(s1=t1)
¬(sn=tn)

Figure 4.15: Transformation of the extension step with a predicate substitution axiom

- *Predicate substitution:* we have performed an extension step at subgoal $\neg P(t_1,\ldots,t_n)$ using $P(y_1,\ldots,y_n) \vee \neg P(x_1,\ldots,x_n) \vee \neg(x_1 = y_1)\ldots \vee \neg(x_n = y_n)$. We replace this step with a disagreement extension step involving the clause $P(s_1,\ldots,s_n) \vee L_2 \vee \ldots \vee L_n$ used for the extension step under $\neg P(x_1,\ldots,x_n)$ with disagreement set $\mathcal{D} = \{\neg(t_1 = s_1),\ldots,\neg(t_n = s_n)\}$.

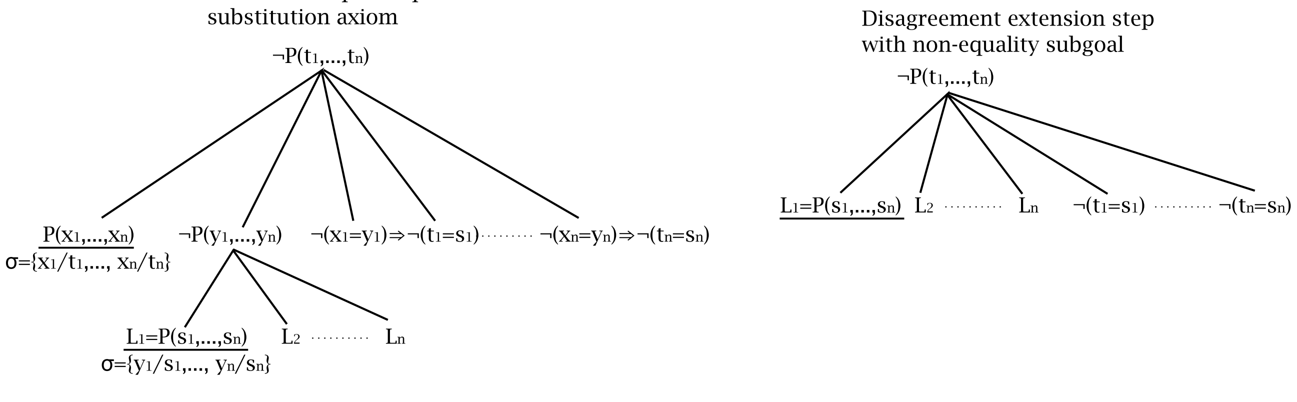


Figure 4.16: Transformation of the extension step with a predicate substitution axiom

Again, this clause must exist for an analogous reasoning as for transitivity, to respect the rules of the connection tableau calculus with the intermediate lemma extension that have been used to close the tableau. We show this equivalence in Figure 4.16.

As for soundness, the order of the subgoals changes from one proof to another but the calculus is strong independent of the selection function used, as mentioned in Section 2.2.1. Once we have applied all the transformation to the sequence $T_1, \ldots, T_n$ of tableau, the resulting sequence $T'_1, \ldots, T'_n$ is a sequence of connection tableau with T'_n closed for S, since it does not use any clause from $Axioms$, constructed using the rules of the connection tableau calculus with the intermediate lemma extension and RUE rules for equality handling. Hence we can conclude that the calculus is complete.

$\square$

Finally and for the sake of completeness, we formalise and summarise the rules for this variant of the Connection Tableau Calculus with the Intermediate Lemma Extension in Table 4.3.2.

$$\textit{Start } (St)\ \frac{\{root\}, S, \{\}, \{\}}{C, S, \{root\}, \{\}}$$

where $C \in S$ is an all-negative clause, $C = \{\neg L_1, \dots, \neg L_n\}$

$$\textit{Extension } (Ext)\ \frac{C \cup \{\neg K\}, S, Branch, Skip}{(C_2 \backslash \{L_1\})\sigma, S, (Branch)\sigma \cup \{\neg K\sigma\}, (Skip)\sigma \ \ || \ \ C\sigma, S, (Branch)\sigma, (Skip)\sigma}$$

C_2 is a renamed instance of $C_1 = \{\underline{L_1}, \dots, L_n\} \in S$ and either
K is not an equality literal and $\sigma(K) = \sigma(L_1)$ or
$K = \neg(t_1 = t_2)$, $L_1 = (s_1 = s_2)$ and $\sigma(K) = \sigma(s_1 = s_2)$ or $\sigma(K) = \sigma(s_2 = s_1)$

$$\textit{Reduction } (Red)\ \frac{C \cup \{K\}, S, Branch \cup \{L\}, Skip}{C\sigma, S, (Branch)\sigma \cup \{L\sigma\}, (Skip)\sigma}$$

with either K not an equality literal and $\sigma(K) = \sigma(\overline{L})$ or
$K = \neg(t_1 = t_2)$, $L = (s_1 = s_2)$ and $\sigma(\overline{K}) = \sigma(s_1 = s_2)$ or $\sigma(\overline{K}) = \sigma(s_2 = s_1)$ or
$K = (t_1 = t_2)$, $L = \neg(s_1 = s_2)$ and $\sigma(K) = \sigma(s_1 = s_2)$ or $\sigma(K) = \sigma(s_2 = s_1)$

$$\textit{Skip } (Sk)\ \frac{C \cup \{K\}, S, Branch, Skip}{C, S, Branch, Skip \cup \{K\}}\ \text{with } K \text{ a positive literal}$$

$$\textit{Lemma derivation } (Lem)\ \frac{\{\}, S, Branch, Skip}{\{root\}, S \cup Skip, \{\}, \{\}}\ \text{with } Skip \neq \emptyset$$

$$\textit{Disagreement extension } (Dis)$$
$$\frac{C \cup \{\neg K\}, S, Branch, Skip}{(C_2 \backslash \{L_1\})\sigma \cup \mathcal{D}\sigma \cup I\sigma, S, (Branch)\sigma \cup \{\neg K\sigma\}, (Skip)\sigma \ \ || \ \ C\sigma, S, (Branch)\sigma, (Skip)\sigma}$$

C_2 is a renamed instance of $C_1 = \{\underline{L_1}, \dots, L_n\} \in S$ and either
K is not an equality literal, the top-most disagreement set of K and L_1 is $\mathcal{D}$
with disagreement substitution σ and $I = \{\}$ or
$K = \neg(t_1 = t_2)$, $L_1 = (s_1 = s_2)$, the origin disagreement set of t_i, s_j is $\mathcal{D}$
with disagreement substitution σ and $I = \{\neg(t_k = s_l)\}, k \neq i, l \neq j$

$$\textit{Decomposition } (Dec)\ \frac{C \cup \{\neg(s = t)\}, S, Branch, Skip}{C\sigma \cup \mathcal{D}\sigma, S, (Branch)\sigma \cup \{\neg(s = t)\sigma\}, (Skip)\sigma}$$

and $\mathcal{D}$ is the top-most disagreement set of s and t with disagreement substitution σ

Table 4.1: RUE Rules for the Connection Tableau Calculus with the Intermediate Lemma Extension

Chapter 5

Architecture of the prover (I): Basics

In this chapter we explain how we have implemented the basic version of the prover based on the Connection Tableau Calculus with the Intermediate Lemma Extension, as defined in Section 3.1. As we advanced in the Introduction chapter, the language chosen to implement the prover is Prolog. We have approached the design of the different versions of our prover as a Prolog Technology Theorem Prover. The particularities of such systems in general were introduced in Section 2.3. In this chapter we detail some decisions that we have made regarding the implementation of the basic calculus and other features and simple shortcuts that are common to all versions of the prover. We present some of the code that is part of our implementation but for the sake of brevity and clarity, a simplified version omitting irrelevant fragments for the aspect being discussed is reproduced in each section.

We provide a small summary of performance results at the end of the chapter, in Section 5.5 and, just as a matter of interest, we discuss briefly the basis for another possible implementation in Section 5.6.

5.1 Preprocessing the input set

Before the proof search starts, the clauses in the input set S are written into the Prolog database. This allows us to use the indexing mechanism of Prolog and also the process of adding a new lemma to S is replaced by asserting a new clause in the database. Atoms are represented by Prolog atoms and the negation is represented by `-`. The substitution σ found during the proof search is stored implicitly by Prolog. For every clause $C \in S$, we add a fact of the form `cla(H,B,L)` as follows:

- If C is a start clause (that is, it does not contain any conclusion or lemma literals), we add `cla([#],B,L)`, where `B` is a list containing the literals in C.
- If C is not a start clause, we select first a conclusion literal in C. In the current version, we pick the first positive literal that appears in C in order. If we have chosen `Lit`, the fact `cla([Lit],B,L)` is inserted in the database, where `B` is the list of literals in $C \backslash Lit$. A fact of this form is also inserted when a new lemma is generated.

The parameter `L` refers to the *level of the clause*, used by the iterative deepening strategy, and it is detailed in Section 5.3 later in this chapter. Initially, it is 0 for ground clauses and 1 for not ground clauses. Before inserting the corresponding fact for a clause C, we check if C is a tautology (i.e., a clause that contains complimentary literal) and in that case C is not inserted. Obviously it is not necessary to check that when asserting a new lemma since all literals in the lemma are positive.

Example 27. The set of clauses $S = \{\neg P(a), \neg Q(x,y) \vee \underline{R(b)}, \neg R(b), \underline{P(x)} \vee Q(x,y) \vee R(c)\}$ is stored as the following set of facts in the Prolog's database:

```
cla([#],[p(a)],0).
cla([r(b)],[q(X,Y)],1).
cla([#],[r(b)],0).
cla([p(X)],[q(X,Y),r(c)],1).
```

In contrast to for example leanCoP, that inserts a fact for each clause and for each literal in order to implement the normal Connection Method without lemmas, we just need one element in the database to represent each clause, thanks to the skip rule.

Using this representation, we use the first argument in `cla/3` to index and search the different clauses. When we try to apply extension to solve a subgoal $\neg Lit$ or `-Lit`, we can use Lit as index with `cla([Lit],B,L)`, that succeeds if there is a clause with a conclusion literal unifiable with Lit. This technique makes use of the Prolog's built-in indexing mechanism on the first argument and suitable clauses are found quicker. It integrates one of the main features of prolog technology theorem provers to improve performance.

It is important to be careful with the kind of unification employed to do that. We explained in Section 2.3.1 that by default most Prolog implementations omit occurs-check in their unification algorithm. In our case, not having sound unification can lead to incorrect deduction and to generation of cyclic lemmas (containing infinite or cyclic terms of the form $f(f(f(\ldots)))$, represented in Prolog by `f(**)`). One possibility that works in any Prolog implementation is to make use of the ISO predicate `unify_with_occurs_check(+Term1,+Term2)`, that behaves as `=/2` but using sound unification. The code would be like this, for subgoal `Lit`.

```
cla([Lit1],B,H), unify_with_occurs_check(Lit1,Lit)
```

However, this has the major disadvantage that we loose the Prolog's indexing mechanism, since we have to look clauses in turn before we find a right one. Instead of doing that, we use the runtime Prolog flag available in ECLiPSe [2] and SWI-Prolog [3], that for the latter is activated with `set_prolog_flag(occurs_check,true).` This has the advantage that sound unification is always used, including in built-in predicates such as `member/3` or `subsumes/2`.

5.1.1 Equality axioms

In the case of problems that contain equality, in the simple version of the prover in which no built-in mechanism are implemented to deal with equality literals, the equality axioms in clausal form stated in Chapter 4 have to be added to the input set before the proof start. They are asserted into the Prolog's database as normal clauses. We add the 3 basic equality axioms, reflexivity, symmetry and transitivity:

```
cla([#],[X=X]).
cla([Y=X],[-(X=Y)]).
cla([X=Z],[-(X=Y),-(Y=Z)]).
```

And finally for each predicate symbol P and function symbol f of arity n we add one axiom of the form:

```
cla([p(Y1,...,Yn)],[-p(X1,...,Xn),-(X1=Y1),...,-(Xn=Yn)]).
cla([f(Y1,...,Yn)=f(X1,...,Xn)][-(X1=Y1),...,-(Xn=Yn)]).
```

5.1.2 Clausal transformation

To finish this section about the preprocessing of the input set, we discuss the case in which the input is not in clausal form, but a set of arbitrary first order sentences. In order to transform this input into a set of clauses suitable for our prover, we apply the simple transformation algorithm below to each formula in the set S, similarly to other provers as leanCoP 1.0 [33].

1. All variables in the given formula are renamed, so that variable names in the given formula that do not refer to the same variable are different.
2. The formula is transformed into a Skolemized negation normal form, Skolemized NNF.

   A first-order logic formula is said to be in Skolemized negation normal form if it is of the form:

$$\forall x_1, \ldots, x_n \;\; F$$

   where F is a quantifier-free formula in negation normal form, NNF. A formula is in negation normal form if it contains exclusively the logical connectives $\neg, \wedge, \vee$ and negation occurs only immediately above atoms. For example, $\neg P(a) \vee (Q(x) \wedge \neg R(b, y))$ is in negation normal form. Once we have a formula in Skolemized NNF we consider all variables in F to be universally quantified so we get rid of $\forall x_1, \ldots, x_n$ and those variables become *free variables.* For Skolemization, we need to define the list of free variables of a formula F as follows:

$$fv(F) = \begin{cases} fv(F_1) \cup \{x\} & \text{if } F = \forall x\ F_1, \text{ or } F = \neg\exists x\ F_1 \\ fv(F_1) \cup fv(F_2) & \text{if } F = F_1 \vee F_2, F = F_1 \rightarrow F_2, F = \neg(F_1 \wedge F_2), \\ & F = F_1 \wedge F_2, F = \neg(F_1 \rightarrow F_2), F = \neg(F_1 \vee F_2), \\ & F = F_1 \leftrightarrow F_2 \text{ or } F = \neg(F_1 \leftrightarrow F_2) \\ fv(F_1) & \text{if } F = \neg\neg F_1, F = \neg\forall x\ F_1, \text{ or } F = \exists x\ F_1 \\ \emptyset & \text{if } F \text{ is a literal} \end{cases}$$

   The transformation to Skolemized NNF is then performed through the following rewrite rules, applying them recursively to formulas F, F_1, F_2.

   - $nnf(\neg\neg F) = nnf(F)$
   - $nnf(\neg(\forall x\ F)) = nnf(\exists x; \neg F)$
   - $nnf(\neg(\exists x\ F)) = nnf(\forall x; \neg F)$
   - $nnf(\neg(F_1 \vee F_2)) = nnf(\neg F_1 \wedge \neg F_2)$
   - $nnf(\neg(F_1 \wedge F_2)) = nnf(\neg F_1 \vee \neg F_2)$
   - $nnf(F_1 \rightarrow F_2) = nnf(\neg F_1 \vee F_2)$
   - $nnf(\neg(F_1 \rightarrow F_2)) = nnf(F_1 \wedge \neg F_2)$
   - $nnf(F_1 \leftrightarrow F_2) = nnf((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$
   - $nnf(\neg(F_1 \leftrightarrow F_2)) = nnf((F_1 \wedge \neg F_2) \vee (\neg F_1 \wedge F_2))$

- $nnf(\forall x\ F) = nnf(F)$, ($x$ is a free variable of F).
- $nnf(\exists x\ F) = nnf(F')$ where all F' is obtained from F by replacing all occurrences of x by a Skolem new constant or Skolem function depending on $fv(F)$. For example, given the formula $F = \forall x\ (\exists y\ R(x, y))$, we apply the following succession of rewriting steps:
  $nnf(\forall x\ (\exists y\ R(x, y), \{\}) = nnf(\exists y\ R(x, y), \{x\}) = nnf(R(x, f(x)), \{x\})$, where f is a new function symbol.
- $nnf(F_1 \vee F_2) = nnf(nnf(F_1) \vee nnf(F_2))$
- $nnf(F_1 \wedge F_2) = nnf(nnf(F_1) \wedge nnf(F_2))$
- $nnf(F) = F$ if F is a literal.

Double negations are eliminated, negation is pushed inside using De Morgan's laws and implications and equivalences are replaced by their definitions. Existential quantifiers are removed by Skolemization and variables universally quantified are kept as free variables (universal quantifiers become implicit).

In the actual implementation done for this transformation procedure we have included a small optimisation suggested in [5] and implemented in leanT^AP [6] (a complete and sound theorem prover for classical first-order logic based on free-variable semantic tableaux implemented in Prolog with source code of merely 360 bytes in the smallest version) and all versions of leanCoP [33, 31]. The optimisation is based in changing the sequence of disjunctively connected formulas. During transformation, we store the number of *disjunctive paths* in a formula (i.e., the number of branches that a fully expanded tableau for it would have, see Section 2.1) and this is used when handling disjunction and conjunction in the transformation. In both cases, the less branching formula is put to the left and in that way, the number of choice points during the proof search is reduced. The number of disjunctive paths can be defined as follows for a formula F:

$$paths(F) = \begin{cases} paths(F_1) + paths(F_2) & \text{if } F = F_1 \vee F_2, F = F_1 \rightarrow F_2, \text{ or } F = \neg(F_1 \wedge F_2) \\ paths(F_1)paths(F_2) & \text{if } F = F_1 \wedge F_2, F = \neg(F_1 \rightarrow F_2), \text{ or } F = \neg(F_1 \vee F_2) \\ paths(F_1 \wedge F_2) + paths(\neg F_1 \wedge \neg F_2) & \text{if } F = F_1 \leftrightarrow F_2 \\ paths(F_1 \wedge \neg F_2) + paths(\neg F_1 \wedge F_2) & \text{if } F = \neg(F_1 \leftrightarrow F_2) \\ paths(F_1) & \text{if } F = \neg\neg F_1, F = \forall x\ F_1, F = \neg\forall x\ F_1, \\ & \ F = \exists x\ F_1 \text{ or } F = \neg\exists x\ F_1 \\ 1 & \text{if } F \text{ is a literal} \end{cases}$$

With this consideration, we modify the following two cases for the previous procedure:

- $nnf(F_1 \vee F_2) = \begin{cases} nnf(nnf(F_1) \vee nnf(F_2)) & \text{if } paths(F_1) > paths(F_2) \\ nnf(nnf(F_2) \vee nnf(F_1)) & \text{otherwise} \end{cases}$
- $nnf(F_1 \wedge F_2) = \begin{cases} nnf(nnf(F_1) \wedge nnf(F_2)) & \text{if } paths(F_1) > paths(F_2) \\ nnf(nnf(F_2) \wedge nnf(F_1)) & \text{otherwise} \end{cases}$

3. The NNF formula is transformed into conjunctive normal form, CNF, by applying the distributive laws of $\vee$ and $\wedge$. Recall that a formula is in CNF if it is a conjunction of clauses. A formula in NNF is transformed into CNF recursively as follows (F, F_1, F_2, F_3 are formulas in NNF):

   - $cnf((F_1 \wedge F_2) \vee F_3) = cnf(F_1 \vee F_3) \wedge cnf(F_2 \vee F_3)$
   - $cnf(F_1 \vee (F_2 \wedge F_3)) = cnf(F_1 \vee F_2) \wedge cnf(F_1 \vee F_3)$

- $cnf(F_1 \wedge F_2) = cnf(F_1) \wedge cnf(F_2)$
- $cnf(F_1 \vee F_2) = \begin{cases} cnf(cnf(F_1) \vee cnf(F_2)) & \text{if } \wedge \text{ occurs in } cnf(F_1) \text{ or } cnf(F_2) \\ cnf(F_1) \vee cnf(F_2) & \text{otherwise} \end{cases}$
- $cnf(F) = F$ if F is a literal.

4. Finally, the CNF formula $F = C_1 \wedge \ldots \wedge C_n$ is written as a set of clauses $S = \{C_1, \ldots, C_n\}$ where each clause $C_i = L_1 \vee \ldots \vee L_n$ is written as a set of literals $\{L_1, \ldots, L_n\}$

5.2 Core prover and intermediate lemma generation

The implementation of the basic connection tableau calculus with the intermediate lemma extension consists of one predicate, `prove/9`, head of two main clauses plus one fact (base case). The two clauses represent the two cases that we have for subgoals, undertaking either a positive or negative literal. Open branches are selected in a depth-first way (the search method employed is discussed in detail in Section 5.3)

When attempting to solve a negative subgoal $\neg L$, the unique choice is to apply a extension rule searching for a suitable clause, with a matching conclusion literal, in the current database. Once this clause $\{H\} \cup B$ has been found, literals in B have to be tackled with $\neg L$ added to the current branch, as all literals in B would be nodes below L. When all subgoals in B have been solved, $\neg L$ is completely solved and we continue with the resting subgoals in C after $\neg L$. Finally, lemma literals produced as a result of solving first the literals in B and second the literals in C, $Skip_1$ and $Skip_2$ respectively, are combined in a single set $Skip$. This can be implemented in Prolog code as follows:

```
prove([-Lit|Clause],Branch,...,Skip,...) :-
            ...
            cla([Lit],B,L),
            prove(B,[-Lit|Branch],...,Skip1,...),
            ...
      prove(Clause,Branch,...,Skip2,...),
      combine_lemma_literals(Skip1,Skip2,Skip),
      ...
```

The parameters `Clause, Branch, Skip` corresponds $C, Branch$ and $Skip$ in the tuple $C, S, Branch, Skip$ used to formalise the rules of the calculus in Table 3.1. The set S is contained in the Prolog's database and therefore not needed as an extra argument of the calculus (although this could have been a possible implementation that does not make use of Prolog's indexing mechanism).

Combining the lemma literals in $Skip_1$ and $Skip_2$ is not done as as simple set union operation or append operation for lists, but instead we apply a kind of *skip-factoring* operation (not to be confused with the shortcut *Factoring* for tableaux discussed in Section 2.2.2). It is similar to the rule skip-factoring of the SOL Tableau Calculus (see Section 3.2.1) and because is just used for skipped literals we have chosen this name. What it is actually implemented is the definition of *safe factor of a clause.*

Definition 33. (Factor of a clause, safe factor of a clause). Given a clause $C = L_1 \vee L_n \vee D$, where $L_1, \ldots, L_n$ are literals of the same sign unifiable with mgu σ and D is a clause, F is a *factor* of C if $F = (L \vee D)\sigma$, where $L = L_i\sigma$ for each $i, 1 \leq i \leq n$. If additionally, F subsumes C, F is a *safe factor* of C.

If C subsumes D, then $C \models D$ and if C is a factor of D, then $D \models C$. This implies that both clauses C and D are equivalent and D can be substituted by C. If C is a non-safe factor of D (i.e., C does not subsume D), then D can not be replaced by C.

Example 28. The clause $C = P(a, y) \vee P(x, y) \vee P(a, b) \vee Q(y, c)$ factors to $P(a, b) \vee Q(b, c)$ non-safely and to $P(a, y) \vee P(a, b) \vee Q(y, c)$ safely. Notice that $P(a, b) \vee Q(b, c)$ does not subsume C so it can not replace it, as solutions with other substitutions for y (such as $P(a, c) \vee P(a, b) \vee Q(c, c)$) may be missed.

Then *Skip* is the smallest safe factor of $Skip_1$ and $Skip_2$. The predicate `combine_lemma_literals/3` starts with $Skip_2$ and successively adds literals $L \in Skip_1$ if they are not unifiable with a literal K already in *Skip* with mgu σ, such that $(Skip)\sigma \subseteq Skip \cup \{L\}$. If this is the case, L is not added to *Skip* and *Skip* is replaced by $(Skip)\sigma$.

We have two possibilities of solving a positive subgoal *Lit*. The first one is to apply the reduction rule, if it is possible. We search in the current branch for a literal $\neg K$ such that *Lit* and K are unifiable. All literals in the branch are negative, since positive literals are always leaves. If the reduction attempt fails, a skip step is performed and L is added to the current list of lemma literals. Below we provide a possible implementation of this.

```
prove([Lit|Clause],Branch,...,Skip,...) :-
     ...
     member(Lit,Branch),
     prove(Clause,Branch,...,Skip,...),
     ;
     prove(Clause,Branch,...,Skip1,...),
     combine_lemma_literals([Lit],Skip1,Skip),
     ...
```

The base case for the predicate `prove/9` occurs when the current list of open subgoals is empty, i.e.,

```
prove([],_,...,[],...).
```

In that case, `prove` succeeds with an empty list of lemma literals.

The only rule of the calculus not described yet in this section is the start rule. A start clause is searched using `#` to index the head literal and then the core prover is called with the rest of literals of that clause. This is done from a top predicate `prove/2`. This predicate also checks if the set of skipped literals is empty when `prove/9` returns. If this is so, it means that a closed tableau has been found and the proof search finishes, else we assert the new lemma contained in `Skip` into the database as explained above and we start again. We show a possible code to perform this:

```
prove(Proof,MaxDepth) :-
     cla([#],C,Level),
     prove(C,[-(#)],...,Skip,...),
     ...
     (Skip = [] -> true ;
          assert_lemma(Skip),
                prove(Proof2,MaxDepth)),
          ...
```

The predicate `prove(-Proof,+MaxDepth)` takes two arguments. The first one corresponds to the proof found and is updated during the search. We do not detail this part. The second is used for the iterative deepening strategy and is explained in next Section 5.3.

A particularity of the predicate `assert_lemma(+Skip)` that is worth mentioning is that we check both forward and backward subsumption. First, if the clause contained in `Skip` is subsumed by another clause previously in the database (forward subsumption), the lemma is not added and `assert_lemma/2` fails. The search mechanism backtracks in order to construct a different tableau that generates another lemma. If the clause defined by `Skip` is not subsumed, we delete all clauses that it subsumes if any (backward subsumption), a head literal `L` is selected from it (the first one) and the fact `cla([L],B,Level)` is asserted. We explain the *level* of the lemma in the next section.

5.3 Depth First Search strategy

In Section 2.3.3 we introduced the necessity of providing Prolog with a complete search strategy and proposed depth first iterative deepening search as an alternative to Prolog's unbounded depth first search. In this section we discuss the details of our particular implementation of this procedure.

To bound the search the proof depth can be used, that is, the length of the active branch in the tree. We count only extension steps that involve clauses with free variables, i.e., we do not increase the length of the active branch when the extension step uses a ground clause. Reduction steps do not increase the current depth either and actually, they can only be performed when the goal to be proved is a positive literal, that otherwise would be part of a lemma. In each iteration, we consider exclusively proofs whose depth is less than the current maximum depth, which is increased in the next iteration. As described, this is a complete decision procedure for propositional logic and it can refute some unsatisfiable sets of first order logic clauses.

This procedure is enough for general connection-based theorem provers but our lemma extension needs of an additional feature to achieve completeness. In connection tableau calculus, a proof is found by constructing an unique closed tableau for a given set of clauses. However, with the intermediate lemma extension several intermediate open tableaux may be needed to generate lemmas that may be used in successive tableaux before the actual closed tableau is found. This implies that when using a lemma generated at a certain depth in a previous tableau, we have to update the length current branch accordingly, and not just by 1 as it was a normal extension step. Otherwise, it may be the case that infinitely many lemmas are generated at a certain depth, whereas the actual proof can only be found at a larger depth. In order to clarify this, let us consider the following example:

Example 29. Consider the set of clauses S:

$$S = \{\neg P(x) \vee \neg Q(f(x)), P(a), Q(x) \vee P(x), Q(f(b)) \vee \neg S(b), S(x) \vee \neg T(x), T(x), P(b)\}$$

This set is unsatisfiable and a closed tableau of depth 2 (2 extension steps involving not ground clauses) for it is shown in Figure 5.1.

However, when trying to find this proof using simple iterative deepening search bounded on the length of the active branch, the tableau in Figure 5.1 can never be constructed. Using just the clauses $\neg P(x) \vee \neg Q(f(x)), P(a), Q(x) \vee P(x)$ with a depth of 1 (only one extension step involving clause with free variables in the right branch) it is possible to generate infinitely many lemmas, $P(f(a)), P(f(f(a))), P(f(f(f(a)))) \ldots$ as it can be seen in Figure 5.2.

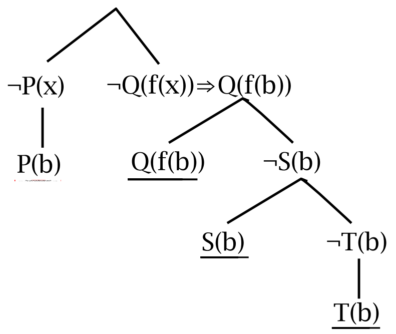


Figure 5.1: Closed tableau of depth 2

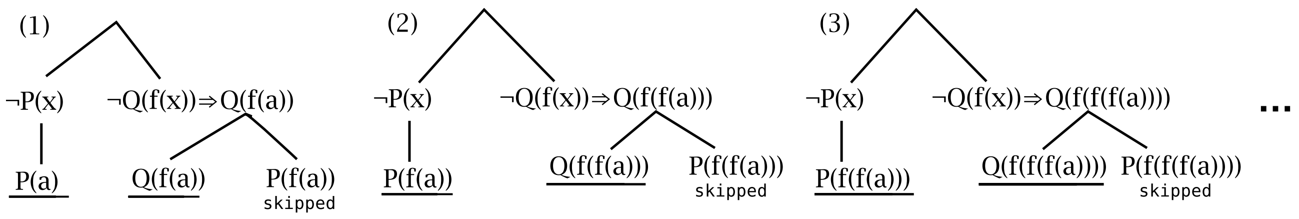


Figure 5.2: Generating infinitely many lemmas at depth 1

To solve this problem we introduce the concept of *level* of a clause C.

Definition 34. (Level of a clause). The *level* of a clause C with respect to a given initial set of clauses S is defined as follows:

$$level(C) = \begin{cases} 0 & \text{if } C \text{ is ground and } C \in S \\ 1 & \text{if } C \in S \text{ and it contains free variables} \\ 1 + \max\{level(D) : D \in skip(T)\} & \text{if } C \text{ is an intermediate lemma generated by tableau } T \\ \text{not defined} & \text{otherwise} \end{cases}$$

This level is represented in our implementation by the third argument in the predicate `cla(H,B,Level)`.

Then if the active branch B consists of the literals $L_1, \ldots, L_n$ belonging to instances of the clauses $C_1, \ldots, C_n$, the proofs considered are restricted to those with

$$length(B) = \sum_{i=1}^{n} level(C_i) < MaxDepth$$

In each step of the iterative search $MaxDepth$ is increased until a closed tableau is found. We consider then the depth of the proof to be bounded by the maximum length of its active branches. Notice that now the start clause participates also in the depth of the proof.

Example 30. Applying this to the previous example, the lemmas generated by tableaux (1), (2) and (3) in Figure 5.2 have respectively a level of 2, 3 and 4. The proof shown in Figure 5.1 can be constructed with $MaxDepth = 4$ but this is not the case for tableau (3) in Figure 5.2. The set of tableaux constructed by our prover to finally find the closed one is shown in Figure 5.3.

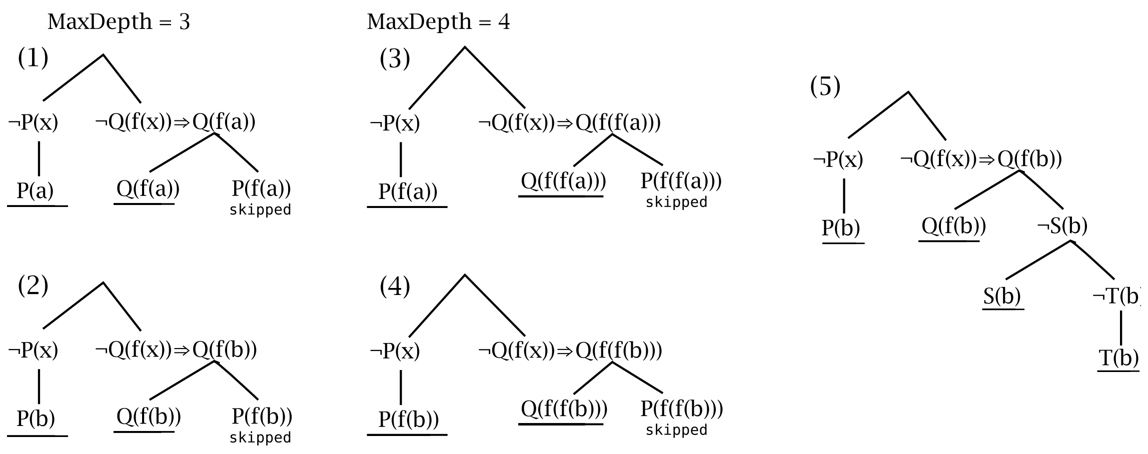


Figure 5.3: Example of proof constructed with $MaxDepth = 4$

To achieve the iterative deepening search that takes into account the level of the clauses, we need to insert and modify some lines of code in our prover. First, an additional clause for the predicate `prove/2`, to increase the maximum depth in case of failure of all alternatives at the previous $MaxDepth$. The first line of this clause checks that there is at least one not ground clause or lemma, otherwise the goal `prove(-Proof,+MaxDepth)` fails.

```
    prove(Proof,MaxDepth) :-
          cla(_,_,L), L>0,
          MaxDepth1 is MaxDepth+1, prove(Proof,MaxDepth1).
```

Also, we need to include in the predicate `prove/9` the necessary arguments and lines to check that the rule being applied fulfils the maximum depth restrictions, and to calculate the level of the next lemma generated. Just the code for negative subgoals has to be modified, since neither the skip operation nor the reduction rule are affected by the depth of the tableau. The modified code looks as follows:

```
prove([-Lit|Clause],Branch,...,GlobalLevel,MaxLevelAcc,MaxLevel) :-

      ...

      cla([Lit],B,Level),

      K is Level + GlobalLevel, K<MaxDepth,

      (Level>MaxLevelAcc -> MaxLevelAcc1 is Level; MaxLevelAcc1 is MaxLevelAcc), (*1)

      prove(B,[-Lit|Branch],...,K,MaxLevelAcc1,MaxLevel1), (*2)

      ...

   prove(Clause,Branch,...,GlobalLevel,MaxLevel1,MaxLevel), (*3)

   ...
```

`MaxDepth` is the current maximum depth allowed for a tableau in the search. The parameter `GlobalLevel` corresponds to the current level of the tableau, that is, to $GlobalLevel = length(Branch) = \sum_{i=1}^{n} level(C_i)$ before performing the next proof step. Basically, when an extension step is attempted with clause $\{Lit\} \vee B$ of level $Level$, we check that $K = Level + GlobalLevel < MaxDepth$, as we have explained above. When we progress with the next subgoals in B, the global level for them is K (*GlobalLevel* updated with the new clause used), as we can see in line `(*2)`

`MaxLevelAcc` is an accumulator used to calculate the current maximum level of the clauses in the tableau, in order to assign a level to a lemma generated within this tableau. The accumulator is being updated during the search and the final result is contained in `MaxLevel`. In line `(*1)` we check if the clause being used for the extension step exceeds the current maximum value and if this is so, `MaxLevelAcc` is updated. This new value of `MaxLevelAcc` is used when solving the subgoals in B and when they are solved, `MaxLevel1` contains the maximum level of a clause in this part of the tableau. This value is used as current maximum value in the next call to solve the subgoals contained in the siblings nodes of $\neg Lit$, as we can see in `(*3)`. Also, the global level for this call is the same as before solving $\neg Lit$, since the steps taken to do this do not affect the depth to the right of $\neg Lit$.

Finally, in the top-level call to this predicate by `prove/2`, `GlobalLevel` and `MaxLevelAcc` are initialised with the level of the start clause and the returned maximum value for the level of the clauses is assigned as level for the lemma generated (by an additional argument in `assert_lemma(+Skip,+Level)`).

```
prove(Proof,MaxDepth) :-
      cla([#],C,Level),
      prove(C,[-(#)],...,Skip,...,Level,Level,MaxLevel),
      ...
      (Skip = [] -> true ;
            assert_lemma(Skip,MaxLevel),
            prove(Proof2,MaxDepth)),
      ...
```

5.3.1 Refinement of subsumption

The introduction of *level* for each clause makes necessary to refine the definition of subsumption used until now, to include the level of the clause. It may be possible to generate the same lemma with two (or more) than one different tableau, each one with different level. Depending on the order of the original clauses or the order in which new lemmas are incorporated to the set, a lemma already in the set of clauses may forward subsume a new lemma that has been generated later in the search with smaller level. In this case a proof of a certain length may never be found.

To avoid this, the definition of subsumption is slightly modified as follows.

Definition 35. (Clause subsumption with level). A clause C *subsumes* another clause D if there exists a substitution σ such that $C\sigma \subseteq D$ and $level(C) \leq level(D)$.

5.4 Simple shortcuts

From all the simple shortcuts introduced in Section 2.2.2, re-use is the only one implemented. Additionally, regularity is checked every time that a new negative subgoal is tackled. In both cases, the substitution σ is not modified. That means that the regularity condition is satisfied if the open list of subgoals (the current tableau clause) does not contain a literal that is *syntactically* identical with a literal in the branch. It is implemented inserting the following line in the code:

```
prove([-Lit|Clause],Branch,...,GlobalLevel,MaxLevelAcc,MaxLevel) :-
```

```
        \+ (member(LitC,[-Lit|Clause]), member(LitB,Branch), LitC==LitB),
        ...
```

We use `member` to enumerate all the possible literals in the current tableau clause and in the branch, and then we check if there is a pair of syntactically identical literals. The whole sentence is negated with `\+Goal` and succeeds if `Goal` can not be proven. Another weaker form of regularity could have been implemented, called *strictness*. Strictness imposes that no ground clause is used more than once in a branch.

Notice that we do not need to check regularity for positive literals because it is possible just one occurrence of a positive literal in a branch, so regularity is guaranteed for them.

The same syntactic approach is used with re-use. To implement this shortcut, we use the list `Reuse`, that is added as an additional parameter to the head of the predicate `prove`. Intuitively, it is like the Prolog version of extending the tuple $C, S, Branch$ to $C, S, Branch, R$ in the formal definition of the calculus with re-use given in Section 2.2.2, to keep track of solved literals that can be re-used. The possibility of applying this shortcut is checked in first place when a new negative subgoal is tackled (before applying extension). As it has been said, the substitution σ is not modified, and therefore we only apply re-use if the list `Reuse` contains a literal that is syntactically identical with our current subgoal. Additionally, once a subgoal is solved, we must incorporate it to the list `Reuse`, before progressing to the rest of the open subgoals. We do that modifying the parameter in the call to `prove/9` for the siblings of $\neg Lit$. This is implemented with the following modifications to `prove/9` for negative subgoals:

```
prove([-Lit|Clause],Branch,...,Reuse,...) :-
            ...
            (member(-LitR,Reuse), Lit==LitR
            ;
            cla([Lit],B,L),
            prove(B,[-Lit|Branch],...,Reuse,...),
            ...
            ),
        prove(Clause,Branch,...,[-Lit|Reuse],...),
        ...
```

It is important to note that with this syntactic condition the application of re-use is perfectly compatible with the intermediate lemma extension without additional considerations. If a subgoal $\neg Lit$ in T is solved with a sub-tableau T' such that $skip(T') = \{L'_1, \ldots, L'_n\}$ and then it is re-used to solve $\neg K = \neg Lit$, it would be like copying T' below K. Then, when the whole tableau T is solved, we would have $skip(T) = \{L_1, \ldots, L_m, L'_1, \ldots, L'_n, L'_1, \ldots, L'_n\}$ that safely factors to $\{L_1, \ldots, L_m, L'_1, \ldots, L'_n\}$.

5.5 Outline of performance

The simple version of the prover can solve 2350 problems out of 13776 CNF and FOF problems in the TPTP Library v4.0.1., which corresponds to 17% of them, using a time limit of 120 seconds. The solved problems are almost evenly distributed between CNF and FOF: 1091 out of 6800 CNF problems and 1259 out of 6976 FOF problems are proved. It is still far from the performance results of leanCoP 2.0, which solves approximately 25% of the 5051 FOF problems in the release v.3.7.0 of the TPTP Library using a time limit of 600 seconds, with its regular version (a version that uses an enhanced definitional clausal transformation for FOF problems, regularity and re-use but no other shortcuts or pruning techniques).

At the view of the first test results, it seems that it is not enough to replace a big search space dedicated to find a deep, closed tableau by a series of `smaller` search spaces leading to more shallow tableaux and derived lemmas. Without a control mechanism in the lemma generation, the growth of the global search space prevent the intermediate lemma generation from improving the results achieved by the connection tableau calculus alone.

5.6 Alternative implementation: *compiling* the input set

We introduce briefly another possibility for the implementation of the prover, different from the one we have chosen. In our implementation, we *interpret* the input set in the sense that we apply the rules of the calculus to it and the clauses are always passive elements. Although we represent them in the database intuitively as clauses, they are not clauses from the point of view of Prolog, but simple facts of the form `cla(H,B,L)`.

An alternative implementation could take advantage of the fact that the intermediate lemma extension allows to regard a set of arbitrary clauses as a set of Horn clauses, that is, a Prolog program. Then, the idea would be to *compile* the input set as Prolog clauses, run the appropriate query and leave Prolog to deal with it. If the query results in a lemma, it would be compiled again in case it is not subsumed and the process would be restarted. To clarify this idea, let us consider a simple example with only propositional variables (predicates of arity 0)

Example 31. We are given the input set $S = \{A \vee B \vee \neg C, A \vee \neg B \vee \neg C, C \vee D, G \vee \neg A, G \vee \neg D, \neg G\}$, that would be compiled as:

```
c([d|X],X).
a([b|X],Y):- c(X,Y).
a(X,Y):- b(X,Z),c(Z,Y).
g(X,Y):-a(X,Y).
g(X,Y):-d(X,Y).
```

The two arguments of each predicate form a difference list that contains the lemma literals of the clause. For example, for clause `c([d|X],X).`, the lemma literals are `[d|X]-X`, i.e., `d`, and for clause `a([b|X],Y):- c(X,Y).` the lemma literals are `[b|X]-Y`, with `X-Y` being the lemma literals generated when solving `c(X,Y)`. The only all-negative clause, $\neg g$, is the query, and is called from a predicate prove of the form:

```
prove :-
   g(X,[]),
   (X=[] -> print('Unsatisfiable') ;
       safe_factoring(X,F),
       \+ subsumed_lemma(F),
       handle_lemma(F),
    prove).
```

When invoking `?- prove.` the goal `g(X,[])` is tackled. If it is solved, `X` will contain the resulting list of lemma literals, that are combined through safe factoring. If the corresponding lemma is not subsumed, the predicate `handle_lemma(F)` must assert a fact with for one of the literals as predicate, of the previous form, and then the process is restarted. An example of trace for the previous problem would be (we number and rename the variables of each clause on the left-most column, for clarity, and in the right the Prolog clause used and the unifier involved):

```
1.  c([d|X1],X1).
2.  a([b|X2],Y2):- c(X2,Y2).
3.  a(X3,Y3):- b(X3,Z3),c(Z3,Y3).
4.  g(X4,Y4):-a(X4,Y4).
5.  g(X5,Y5):-d(X5,Y5).
```

`?- g(X,[]).`

`?- a(X,[]).  (4, X4/X, Y4/[])`

`?- c(X2,[]).  (2, X/[b|X2], Y2/[])`

`?- c([d],[]).  (1, X1/[], X2/[d])`

`g(X,[])` suceeds with `X=[b,d]` that factors to itself

Clause `6.  b([d|X6],X6).` is added to Prolog's database

Start again `?- g(X,[]).`

Lemma `[b,d]` is generated as before and subsumed

Backtrack at `?- a(X,[]).`

`?- b(X,Z3),c(Z3,[]).  (3, X3/X, Y3/[])`

`?- b([d|X6],X6),c(X6,[]).  (6.  X/[d|X6],Z3/X6)`

`?- c(X6,[]).`

`?- c([d],[]).  (1, X1/[], X6/[d])`

`g(X,[])` suceeds with `X=[d,d]`, that factors to `F=[d]`

Clause `7.  d(X7,X7).` is added to Prolog's database

Start again `?- g(X,[]).`

Lemmas `[b,d]` and `[d]` are generated as before and subsumed

Backtrack at `?- g(X,[]).`

`?- d([],[]).  (7, , X/X7, X7/[])`

And finally `g(X,[])` suceeds with `X=[]`

This is a simplified version that works for the example but an implementation like that is incomplete. in general We must add the iterative deepening depth first search. This can be done just by extending each predicate with two arguments that represents the current depth and the maximum depth. The current depth is updated and checked in the body of each clause:

```
c([d|X],X,K,Max) :- K1 is K + 1, K < Max.
a([b|X],Y,K,Max):- K1 is K + 1, K < Max, c(X,Y,K1,Max).
...
```

And the maximum depth is added as argument to the predicate `prove`, that now has to be called from a predicate `top_prove`, and also needs an additional clause to increase the maximum depth in case of failure.

The previous example contained just one all-negative clause, that is, one query. It is simple to deal with the case that several possible start clauses or queries exist in the input set S. If $C_1, \ldots, C_n$ are start clauses, it suffices to add a new predicate symbol G that do not occur in S as the conclusion literal to each start clause, $C_i \vee G$ and then make $\neg G$ the only start clause.

This type of *compiled* implementation has the obvious advantage of being faster. It does not use the reduction rule and takes full advantage of the Prolog inference engine. The disadvantages are that the kind of shortcuts that can be used is much more limited, as there is no easy way of keeping track of the literals that have been solved or the substitution used at some point of the search.

Chapter 6

Architecture of the prover (II): Extensions

Throughout the previous chapter we have described the implementation of a basic prover based on the Connection Tableau Calculus with the Intermediate Lemma Extension, as a Prolog technology theorem prover. We have explained the main decisions taken and provided the most relevant pieces of the code, in order to give the reader a general idea of the whole system. In this chapter we build upon the simple version of the prover to implement two extensions discussed in Chapters 3 and 4, in two new versions of the prover. Namely, we have programmed the mechanism based on RUE resolution to deal with equality literals formalised in Section 4.3.2 and the main pruning technique presented for the SOL Tableau Calculus, the Local Failure Caching procedure detailed in Section 3.2.2. The next two sections provide details on both versions.

Moreover, we have developed a further refinement of the Intermediate Lemma Extension, called *Specific Lemmas*, as case study. This extension has been also programmed as an independent version of the simple prover and it has been inspired by the idea of production fields taken from the SOL Tableau Calculus together with a feature of the Hyper-resolution method. This is comprehensively explicated in Chapter 7.

6.1 Built-in mechanism for equality reasoning

In order to test the effectiveness of the built-in equality mechanism described in Section 4.3.2, we have implemented a version of the prover presented throughout Chapter 5 that incorporates the new rules for equality literals and does not rely on the equality axioms. In this section we describe the modifications and additions made to the basic code, presenting just the relevant pieces to maintain clarity.

The basis of the new rules is the concept of *disagreement set* and as a consequence, a predicate to calculate disagreement sets between literals has been implemented, `disagreement(+S,+T,-D)`, where `D` is the top-most/origin disagreement set between `S` and `T`. The associated disagreement substitution is implicitly defined and stored by Prolog's unification mechanism.

The existing predicate that has needed more adjustment has been obviously the core predicate, `prove/9`. It has been extended with two clauses, in order to implement the new disagreement extension and decomposition rules. We recall that two cases were considered for the disagreement extension rule, equality and non-equality subgoal, so we have written two different clauses to deal with them.

First we discuss the code for the disagreement extension rule with a non-equality subgoal:

```
prove([-Lit|Clause],Branch,...,Skip,...) :-
      Lit\=(_=_),
      ...
      cla([Lit1],B,Level), disagreement(Lit,Lit1,D), (*1)
      append(D,B,Rest), (*2)
      prove(Rest,[-Lit|Branch],...,Skip1,...), (*3)
      ...
      prove(Clause,Branch,...,Skip2,...),
      combine_lemma_literals(Skip1,Skip2,Skip),
      ...
```

Apart from the first line that checks if the subgoal has a predicate symbol other than equality, the numbered lines are the differences with the normal predicate for negative subgoals in which the extension rule is implemented. Line (*1) presents the main drawback of the disagreement extension rule as implemented. In order to apply the disagreement extension rule, we must find a clause with a head that has a disagreement set $\mathcal{D}$ with *Lit*. Because *Lit* and the head of that clause are not necessarily unifiable (and hence this rule is applied instead of the normal extension rule), we loose the possibility of using *Lit* as index for the first argument in the predicate `cla/3`. We use `cla([Lit1],B,Level)` to enumerate the clauses in the current database and `disagreement(Lit,Lit1,D)` to find a suitable one. We have implemented it in this way to use the same database as for the simple prover and to keep the modification minimum. In a future version of the prover, the predicate `cla/3` could be extended to have an additional argument with the predicate symbol and use it for indexing in the disagreement extension rule.

When a disagreement set $\mathcal{D}$ has been found, the list of open subgoals that must be proved to solve $\neg Lit$ is $B \cup \mathcal{D}$. Then, in line (*2) we append together the disagreement set of `Lit` and `Lit1`, `D`, with the literals in `B` using the predefined predicate `append(-L1,-L2,-L3)` and the result, `Rest`, is now the list of open subgoals to prove in order to finish solving `-Lit`.

The implementation of the disagreement extension rule for equality literals $Lit = \neg(s_1 = s_2)$ follows the same structure, but considering symmetry and adding the extra inequality term to list of open subgoals to be proved to solve *Lit* (see Section 4.3.2). Since the decomposition rule can only be applied to negative equality subgoals, it is implemented in the same clause as the disagreement extension rule (in a similar way as the reduction and skip rules for positive subgoals) and it is tried first as stated in the order indicated in Section 4.3.2.

```
prove([-(S1=S2)|Clause],Branch,...,Skip,...) :-
   ((S1=S2, D=[] ; \+ atomic(S1), \+ atomic(S2), disagreement(S1,S2,D)) -> Rest=D, (*1)
     ... ;
     cla([T1=T2],B,Level), (*2)
     (disagreement(S1,T1,D) -> append([-(S2=T2)|D],B,Rest) ;
        (disagreement(S1,T2,D) -> append([-(S2=T1)|D],B,Rest) ;
           (disagreement(S2,T1,D) -> append([-(S1=T2)|D],B,Rest) ;
              (disagreement(S2,T2,D) -> append([-(S1=T1)|D],B,Rest) ; fail)))),
    ...
  ),
  prove(Rest,[-Lit|Branch],...,Skip1,...),
  prove(Clause,Branch,...,Skip2,...),
  combine_lemma_literals(Skip1,Skip2,Skip),
  ...
```

Line `(*1)` implements the decomposition rule, which can be applied if $s_1 = s_2$ with an empty disagreement set $\mathcal{D}$ or if s_1 and s_2 are compound terms (i.e., they are of the form $f(r_1, \ldots, r_n)$ with f a function symbol of arity at least 1), checked with the two negated calls to `atomic(+Term)`, and they have a top-most disagreement set $\mathcal{D}$ obtained with `disagreement(S1,S2,D)`. If these conditions hold [1], we must prove the inequalities in $\mathcal{D}$ and hence `Rest=D`.

If the attempt to perform a decomposition step at $\neg(s_1 = s_2)$ fails, we try a disagreement extension step. First we select a possible clause from the database in line `(*2)` and then we check whether it is suitable or not. The 4 lines below `(*2)` test in turn the 4 possibilities of forming a disagreement set between s_1, s_2 and t_1, t_2 due to the symmetry property of equality. If none of them succeeds, the predicate `fail` (that always fails) is called. There may be more than one disagreement set, and in case of failure and backtracking, several or all of them can be explored during the search. The list of open subgoals when the disagreement set $\mathcal{D}$ has been found is $\mathcal{D} \cup B \cup \{s_i \neq t_j\}$, where s_i and t_j are the terms of the equations that do not participate in $\mathcal{D}$. This list is assigned to `Rest` using the built-in predicate `append(-L1,-L2,-L3)` as before. The rest of the body is not modified.

Regarding our iterative deepening depth-first search method, two considerations are needed for the new rules:

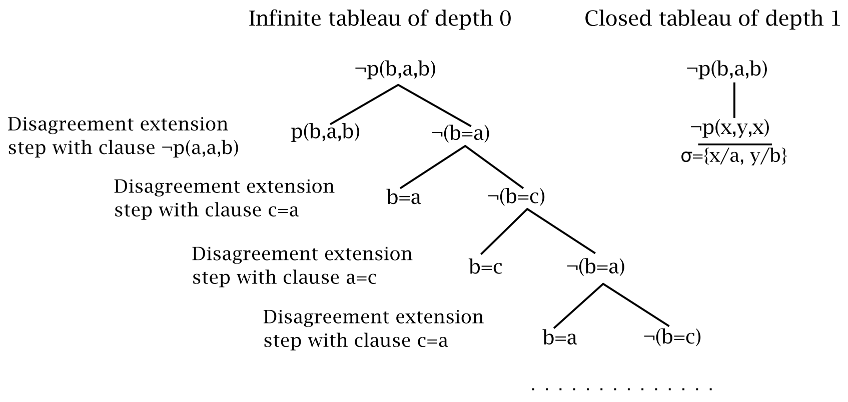
- Applying a decomposition step is assigned a level of 1, that is, the global level is increased by 1 when compared against the maximum depth allowed to check if the rule is applicable.
- When a disagreement extension step is applied using clause C, the global level is increased by $\mathit{level}(C) + 1$. The reason is that for clauses that are ground and therefore have a level of 0, performing this step must be counted because of all the new inequalities generated. Otherwise, the calculus with the RUE resolution rules could fail when looking for a closed tableau for a set of clauses, since we could generate infinite tableaux with depth 0 whereas the solution actually includes other step with a not ground clause and must be found at a deeper level. Example 32 provides clarification for this.

Example 32. Let us consider the following unsatisfiable set of clauses:

$$S = \{P(a, a, b), \neg P(b, a, b), c = a, P(x, y, x)\}$$

Clearly, there is a closed tableau using clauses $\neg P(b, a, b)$ and $P(x, y, x)$ and one extension step. This tableau would have depth 1, since the extension step involves a not ground clause. However, if the disagreement extension step is assigned only the level of the clause involved, the prover can not find the previous closed tableau since an infinite tableau of depth 0 can be generated using clauses $\neg P(b, a, b)$ and $c = a$, as depicted in Figure 6.1.

[1]The *if-then-else* construct `Cond->Then;Else` succeeds if `Cond` and `Then` succeed, or if `Cond` fails and then `Else` succeeds



Infinite tableau of depth 0
Closed tableau of depth 1
¬p(b,a,b)
Disagreement extension step with clause ¬p(a,a,b)
p(b,a,b)
¬(b=a)
Disagreement extension step with clause c=a
b=a
¬(b=c)
Disagreement extension step with clause a=c
b=c
¬(b=a)
Disagreement extension step with clause c=a
b=a
¬(b=c)
. . . .
¬p(b,a,b)
¬p(x,y,x)
σ={x/a, y/b}

Figure 6.1: Example of infinite tableau of depth 0 and closed tableau of depth 1

It is necessary to point out that this case of infinite tableau can be avoided with regularity, since it does not allow two identical literals to appear in the same branch. However, as we are introducing in this section how to implement the RUE resolution rules for tableau in general (without considering additional shortcuts), we need this increment in the global level for the disagreement extension rule. When combined with regularity, the previous case of infinite tableau will be pruned by the iterative search before that the regularity condition is violated.

Finally, the clause of `prove/9` implementing the normal extension rule is modified to include the symmetry property for negative equality subgoals by adding lines `(*1)` and `(*2)` in the code:

```
prove([-Lit|Clause],Branch,...) :-
     ...
     (Lit=(Lit1=Lit2) -> (*1)
        (cla([Lit1=Lit2],B,Level) ; cla([Lit2=Lit1],B,Level)) (*2)
         ;
        cla([Lit],B,L)
        ),
        prove(B,[-Lit|Branch],...),
        ...
  prove(Clause,Branch,...),
  ...
```

6.2 Pruning methods from the SOL Tableau Calculus

As it has been already argued in various sections of this report, the similarities shared by the lemma generation and consequence finding tasks have motivated our attempt to incorporate successful methods and pruning techniques developed for the latter in our prover. In this section we explain how we have incorporated skip minimality and local failure caching, taken from the SOL Tableau Calculus described in Section 3.2.1, into the basic prover presented in Chapter 5.

First, we must note that the rules of the SOL Tableau Calculus and the rules of the intermediate lemma extension are not exactly the same. In particular, there are three main differences and for each one we explain why it does not involve any difficulties:

- The SOL skip rule is applied *optionally*, that is, a literal that can be potentially part of a consequence can be also closed by an extension/reduction step. In our case, the only alternative to skip is the reduction rule. This affect neither the skip minimality property nor the local failure caching procedure.
- There is a rule in SOL tableau calculus, skip-factoring, not present in the connection calculus with the intermediate lemma extension. This rule is indeed not defined in the set of rule formalised in Section 3.1 but our implementation actually contains a restricted version of it. We recall from the previous chapter the safe factoring operation performed when joining two sets of lemma literals produced in two different branches of the tableau, and implemented in the predicate `combine_lemma_literals(+Skip1,+Skip2,Skip)`. As defined for the SOL calculus, skip-factoring is non-safe, but since this calculus computes all consequences in turn (that is, backtracking is performed through skip-factoring, as for any other rule), no solution is missed. Since we do not backtrack through the operation of combining two sets of lemma literals, safe-factoring is required.
- There is a rule in the intermediate lemma extension devoted to the lemma derivation itself. This rule assumes a solved tableau as premise and has the effect that a lemma is derived from it and added to the input set of clauses. Although this has not been defined explicitly as a rule in the SOL tableau calculus, it is done implicitly in the description of the procedure *EnumConqs(T)* that we provided following the authors' nomenclature. We refer specifically to the lines:

```
...
if $T$ is a solved tableau, then:
  – $C := skip(T)$
  – $Conqs := \mu(Conqs \cup C)$
...
```

That is, the derived lemma/consequence C is added to the set of consequences (the input set in our case) and the subsumed clauses are removed.

With the previous details already clarified, we are ready to explain how we have implemented the two pruning techniques mentioned above. Due to the complexity of the local failure caching procedure, we have decided to dedicate a separate subsection to it. Regarding skip-minimality, we advanced that this is a restriction of the kind of regularity, rather than a shortcut. To maintain skip-minimality, we must ensure that for any solved tableau T produced during the proof search, $skip(T)$ is not subsumed by any clause in the database. This can be achieved in several ways and we have opted to place line `(*1)` into the code of `prove/9` for negative subgoals:

```
prove([-Lit|Clause],Branch,...,Skip,...,MaxLevel) :-
           ...
           cla([Lit],B,L),
           prove(B,[-Lit|Branch],...,Skip1,...,MaxLevel1),
           ...
     prove(Clause,Branch,...,Skip2,...,MaxLevel),
     combine_lemma_literals(Skip1,Skip2,Skip),
     \+ subsumed(Skip,MaxLevel), (*1)
     ...
```

The level assigned to $skip(T)$ to test if it is subsumed according to the redefined definition of subsumption given in Section 5.3.1 is the maximum level of sub-tableau T.

6.2.1 Implementation of Local Failure Caching

Before detailing the particularities of the implementation, we remind briefly characteristics of this pruning method, linking them to the context of the intermediate lemma extension. The local failure caching procedure is based on the concept of failure substitution. When a subgoal K is solved while developing a tableau, the partial substitution σ found is stored at node K. If eventually the proof cannot be completed and the search procedure backtracks over K, σ is turned into a failure substitution. In any alternative solution process of K from T, if a substitution τ is computed but is such that one of the failure substitutions stored at K, σ, is more general than τ, then the proof procedure immediately backtracks. By doing so, we avoid solving K again with a more specific substitution If K where to be solved with τ, any possible derived lemma $skip(T)$ would be subsumed by a previous lemma C generated from K using σ, because there would exist θ such that $\tau = \sigma\theta$. When the search procedure backtracks at a node above K, all failure substitutions stored at K can be deleted.

The main difficulty faced when implementing this procedure by modifying our basic prover is how to store the substitutions. So far our code does not have to deal with substitutions of any kind, as they are handled implicitly by Prolog. Now we must incorporate the ability of storing and accessing them from different points in the whole search tree.

As we did with clauses, we use the Prolog database to store substitutions and because substitutions are computed through Prolog's unification algorithm and are never made explicit, we must store $K\sigma$ to actually have the substitution σ computed at node K. Then, for two different substitutions σ and τ computed at K, σ is more general than τ if $K\sigma$ subsumes $K\tau$ but $K\tau$ does not subsume $K\sigma$.

In order to access the substitutions, we must be able to link them in a univocal manner with the literal K. Moreover, a clause C can be used in a tableau more than once, and so a literal K can appear more than once in different nodes in the tableau. We need to identify separately substitutions for K stored at different nodes. For the first identification issue, we have assigned an unique number to each negative literal [2] in the input set. This is done before the proof search start, when the input set is asserted into the Prolog's database. Then this number can be used to index substitutions stored at nodes with K in a tableau. Regarding the identification of different instances, a simple method is to use the branch above K, composing a *branch numerical code* with the list of numbers assigned to each literal in the branch (all of them must be negative as they are not leaves).

To meet the identification requirements exposed above, literals of a clause $C = L_1 \vee \ldots \vee L_n$ are stored in the database as pairs (L_i, id_i), where id_i is an unique natural number greater than 0 for negative literals and 0 for positive literals. We have maintained the structure for the predicate `cla(-Head,-Body,-Level)`. For example, clause $\underline{P(x,a)} \vee \neg Q(y) \vee \neg R(b) \vee P(y,b)$ could be stored as `cla([(p(X,a),0)],[(-q(Y),1),(-r(b),2),(p(Y,b),0)], 1)`. The code for a branch $B = \neg K_1, \ldots, \neg K_m$ is the list $(id_1, \ldots, id_m)$ where id_i is the number assigned to literal K_i. Substitutions are stored using two predicates, `node/3` and `nodef/3`, the first for substitutions in general and the second for failure substitutions. Both share the same structure `node/nodef(-Id,-Subst,-CodeOfBranch)`, so for σ stored at node K, `-Id` is the numerical identifier of K, `-Subst` is $K\sigma$ (stored as positive literal, as the sign is irrelevant) and `-CodeOfBranch` is the code of the branch above K. The empty branch is assigned the code `#`, for it represents the root of the tableau as the head of the start clauses.

Notice that it is important to use fresh variables in the literal $K\sigma$ when storing a new substitution.

[2]Leaving positive literals unidentified makes impossible to store substitutions at them. Substitutions at positive literals are actually relevant because different applications of the reduction rule can lead to different paths in the search and we may be able to cut some of them. The reason why we have considered applying local failure caching exclusively to negative literals is simplicity. In any case, reduction is considerably less applied than extension and given that this is our first attempt of incorporating this technique into the intermediate lemma extension, we have left positive literals for future versions.

$K\sigma$ must not be affected by other unifications performed to its right, nor by subsequent substitutions computed at K when exploring alternative paths.

Example 33. We illustrate everything explained so far with an example. Let S be the set of clauses

$$S = \{\neg Q(x,y) \vee \neg P(x) \vee \neg R(y,x), \underline{Q(a,z)} \vee \neg R(b,w), \underline{R(u,c)} \vee P(b), \underline{P(a)}, \underline{R(v,b)}\}$$

and suppose that it is stored in the database as:

```
cla([#],[(-q(X,Y),1), (-p(X),2), (-r(Y,X),3)], 1).
cla([(q(a,Z),0)],[(-r(b,W),4)], 1).
cla([(r(U,c),0)],[(p(b),0)], 1).
cla([(p(a),0)],[], 0).
cla([(r(V,b),0)],[], 1).
```

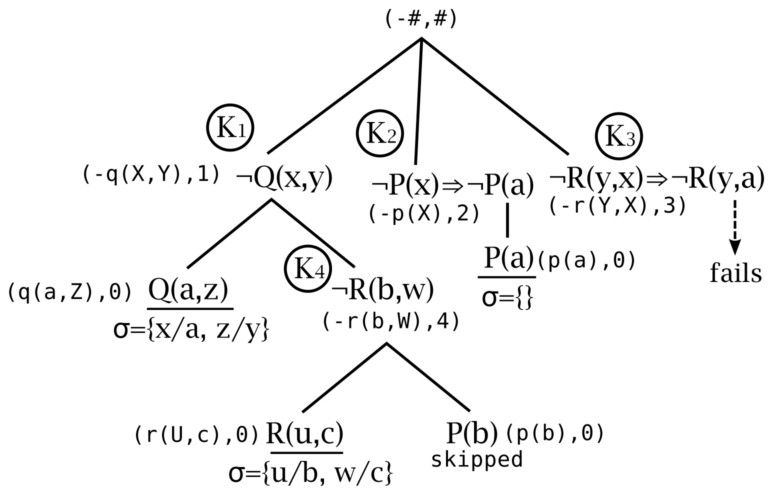


Figure 6.2: Example of tableau with identification of literals and branches

In Figure 6.2 we can observe a possible tableau developed during a refutation search for S. The substitutions stored at nodes K_1, K_4, K_2 just before node K_3 is explored are:

K_1: `node(1,q(a,Y1),[#]).`

K_4: `node(4,r(b,c),[1,#]).`

K_2: `node(2,p(a),[#]).`

When attempting to solve subgoal $\neg R(y,a)$ at node K_3 the procedure fails and hence backtracks first to node K_2, where no alternatives are available and finally to node K_1. At this point, substitutions for K_1 and K_2 are turned into failure substitutions and all substitutions stored below K_1 and K_2 are deleted (i.e., those with a branch's code $(\#, 1)$ and $(\#, 2)$ respectively). The substitutions database is left with:

K_1: `nodef(1,q(a,Y1),[#]).`

K_2: `nodef(2,p(a),[#]).`

To finish with this section, before summarising the performance of the extensions presented in this chapter, we provide the essential pieces of code needed to implement the local failure caching procedure, that is, to store, delete, use and turn into a failure each substitution constructed during the search. The clause of our core predicate `prove/9` that deals with negative subgoals is extended as follows:

```
prove([(-Lit,Id)|Clause],Branch,...) :-
   ...
      cla([(Lit,_)],B,Level),
      code(Branch,CodeBranch), (*1)
      \+ turn_into_failure_substitutions(Id,CodeBranch), (*2)
      \+ (nodef(Id,Subst,CodeBranch), (*3)
      ..., subsumes(Subst,Lit), (*4)
      ..., \+ subsumes(Lit, Subst)), (*5)
      ...
      delete_failure_substitutions(Id,CodeBranch), (*6)
      prove(B,[(-Lit,Id)|Branch],...),
      copy_term(Lit,NewSubst), assert(node(Id,NewSubst,CodeBranch)) (*7)
   prove(Clause,Branch,...),
   ...
```

Line `(*1)` is a simple call to a predicate `code(+Branch,-CodeOfBranch)` that given a list of literals contained in `Branch` calculated the code representing that branch (the list of codes of literals in `Branch`). This is used in line `(*2)` by a predicate that performs a search in the database to find all substitutions for the current subgoal (identified by `Id`) and that branch. If there is any and we are in this part of the code, it is because we have backtracked at the node. Hence the substitutions are turned into failure by this predicate. The reason why line `(*2)` is negated is because `turn_into_failure_substitutions/2` always fails. This is a very common trick employed in Prolog to compute all the solutions for a predicate, and it is actually how the universal quantifier $\forall x A(x) \rightarrow F(x))$ can be implemented: `all(X, F, A) :- \+ (F, \+ A).`

Lines `(*3)`, `(*4)` and `(*5)` check that there is no failure substitution σ stored at $\neg K$ (retrieved from the database with line `(*3)`) such that $K\sigma$ subsumes $K\tau$ (line `(*4)`, $K\tau$ is now *Lit*, after implicit unification to apply the extension rule) but $K\tau$ does not subsume $K\sigma$ (line `(*5)`).

Line `(*6)` is a call to a predicate that deletes all failure substitutions stored below $\neg K$, since we are going to explore a different path. We identify them with `Id` and the current branch. After the call to compute the subtree that solves $\neg K$, we must store the new substitution found. This is coded in line `(*7)`. First with `copy_term(+In,-Out)` we get fresh variables for those in `Lit` and then we assert a new fact representing $K\sigma$ in the database.

6.3 Outline of performance

The results obtained from the TPTP Library by the version with the built-in equality mechanism are disappointing in the sense that better results can be achieved with the addition of the equality axioms. This version can solve 2289 problems out of 13776 CNF and FOF problems, which is 61 problems less than the simple version. Furthermore, the simple version can solve more problems that contain equality and more problems that only contain equality (pure equality problems). All results are summarised in Table 8.2 in Chapter 8, dedicated to performance analysis. This suggests that rather than increasing efforts in creating and improving better built-in mechanisms for tableau systems, it would be more effective to devise more efficient pruning techniques and shortcuts, while equality is dealt by explicit use of the axioms.

With respect to the SOL tableau calculus version, the results are as expected, worse than the simple version and also worse than the version with the RUE rules. It can solve only 2171 problems from the TPTP Library, which situates it 179 problems below the simple version. We do not believe that this is caused by the unsuitableness of local failure caching for refutation finding, but by our particular adaptation and implementation. We have not implemented the SOL Tableau Calculus itself, with the $EnumConqs(T)$ procedure defined in Section 3.2.1, but have embedded its rules and pruning techniques into our previously implemented simple version, in a relatively forced way.

Chapter 7

Case study: Specific Lemmas

Apart from the pruning techniques discussed in Sections 3.2.2 and 6.2, one of the features the SOL Tableau Calculus that apparently offers more possibilities to control the lemma generation is the use of *production fields*. Whereas the Local Failure Caching procedure can be directly applied to a normal theorem prover used for refutation finding, a language to differentiate which consequences are sought is of little usefulness when the only consequence of interest is $\perp$, the empty clause or a a closed tableau. However, as we have mentioned before, the generation of intermediate lemmas can be seen as a special case of consequence finding even if the ultimate goal of the search is to find a closed tableau. Through this chapter, we will study carefully a possible modification of the Intermediate Lemma Extension based on the production fields defined for the SOL Tableau Calculus combined with an idea from the *Hyper-resolution* method.

As pointed out in Section 3.2.1 there are substantial differences between *consequences* and *intermediate lemmas*. Consequences as such do not participate directly in the search procedure, but they are the result of it. Intermediate lemmas are intermediate results that directly participate in the search procedure. For that reason, if we want to restrict or specify in some way a language that must be satisfied by lemma and conclusion literals, this language must fulfil some requirements to guarantee soundness and completeness of the proof method. Some of those requirements are obvious. For example, every lemma literal must satisfy the language for conclusion literals, otherwise it would not be possible to choose a conclusion literal when a new clause is generated as a lemma. Moreover, if a literal L belongs to the language, $\overline{L}$ must not belong to it, not only to avoid generating tautologies in the case of lemma literals but also because that would not guarantee the existence of start clauses in unsatisfiable sets (recall the argument given for that in Section 3.1 or consider the unsatisfiable set $\{P(a), \neg Q(x) \vee \neg P(a), Q(b)\}$ and suppose that we were to allow $P(a), \neg Q(x)$ and $Q(b)$ to be lemma/conclusion literals). [1]

For consequence finding, in order to guarantee completeness in the SOL Tableau Calculus, it was enough to ensure that the production field was stable. We will see that this property is not sufficient for our purpose. So far we have considered all positive literals to be lemma and conclusion literals. If we want to further restrict this language without any additional consideration, the resultant method would be probably incomplete. We present two examples to illustrate this.

Example 34. Suppose that we choose an stable production field $\mathcal{P} = \{P(x,y)\}$ and restrict both lemmas and conclusion literals to belong to $\mathcal{P}$, without any further consideration. Let S be the following unsatisfiable set of clauses:

[1] In fact, the existence of start clauses can be easily guaranteed for a set S by simply adding a new literal G, that does not occurs in S, to each clause $C_i \in S$, and then adding $\neg G$ to S as the only start clause: $\{\neg G, G \vee C_1, G \vee C_2, \ldots, G \vee C_n\}$. This is a minor problem when defining a new language for lemma/conclusion literals, as we will see through this section.

$$S = \{\neg P(a,b), P(x,z) \vee \neg Q(z,y), Q(b,c)\}$$

Applying Connection Tableau Calculus with the Intermediate Lemma Extension to find a closed tableau for S would fail, as we can observe in Figure 7.1. $\neg P(a,b)$ and $Q(b,c)$ are the possible start clauses and clause $\underline{P(x,z)} \vee \neg Q(z,y)$ only has the choice of $P(x,z)$ as conclusion literal. Literals $\neg Q(b,y)$ and $Q(b,c)$ at nodes N_1 and N_2 can be neither closed in any way nor skipped to be part of a derived lemma because they do not belong to $\mathcal{P}$.

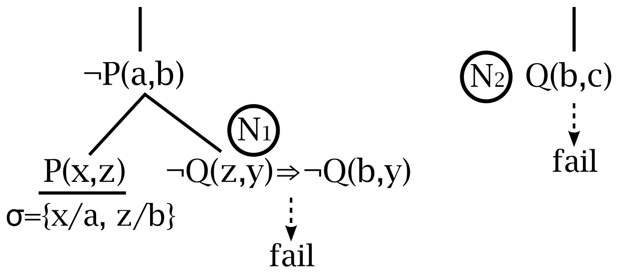


Figure 7.1: Example of failure with restriction $\mathcal{P} = \{P(x,y)\}$

If in Example 34 we had permitted $Q(x,y)$ to be chosen as conclusion literal, we would have been able to find a refutation for S, by applying and extension step with clause $Q(b,c)$ at node N_1. Nevertheless, even if we choose a more general language for conclusion literals that considers all predicate symbols in the language, completeness is not ensured. Next Example 35 shows us the case in which we choose different languages for lemma and conclusion literals

Example 35. Suppose that we consider $\mathcal{P}_L = \{P(x,y)\}$ to be the restricted language for lemma literals and $\mathcal{P}_C = \{P(x,y), \neg Q(x), R(x)\}$ to be the restricted language for conclusion literals, and we are given the following unsatisfiable set of clauses:

$$S = \{R(x) \vee P(y,x), \neg Q(y) \vee R(c) \vee \neg P(b,y), \neg R(c), \neg P(a,z) \vee Q(z)\}$$

Possible start clauses are $\neg R(c)$ and $\neg P(a,z) \vee Q(z)$, and suppose that we choose $R(x)$ and $\neg Q(y)$ to be conclusion literals in $\underline{R(x)} \vee P(y,x)$ and $\underline{\neg Q(y)} \vee R(c) \vee \neg P(b,y)$, respectively.

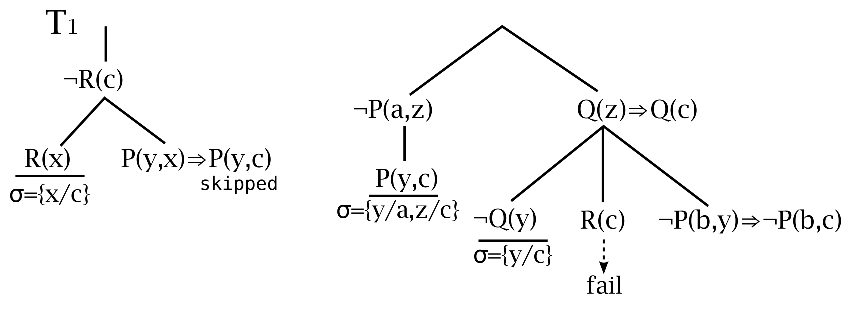


Figure 7.2: Example of failure with restrictions $\mathcal{P}_L = \{P(x,y)\}$ and $\mathcal{P}_C = \{P(x,y), \neg Q(x), R(x)\}$

Although we are able to generate the lemma $P(y,c)$ with tableau T_1, Figure 7.2 shows that the procedure fails with any start clause and with any sequence of steps applied. Again, the problem is that we encounter a node in which neither the skip rule is applicable (because the literal does not satisfy P_L) nor a extension step is possible (because there are not any clause with a matching selected conclusion literal).

In Figure 7.3 we can see a possible refutation obtained with the intermediate lemma extension considering $\mathcal{P}_C$ to be the language satisfied conclusion and lemma literals.

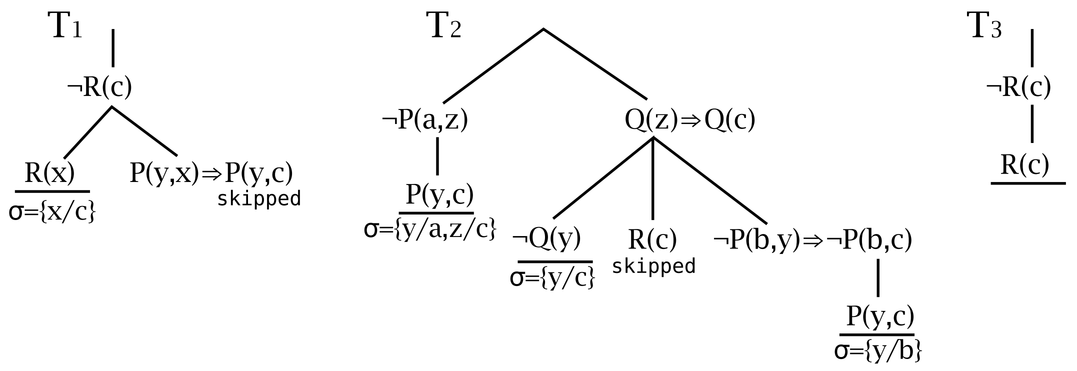


Figure 7.3: Refutation for S with restriction $\mathcal{P}_C = \{P(x, y), \neg Q(x), R(x)\}$

Examples 34 and 35 give us an intuition of what is happening. It seems that we must ensure that our language for lemma literals (and hence also for conclusion literals) *in some sense covers* all possible predicate symbols in the language. We formalise this intuition in the next subsections.

7.1 Uniform interpretations

Apart from the idea of production fields taken from the SOL Tableau Calculus, our approach has been inspired by an interesting feature of the *hyper-resolution* method: the use of *uniform interpretations*. Hyper-resolution [39] is a sound and complete refinement of resolution (see Section 1.3.1), backbone of the widely used Otter family of theorem provers [26]. Let us introduce briefly this theorem proving technique.

In the hyper-resolution strategy, clauses are divided into *nucleii* (those with one or more negative literals) and *electrons* (those with only positive literals). Hyper-resolution steps occurs between one nucleus and one or more electrons and there must be an electron clause for each of the positive literals in the nucleus. The nucleus is sequentially resolved with electrons until there is no more negative literals, so several intermediate binary resolution steps are combined into one hyper-resolution step. Note that each electron involved in an intermediate step can be a different copy/renamed instance of the same electron clause. Final resolvents, called *hyper-resolvents*, are always electrons since only positive literals are left.

Example 36. Consider the set of clauses of Example 35.

$$S = \{E = R(x) \vee P(y, x), C_1 = \neg Q(y) \vee R(c) \vee \neg P(b, y), C_2 = \neg R(c), C_3 = Q(z) \vee \neg P(a, z)\}$$

Some possible hyper-resolvents formed from S are:

- $R_1 = \mathcal{HR}(C_3, E) = Q(z) \vee R(z)$
- $R_2 = \mathcal{HR}(C_1, R_1, E) = R(c) \vee R(z)$
  - First resolve $\neg Q(y) \vee R(c) \vee \neg P(b, y)$ with $Q(z) \vee R(z)$ to give $R(c) \vee \neg P(b, z) \vee R(z)$
  - Then resolve $R(c) \vee \neg P(b, z) \vee R(z)$ with $R(x) \vee P(y, x)$ to give $R(c) \vee R(z) \vee R(z) =$ $= R(c) \vee R(z)$

The restrictions imposed by hyper-resolution are to some extent similar to those imposed by the intermediate lemma extension, with nucleus playing the role of start clauses and positive literals being lemma and conclusion literals. The result of an hyper-resolution step is always electrons (or the empty clause), and the result of a sequence of extension/reduction steps in our framework is always a solved tableau from which we can derive a lemma (or a closed tableau). These similarities explain why we have turned to hyper-resolution when trying to refine the intermediate lemma extension in the connection calculus.

Hyper-resolution has an interesting feature. Suppose a Herbrand interpretation I that assigns $false$ to all atoms, so that all negative literals are *true*. Then, any nucleus is *true* in I and any electron is $false$ in I, and so are any hyper-resolvents. I splits a given set of clauses S in two sets:

- $S_1 = \{C \in S : I \models C\} = nucleii$
- $S_2 = \{C \in S : I \not\models C\} = electrons$

If S is unsatisfiable, S_1 and S_2 are non-empty. It is very easy to see that any instance of a nucleus is itself a nucleus, and that every instance of an electron is also an electron. With respect to the sets, instances of clauses in S_1 and S_2 belong to S_1 and S_2 respectively. For each literal in S_1, either all instances are *true* or all instances are $false$ in I, so whatever unifiers are involved electrons remain electrons and nucleii remain nucleii. Hyper-resolution can be generalised to any Herbrand interpretation that has those properties and splits S in two such sets. The strategy to follow is still to resolve using one clause from S_1 and one or more clauses from S_2, but never more than one clause from S_1, until the result of the intermediate steps is a clause in S_2. Any Herbrand interpretation that satisfy those criteria is called an *uniform interpretation.*

Definition 36. (Uniform interpretation). A Herbrand interpretation I is an *uniform interpretation* if for any literal L in the first-order language and for any substitution σ such that $L\sigma \in S$, $I \models L$ iff $I \models L\sigma$. In other words, if I assigns to any instances of L the same truth value as L.

Since truth values for literals are preserved after substitutions, so are for clauses and thus the criteria stated above are satisfied.

Example 37. Given the set

$S = \{\neg T(c), \neg P(x) \vee \neg Q(f(x)), P(a), Q(x) \vee P(x), Q(f(b)) \vee \neg S(b), S(x) \vee \neg P(f(f(b))), P(b) \vee T(c)\}$

The following are examples of uniform interpretations:

- $\forall x : I(P(x)) = false, I(Q(x)) = true, I(\neg S(x)) = true, I(T(x)) = false$
- $\forall x : I(P(x)) = false, I(Q(x)) = false, I(S(x)) = false, I(T(x)) = false$
- $\forall x : I(P(x)) = true, I(Q(x)) = true, I(S(x)) = true, I(T(c)) = false, I(T(a)) = false, I(T(b)) = false, I(T(f(x))) = false$

And the following is an example of non-uniform interpretations:

- $\forall x : I(P(a)) = false, I(P(b)) = true, I(P(f(x))) = true I(Q(x)) = true, I(S(x)) = true, I(T(x)) = false$

  We have $I \models P(a)$ but $I \not\models P(b)$, both instances of $P(x)$.

If S is unsatisfiable, any uniform interpretation for S splits it into two non-empty sets S_1 (nucleii) and S_2 (electrons) as specified before. Our approach to define a language that splits the clauses of a set between start clauses and non-start clauses or *rules* is based on this idea. We introduce and formalise it in the next section.

7.2 Defining a language for the intermediate lemma extension

In this section we show how we can define a language that determines lemma and conclusion literals and we prove that if this language is based on an uniform interpretation for a set S, S is unsatisfiable iff a closed tableau can be constructed for S with the connection tableau calculus and the intermediate lemma extension. In other words, we prove that our specification is sound and complete. Moreover, in the next section we explain how this specification could be used to improve the performance of the prover.

Definition 37. (Language induced by an interpretation). Let S be a set of clauses, $\mathcal{L}$ the set of literals of the first-order language, and I an interpretation for atoms in S. The *language induced* by the interpretation I is the production field

$$\mathcal{P}_I = \langle \{L \in \mathcal{L} : I \models L\} \rangle$$

that is, the set of literals that are true under the interpretation I.

Proposition 9. *(Stability of languages induced by an uniform interpretation). If the interpretation I for S is uniform, the production field*

$$\mathcal{P}_I = \langle \{L \in \mathcal{L} : I \models L\} \rangle$$

is stable.

Proof. (Stability of languages induced by an uniform interpretation). Let I be an uniform interpretation for a set S, $\mathcal{P}_I = \langle \{L \in \mathcal{L} : I \models L\} \rangle$, and let C, D be clauses such that C subsumes D. We show that $D \in \mathcal{P}_I$ only if $C \in \mathcal{P}_I$.

Let $C = L_1 \vee \ldots \vee L_n$ and $D = K_1 \vee \ldots \vee K_m$ and assume $C \notin \mathcal{P}_I$. We have that for some $i, 1 \leq i \leq n$, $I \not\models L_i$. Since C subsumes D, there exists a substitution σ such that $C\sigma \subseteq D$, that is, $\{L_1\sigma, \ldots, L_n\sigma\} \subseteq \{K_1, \ldots, K_n\}$. Hence, for each $i = 1 \ldots n$ there is $j, 1 \leq j \leq m$ such that $L_i\sigma = K_j$. Because I is uniform, it assigns the same truth value to any instance of a literal L_i and then if $I \not\models L_i$ for some i, $I \not\models L_i\sigma = K_j$, which implies $D \notin \langle \{L \in \mathcal{L} : I \models L\} \rangle = \mathcal{P}_I$.

□

We can use any language induced by an uniform interpretation to determine which literals can be lemma or conclusion literals. For a set S and an uniform interpretation I for S, L is a lemma/conclusion literal if $L \in \mathcal{P}_I = \langle \{L \in \mathcal{L} : I \models L\} \rangle$. When we employ a specific language different from the positive literals, we refer to the intermediate lemma extension as *Intermediate Specific Lemma Extension*, to distinguish between these two variants of the connection tableau calculus.

Table 7.2 contains the formalisation of the rules for the Connection Tableau Calculus with the Intermediate Specific Lemma Extension. We add the language $\mathcal{P}_I$ to the tuples $C, S, Branch, Skip$. $\mathcal{P}_I$ is obviously not modified by any of the rules, and only used for conditions to be satisfied by the premises. Prior to application of the rules, in all clauses $C \in S$ with at least one literal $L \in \mathcal{P}_I$, one of those literals $L \in \mathcal{P}_I$ is chosen non-deterministically and underlined as conclusion literal.

The rules in the context of the graphical representation for a given set of clauses S, a language $\mathcal{P}_I$ based on an uniform interpretation for S and a marked tableau T for S are essentially the same as in Section 3.1. The only difference is the choice of start clauses, the choice of a subgoal in the extension rule and the condition to be satisfied by a literal for the skip rule to be applicable, to reflect the new language instead of positive literals. The definition is straightforward using the same conditions as in Table 7.2, so we do not detail it here.

$$\mathit{Start}\ (\mathit{St})\ \frac{\{\mathit{root}\}, S, \{\}, \{\}, \mathcal{P}_I}{C, S, \{\mathit{root}\}, \{\}, \mathcal{P}_I}$$

where $C \in S$ is such that $\forall L \in C, L \notin \mathcal{P}_I$

Extension (*Ext*)

$$\frac{C \cup \{K\}, S, \mathit{Branch}, \mathit{Skip}, \mathcal{P}_I}{(C_2 \backslash \{L_1\})\sigma, S, (\mathit{Branch} \cup \{K\})\sigma, (\mathit{Skip})\sigma, \mathcal{P}_I \ \ || \ \ C\sigma, S, (\mathit{Branch})\sigma, (\mathit{Skip})\sigma, \mathcal{P}_I}$$

where $K \notin \mathcal{P}_I$, C_2 is a renamed instance of $C_1 = \{\underline{L_1}, \ldots, L_n\} \in S$ and $\sigma(K) = \sigma(L_1)$

$$\mathit{Reduction}\ (\mathit{Red})\ \frac{C \cup \{K\}, S, \mathit{Branch} \cup \{L\}, \mathit{Skip}, \mathcal{P}_I}{C\sigma, S, (\mathit{Branch})\sigma \cup \{L\sigma\}, (\mathit{Skip})\sigma, \mathcal{P}_I} \quad \text{with } \sigma(K) = \sigma(\overline{L})$$

$$\mathit{Skip}\ (\mathit{Sk})\ \frac{C \cup \{K\}, S, \mathit{Branch}, \mathit{Skip}, \mathcal{P}_I}{C, S, \mathit{Branch}, \mathit{Skip} \cup \{K\}, \mathcal{P}_I} \quad \text{with } K \in \mathcal{P}_I$$

$$\mathit{Lemma\ \ derivation}\ (\mathit{Lem})\ \frac{\{\}, S, \mathit{Branch}, \mathit{Skip}, \mathcal{P}_I}{\{\mathit{root}\}, S \cup \mathit{Skip}, \{\}, \{\}, \mathcal{P}_I} \quad \text{with } \mathit{Skip} \neq \emptyset$$

Table 7.1: Rules of the Connection Tableau Calculus with the Intermediate Specific Lemma Extension

Proposition 10. *(Soundness and completeness for the Connection Tableau Calculus with the Intermediate Specific Lemma Extension). The Connection Tableau Calculus with the Intermediate Specific Lemma Extension is sound and complete.*

- *If a sequence of connection tableau $T_1, \ldots, T_n$ with T_n closed can be constructed from S by application of the rules of the calculus, then S is unsatisfiable.*
- *If S is unsatisfiable, then there exists a sequence of connection tableau $T_1, \ldots, T_n$ constructed using the rules of the calculus and such that T_n is closed.*

Proof. (Soundness and completeness for the Connection Tableau Calculus with the Intermediate Specific Lemma Extension).

To prove soundness and completeness for the calculus, we show that any proof found using the intermediate lemma extension with any specific language based on an uniform interpretation is isomorphic to a proof constructed with the language of all positive literals, and hence constructed with the normal intermediate lemma extension (for which soundness and completeness have been proven in Section 3.1).

Let S be a set of clauses, I an uniform interpretation for S and $\mathcal{P}_I = \langle\{L \in \mathcal{L} : I \models L\}\rangle$. The set S_p is constructed from S by renaming each predicate symbol P as P_p and then modifying the sign of each literal L that occurs in a clause in S as follows:

- If $L \in \mathcal{P}_I$ (and thus $I \models L$), the corresponding literal L' in S_p is positive.
- If $L \notin \mathcal{P}_I$ (and thus $I \models L$), the corresponding literal L' in S_p is negative.

If L is a conclusion literal in a clause ($L \in \mathcal{P}_I$) then the corresponding literal in S_p is positive and is chosen as conclusion literal.

For example, if $\mathcal{P}_I = \langle R(x), P(x), \neg S(x), Q(x), \neg T(x)\rangle$ and

$$S = \{\neg R(x), \neg P(x) \vee S(x), \underline{R(x)} \vee P(x) \vee \neg Q(x), \underline{\neg S(x)}, \underline{Q(x)} \vee \neg T(x), T(x)\}$$

where we have underlined the conclusion literals, we construct

$$S_p = \{\neg R_p(x), \neg P_p(x) \vee \neg S_p(x), \underline{R_p(x)} \vee P_p(x) \vee \neg Q_p(x), \underline{S_p(x)}, \underline{Q_p(x)} \vee T_p(x), \neg T_p(x)\}$$

Since I is uniform, all instances of a literal L are assigned the same truth value by I. We have a new interpretation for S_p that assign *true* to all atoms and by construction, $I \models C \in S$ iff $I_p \models C' \in S_p$, that is, it preserves the truth value of the transformed clauses. Notice that if $I \models L$, the corresponding literal L' is positive and $I_p \models L'$. If $I \not\models L$, then the corresponding literal L' is negative, $L' = \neg L''$, and $I_p \models L''$ implies $I_p \not\models \neg L'' = L'$.

We claim that S is unsatisfiable if and only if S_p is unsatisfiable.

$\Rightarrow$ Assume that S is unsatisfiable and suppose that S_p is satisfiable. Then there exists an interpretation I_p such that $I_p \models S_p$, that is, I_p must make true at least one literal in every clause $C_p \in S_p$. We can construct an interpretation I_s for S as follows: for each literal $K \in C_p$, if $I_p \models K$, I_s assigns *true* to the literal L in S from which K was obtained. If $I_p \not\models K$, then I_s assigns *false* to L.

Clearly I_s is well-defined in the sense that it does not assign *true* to $L_1 = L$ and $L_2 = \neg L$, since the literals K_1 and K_2 in S_p obtained from them have complimentary signs and then $I_s \models K_1$ iff $I_s \not\models K_2$. Furthermore, $I_s \models S$ because there is one literal K obtained from one literal L in a clause in S such that $I_p \models K$ in every clause in S_p. S would be satisfiable, which is a contradiction, so we conclude that S_p must be unsatisfiable.

$\Leftarrow$ Assume S_p unsatisfiable and suppose that S is satisfiable, so we have an interpretation $I_s \models S$. We use the same reasoning as in the previous part, to construct an interpretation for S_p from I_s. We reach a contradiction and can conclude that S must be unsatisfiable.

Now, to show soundness and completeness we use a similar argument, but instead of transforming the sets themselves, we transform the proofs found and in this way we can apply soundness and completeness of the original intermediate lemma extension.

- *Soundness:* Let S be a set of clauses for which a sequence of connection tableau $T_1, \ldots, T_n$ with T_n closed has been found, using the rules of the calculus with $\mathcal{P}_I$ and I an uniform interpretation for S. We apply the same transformation to obtain S_p from S to the sequence of tableaux, i.e., we replace each predicate symbol P with P_p in each node and each literal is signed as negative if the corresponding literal in S does not belong to $\mathcal{P}_I$ (therefore no skip operation has been applied to it in the tableau) or as positive if the corresponding literal in S does belong to $\mathcal{P}_I$ (and thus only reduction or skip operations have been applied to them in the sequence). The resulting sequence of tableau is one obtained by applying the rules of the classical connection tableau calculus with the intermediate lemma extension for S_p. By soundness of this calculus, we can conclude that S_p is unsatisfiable and hence S is unsatisfiable by the previous result.

- *Completeness:* Let S be an unsatisfiable set of clauses and $\mathcal{P}_I$ a language induced by an uniform interpretation I for S. S_p is unsatisfiable and by completeness of the connection tableau calculus with the intermediate lemma extension, we can construct a sequence of connection tableau $T_1, \ldots, T_n$ with T_n closed for S_p. We apply the inverse transformation to the sequence, that is, each predicate symbol P_p is replaced by P, each negative literal $\neg L$ is replaced by L if $I \models \neg L$ and left unmodified otherwise, and each positive literal L is replaced by $\neg L$ if $I \not\models L$ and not modified if $I \models L$. Then, the transformed sequence of tableaux only contains application of the skip rule to those literals that were positive in the original sequence and now are literals belonging to $\mathcal{P}_I$, start clauses are made of literals that prior to the transformation were negative and now do not belong to $\mathcal{P}_I$, as well as literals to which extension steps have been applied. The transformed sequence $T'_1, \ldots, T'_n$ with T'_n closed is a sequence of tableaux that can be constructed using the connection tableau calculus and the intermediate specific lemma extension with the language $\mathcal{P}_I$ for lemma and conclusion literals.

□

Example 38. To illustrate our proof, we show in Figure 7.4 a derivation of a closed tableau for the set

$$S = \{\neg R(x), \neg P(x) \vee S(x), \underline{R(x)} \vee P(x) \vee \neg Q(x), \underline{\neg S(x)}, \underline{Q(x)} \vee \neg T(x), T(x)\}$$

with $\mathcal{P}_I = \langle R(x), P(x), \neg S(x), Q(x), \neg T(x)\rangle$ and in Figure 7.5 the transformed derivation for the set S_p, with $\langle \mathcal{L}^+\rangle$ (the original intermediate lemma extension).

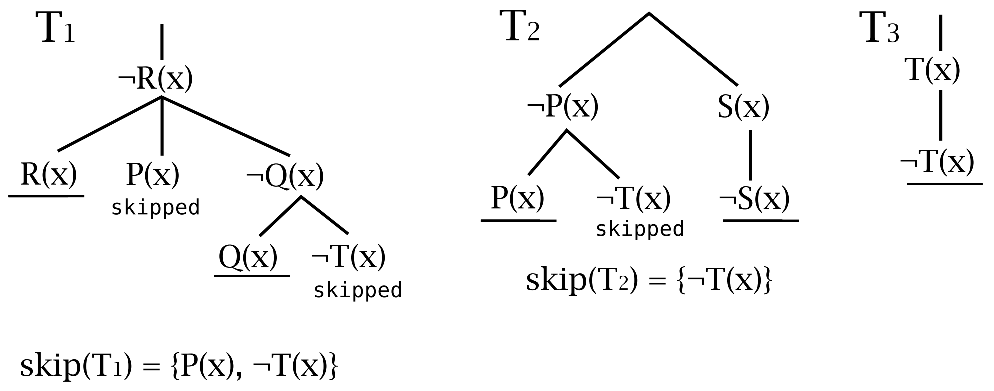


Figure 7.4: Proof for S using the intermediate specific lemma extension

7.3 Automatically generated languages

The previous generalisation of the intermediate lemma extension would be more useful in practise if we had an automatic method to generate an uniform interpretation for a set that, to some extent, helps to reduce the length of the proof or the size of the search space when constructing the proof. We

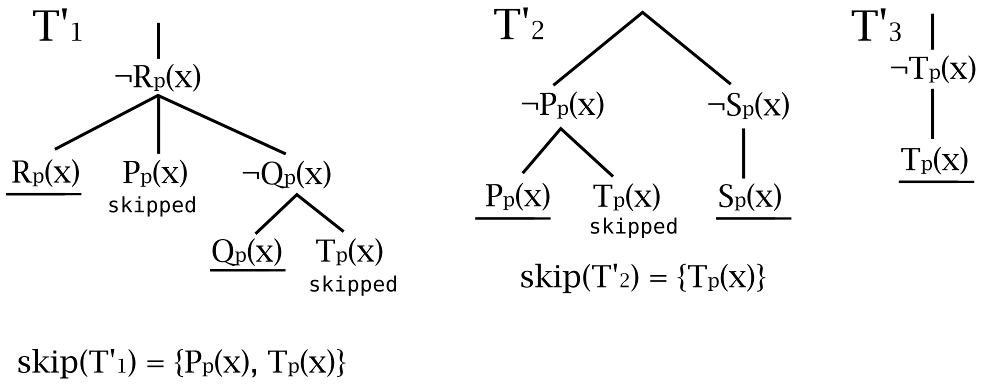


Figure 7.5: Corresponding proof for S_p using the classical intermediate lemma extension

have attempted to design a simple heuristic to choose for each predicate symbol in the set whether positive or negative literals are allowed as lemma/conclusion literals. We explain the details of such heuristic in this section and suggest a further refinement for it, discussing its relevance with statistics obtained from the TPTP Library.

The simplest language $\langle Lit \rangle$ based on an uniform interpretation that considers each predicate symbol separately rather than include just all positive/negative literals is one in which all literals in the set *Lit* are maximally general (see Section 3.2.1), i.e., their arguments are all distinct variables. Then, in a first-order language with predicate symbols $P_1, \ldots, P_n$ of arities $m_1, \ldots, m_n$, the possible languages are defined by

$$\langle \{[\neg]P_1(X_1, \ldots, X_{m_1}), \ldots, [\neg]P_n((X_1, \ldots, X_{m_n}))\} \rangle$$

where $[\neg]P_i(X_1, \ldots, X_{m_i})$ denotes that we may or may not include the negative literals with predicate symbol P_i. If we limit ourselves to such set of possible stable languages based on an uniform interpretation, the only choice left is the sign of each predicate symbol and we aim to select it automatically according to the particular characteristics of the input set.

We recall that the motivation of extending the connection tableau calculus with the intermediate lemma extension is to enable an inference mechanism similar to the SLD-resolution used in Prolog. In this mechanism, each clause is allowed to have an only conclusion literal, rather than the classical approach of the calculus in which an extension step can be made with any of the matching literals in a clause. This helps to reduce the search space in the sense that instead of a complete closed tableau, a set of more shallow and smaller tableaux is generated. Because of the lower depth, the search space for them is usually more reduced. However, if the configuration of the initial set of clauses makes it possible to generate many lemmas, and these lemmas are long clauses, the search space can be dramatically bigger than in the general case. Let us present the following *boundary* example.

Example 39. Consider the set of clauses with only one predicate symbol P

$$\begin{gathered} S = \{P(a,b,c), P(x,e,x), P(e,x,x), P(x,x,e), \neg P(b,a,c), \\ \neg P(u,v,y) \vee \neg P(x,u,z) \vee \neg P(z,v,w) \vee P(x,y,w), \\ \neg P(u,v,y) \vee \neg P(x,u,z) \vee \neg P(x,y,w) \vee P(z,v,w)\} \end{gathered}$$

We can observe that S is made of Horn clauses, with only one possible start clause $\neg P(b,a,c)$. If we applied the connection tableau calculus with the intermediate lemma extension to this set, no lemma would be generated because only one conclusion literal is selected from each non-start clause and no lemma literals left. Only one closed tableau is found and our prover (the simplest version) is able to construct one of depth 5 in 0.15 seconds.

Suppose now that we were to change the language used to select lemma and conclusion literal. The only alternative is to use $\langle \mathcal{L}^- \rangle = \langle \{\neg P(X,Y,Z)\} \rangle$. The situation is completely different. Now we have 4 possible start clauses and a choice of 3 possible conclusion literals in each one of the two non-unit clauses. Whenever we use one of those clauses for an extension step, 2 literals can be potentially skipped and part of a new derived lemma and actually, when searching for a proof in our prover, lemmas of up to length 5 are generated in the process and the prover is not able to find a proof in less than 10 minutes.

We have categorised the previous example as *boundary* because it contained only Horn clauses and in practise the majority of problems contain non-Horn clauses. In version 4.0.1. of the TPTP Library ([51], Section 8.1), 4076 of the 6800 CNF problems are non-Horn problems. However, in average only 20% of the clauses in the 4076 non-Horn problems are in fact non-Horn clauses, as indicated in the CNF synopsis of the TPTP documentation. In light of these statistics and considering the previous example, it is worth to think of a method or heuristic to avoid situations like the above. To this end, to decide if we use positive or negative literals as lemma/conclusion literals for each predicate symbol in order to construct the set *Lit* in $P_I = \langle Lit \rangle$ we use the following procedure.

For each predicate symbol P_i of arity m_i in the first-order language used in S:

1. Compute the tuple (N_p, N_n, D_p, D_n) where:
   - N_p is the total number of occurrences of P_i in a positive literal in S.
   - N_n is the total number of occurrences of P_i in a negative literal in S.
   - D_p is the total number of clauses in S that contains P_i occurring in a positive literal.
   - D_n is the total number of clauses in S that contains P_i occurring in a negative literal.

   In the set of the previous example, the only predicate symbol is P of arity 3 and the corresponding tuple is $(6, 7, 6, 3)$, since P appears in 6 positive literals, all of them in different clauses, and in 7 negative literals, only 3 of them in different clauses.

2. If $(N_n - D_n) < (N_p - D_p)$ then add $\neg P(X_1, \ldots, X_{m_i})$ to *Lit*
   (i.e., assign $I(P(X_1, \ldots, X_{m_i})) = false$ for all X_j, $1 \leq j \leq m_i$).

3. If $(N_p - D_p) < (N_n - D_n)$ then add $P(X_1, \ldots, X_{m_i})$ to *Lit*
   (i.e., assign $I(P(X_1, \ldots, X_{m_i})) = true$ for all X_j, $1 \leq j \leq m_i$).

4. If $(N_n - D_n) = (N_p - D_p)$, then
   - If $N_n < N_p$ then add $\neg P(X_1, \ldots, X_{m_i})$ to *Lit*.
   - Otherwise add $P(X_1, \ldots, X_{m_i})$ to *Lit*.

   In the set of the previous example, as we have $6 - 6 < 7 - 3$, the language resultant would have been $\langle \{P(X,Y,Z)\} \rangle$

The previous heuristic favours the literals with fewer appearances in the same clause, trying to avoid to have a clause with potentially many skipped literals and to approximate to the setting with only Horn clauses.

7.3.1 Further refinement

When we introduced production fields for the SOL tableau calculus in Section 3.2.1 we pointed out that in general, if a production field contains a literal that is not maximally general, then it is not stable. Additionally but without giving details, we mentioned that if we restrict the definition of stability to a concrete set of clauses instead of considering an arbitrary set in the context of a first-order language, a production field with a not maximally general literal could be stable. Let us clarify this with an example.

Example 40. Consider the set of clauses

$$S = \{Q(a,x) \vee P(b) \vee \neg P(f(x)), \neg P(b), \neg P(a,y), P(f(c))\}$$

Regarding to atoms, $P(b)$ and $P(f(c))$ are ground, $P(f(x))$ can only be unified with $P(f(c))$ and $Q(a,x)$ can only be unified with $P(a,y)$, so the second variable does not get instantiated. We could replace $Q(a,x)$ and $P(b)$ by other predicate symbols like $Q(x)$ and $R(b)$ without affecting the unsatisfiability of the set or the proof found. The key is that $P(f(x))$ and $P(f(c))$ do not have *common instances* with $Q(a,x)$ and $P(b)$, that also do not have common instances between each other.

If we consider exclusively atoms and clauses contained in S to be part of our first-order language and then we define that a production field $\mathcal{P}$ is stable if for all clauses $C, D \in S$ such that C subsumes D, $D \in \mathcal{P}$ only if $C \in \mathcal{P}$, the following production field would be stable: $\mathcal{P} = \{Q(a,x), \neg P(b), P(f(x))\}$. Any clauses C and D in S can contain $\neg P(b)$ and different instances of $P(f(x))$ and $Q(a,x)$ (with different variables or x/c in the case of $P(f(x))$), so if C subsumes D and $D \in \mathcal{P}$, C would also satisfy the condition imposed by $\mathcal{P}$. Moreover, if we restrict in a analogous manner the definition of interpretation to assign a value to atoms in S rather than in the whole Herbrand base, the interpretation given by $\forall x : I(Q(a,x)) = I(P(f(x))) = \mathit{true}$, $I(P(b)) = \mathit{false}$ is uniform. Below we formalise the new definitions.

Definition 38. (Stable production field with respect to a set). Given a set of first-order formulas S, if for any clauses $C, D \in S$ such that C subsumes D, $D \in \mathcal{P}$ only if $C \in \mathcal{P}$, the production field $\mathcal{P}$ is said to be *stable* with respect to S.

Definition 39. (Uniform interpretation with respect to a set). Given a set of first-order formulas S, let $\mathit{atoms}(S)$ be the set of atoms occurring in S and $\mathit{lit}(S)$ the set of literals that can be constructed with $\mathit{atoms}(S)$. An interpretation I with domain $\mathit{atoms}(S)$ is an *uniform interpretation* if for any literal $L \in \mathit{lit}(S)$ and for any substitution σ such that $L\sigma \in \mathit{lit}(S)$, $I \models L$ iff $I \models L\sigma$. In other words, if I assigns to any instances of L that are in $\mathit{lit}(S)$ the same truth value as L.

Since truth values for literals are preserved after substitutions within $\mathit{lit}(S)$, so are for clauses in S.

If we had generated automatically a language for S using the procedure described previously, we would have considered two predicate symbols, P and Q, and after computing the tuples ($(2,2,2,2)$ for P and $(1,1,1,1)$ for Q) we would have chosen the language $\langle Q(x,y), P(x)\rangle$. However, we can further refine the language considering the new definition of stability and interpretation and search for the *most specific* language based on an uniform interpretation, instead of just contemplating literals that are maximally general. In this case, treating $P(b)$, $P(f(x))$ and $Q(a,x)$ separately, the tuples (N_p, N_n, D_p, D_n) are computed according to the *instances* of them that we find in the clauses in S, which in this case are $(1,1,1,1)$ for all of them, and the language $\langle P(b), P(f(x)), Q(a,x)\rangle$ is chosen. Next we formalise the concept of most specific language based on an uniform interpretation, that has been introduced informally. Rather than using the words *most specific* to refer to this language, we prefer to name it as *least general* to contrast with the most general languages ($\langle \mathcal{L}^+\rangle$ or $\langle \mathcal{L}^-\rangle$) based on uniform interpretations.

Definition 40. (Least general language for a set of clauses based on an uniform interpretation). Given a set of clauses S and an uniform interpretation I with respect to S,

$$\mathcal{G}_I = \langle\{[\neg]A_1, \ldots, [\neg]A_n\}\rangle$$

(where $A_1, \ldots, A_n \in \mathit{atoms}(S)$ and $[\neg]A_i$ denotes A_i or its complement) is a least general language for S if and only if:

- For every i, j, $1 \leq i < j \leq n$, A_i and A_j are not unifiable, i.e., there does not exist any substitution σ such that $A_i\sigma = A_j\sigma$, so they do not have common instances.

- For every literal $L \in lit(S)$, there is $i, 1 \leq i \leq n$ such that A_i or $\neg A_i$ subsumes L.

This further refinement allows to achieve a higher degree of freedom when defining the language. However, the procedure of computing the least general set of atoms for a set can be expensive in terms of time and it is reasonable to think that in a significative number of cases, the least general language is one in which all literals are maximally general, so we would have achieved the same result applying our first procedure. We have analysed the TPTP Library in order to know how often this actually happens. For each problem of the TPTP Library we have computed one of the least general languages (the one with all positive literals) and have checked if it is actually less general than the language made entirely of maximally general literals. For CNF problems, only 121 out of 6800 problems have a least general language with not only maximally general literals. For FOF problems, we have to convert them to clauses first. We have used a timeout of 20 seconds for the clausal transformation (explained in Section 5.1.2), since for some problems in the TPTP Library the amount of time needed would make this kind of statistical analysis unfeasible. With this timeout of 20 seconds we have generated the clausal transformation for 5352 out of 6983 problems. Of 5352, only 257 problems have a least general language with not only maximally general literals. The proportion is then 5% for FOF and only 2% for CNF problems. These results justify our decision of saving the computation time needed to find a least general language and use directly the procedure described in the previous section.

7.4 When non-uniform interpretations work

We have included this section just for interest, as a very good example to illustrate everything that have been exposed in this chapter. Sometimes it is possible to find a set of clauses with instances that can be split by a non-uniform interpretation in a manner that still do not cause any contradictory truth value assignment, because the instances that would cause such a problem do not occur in any unsatisfiable subset of the given set. For example, suppose we have different instances of clauses of the form $P(x) \vee Q(x) \vee \neg R(x)$ and $P(x) \vee Q(x) \vee R(x)$ in the given set and our interpretation I assigns $true$ to $R(x)$ and $false$ to $Q(x)$ for all x. We may want to assign $P(x)$ false only in those clauses in which $\neg R(x)$ is present and if this is so, this clauses would be Horn clauses according to $\mathcal{P}_I$.

There is a famous problem called "The Steamroller" set up by L. Schubert of the University of Alberta in 1978 [55]. Despite its apparent simplicity, this problem became well-known because it turned out to be too hard for existing theorem provers due to the size of the search space. The problem is enunciated as follows:

> "Wolves, foxes, birds, caterpillars and snails are animals, and there are some of each of them. Also, there are some grains, which are plants. Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves. Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Therefore, there is an animal that likes to eat a grain-eating animal."

For Schubert's problem a partition through a non-uniform interpretation of the above kind is not only possible, but also turns it from a problem that many theorem provers cannot solve into a set of Horn clauses that can be solved instantly by SLD-resolution (Prolog).

Using the following predicates as abbreviations

$A(x)$: x is an animal	$W(x)$: x is a wolf
$F(x)$: x is a fox	$B(x)$: x is a bird
$C(x)$: x is a caterpillar	$S(x)$: x is a snail
$G(x)$: x is a grain	$P(x)$: x is a plant
$M(x,y)$: x is much smaller than y	$E(x,y)$: x likes to eat y

yields to the clausal formulation (we have numbered the clauses):

$$\begin{gathered}
S = \{\ 1.\ W(w),\ \ 2.\ F(f),\ \ 3.\ B(b),\ \ 4.\ C(c)\ \ 5.\ S(s),\ \ 6.\ G(g),\\
7.\ \neg W(x) \vee A(x),\ \ 8.\ \neg F(x) \vee A(x),\ \ 9.\ \neg B(x) \vee A(x),\\
10.\ \neg C(x) \vee A(x),\ \ 11.\ \neg S(x) \vee A(x),\ \ 12.\ \neg G(x) \vee P(x),\\
13.\ \neg C(x) \vee \neg B(y) \vee M(x,y),\ \ 14.\ \neg B(x) \vee \neg F(y) \vee M(x,y),\\
15.\ \neg S(x) \vee \neg B(y) \vee M(x,y),\ \ 16.\ \neg F(x) \vee \neg W(y) \vee M(x,y),\\
17.\ \neg W(x) \vee \neg F(y) \vee \neg E(x,y),\ \ 18.\ \neg W(x) \vee \neg G(y) \vee \neg E(x,y),\ \ 19.\ \neg B(x) \vee \neg S(y) \vee \neg E(x,y),\\
20.\ \neg B(x) \vee \neg C(y) \vee E(x,y),\\
21.\ \neg C(x) \vee E(x,h(x)),\ \ 22.\ \neg S(x) \vee E(x,i(x)),\\
23.\ \neg C(x) \vee P(h(x)),\ \ 24.\ \neg S(x) \vee P(i(x)),\\
25.\ \neg A(x_1) \vee \neg P(y_1) \vee \neg A(x_2) \vee \neg P(y_2) \vee \neg M(x_2,x_1) \vee \neg E(x_2,y_2) \vee E(x_1,y_1) \vee E(x_1,x_2),\\
26.\ \neg A(x_1) \vee \neg A(x_2) \vee \neg G(y) \vee \neg E(x_1,x_2) \vee \neg E(x_2,y)\}
\end{gathered}$$

Clause 25 states that "animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants" and clause 26 is the negation of the goal.

An analysis of the size of the search space of this problem is provided in [55]. In summary, the difficulties of an automated theorem prover in computing a solution for it are justified by:

- The size of the initial search space (102 distinct clauses can be computed as resolvents or factors in the first generation).
- Schubert's hand-computed solution needs a search depth of 20 to derive the empty clause. This leads to a search space that grows so rapidly that exceeds space and time boundaries of some automated theorem provers, before the empty clause can be found. This is true even using a *set of support* strategy [2] that reduces the initial search space to 28 resolvents and 2 factors.

We have tried to solve this problem with the basic version of our prover and it is able to find a proof at depth 7 in 3 minutes and 6 seconds.

Now, consider the following language (informally specified) for the lemma/conclusion literals:

$$\begin{gathered}
\mathcal{P} = \langle \mathcal{L}^{+} \backslash \{E(x,y)\}\ \cup\ \{E(x,y) \in C : \mathit{Animal}(x) \cap C = \emptyset \text{ or } \mathit{Animal}(y) \cap C = \emptyset\} \cup\\
\{\neg E(x,y) \in C : \mathit{Animal}(x) \cap C \neq \emptyset \text{ and } \mathit{Animal}(y) \cap C \neq \emptyset\}\rangle
\end{gathered}$$

[2]This is a restriction in which resolution steps are allowed only if at least one of the clauses involved in the step is from the set of support (SOS) or derived from such a clause.

where $Animal(z) = \{\neg A(z), \neg W(z), \neg F(z), \neg B(z), \neg C(z), \neg S(z)\}$

That is, all positive literals are considered to be lemma/conclusion literals, except for literals with the predicate symbol E, that are divided in two groups: $\neg E(x, y)$ is a lemma/conclusion literal in a clause C if both x and y are animals in the body of C and $E(x, y)$ is considered a lemma/conclusion literal in C at least one of them is not an animal in the body C. Being an animal, that is, $A(x)$ holds, if $W(x)$, $F(x)$, $B(x)$, $C(x)$ or $S(x)$ holds. An interpretation that partitions the literals like this is clearly not uniform and yet it does not prevent from finding a proof. Let us show how it can be achieved as $\mathcal{P}$ turns S into a set of Horn clauses.

With $\mathcal{P}$, clauses 1-6 are Horn clauses (facts) and so are clauses 7-16 and 23-24, since only one literal in each one satisfies the conditions in $\mathcal{P}$. Clause 17 is also a Horn clause, since x and y are both animals (a wolf and a fox, respectively), so $\neg E(x, y) \in \mathcal{P}$ and it is chosen as conclusion literal. The same happens with clause 19. Clauses 21 and 22 are also Horn clauses as only x is an animal and so $E(x, h(x)), E(x, i(x)) \in \mathcal{P}$.

Clauses 18 and 20 are start clauses (in 18 $\neg E(x, y) \notin \mathcal{P}$ because y is not an animal and in 20 $E(x, y) \notin \mathcal{P}$ because both x, y are).

Finally, in 25 and 26 we have two occurrences of a literal with $E(x, y)$. In 26 x_1 and x_2 are the only animals, so $\neg E(x_1, x_2)$ is the unique choice for a conclusion literal. In 25, x_1, x_2 are animals but y_1 and y_2 are not. This means that we have to choose $E(x_1, y_1)$ as the only possible conclusion literal leaving the Horn clause:

$$\underline{E(x_1, y_1)} \vee \neg A(x_1) \vee \neg P(y_1) \vee \neg A(x_2) \vee \neg P(y_2) \vee \neg M(x_2, x_1) \vee \neg E(x_2, y_2) \vee E(x_1, x_2)$$

In order to find a solution with our normal theorem prover, we rename the literals $E(x, y)$ in which x and y are animals to $E_p(x, y)$ inverting the sign and leave $E(x, y)$ the same in the remaining occurrences. With the renamed clauses, the same transformation to Horn clauses as above is achieved by allowing all positive literals as lemma/conclusion literals (this is the actually the method that we used in our soundness and completeness proof for this extension of the calculus). Our basic prover can find now a proof at depth 6 in only 0.18 seconds.

It is worth mentioning that the prover based on the automatically generated language with the heuristic described before is able to find a (very long) solution at depth 10 in just 1.68 seconds. The language computed is

$$\mathcal{P} = \langle \mathcal{L}^+ \backslash \{M(x, y)\}\} \cup \{\neg M(x, y)\}\rangle$$

That is, all positive literals are lemma/conclusion literals except for those with the predicate symbol M, which have to be negative to be eligible as lemma/conclusion literals. With this language, clauses 25 and 26 are the only non-Horn clauses ($\neg M(x_2, x_1)$ is chosen as conclusion in 25) and there are 7 possible start clauses, 13-19.

The Steamroller problem is included in the TPTP Library [51] in problem `PUZ031-1`, in the domain of *Puzzles* (see next chapter for more details on the features of TPTP Library problems and domains). In order to have a good reference of the effectiveness of the specific lemmas extension, we have attempted to solve this problem with some well-known theorem provers, using the online solving interface System On TPTP [47] and a time limit of 300 seconds. The results are presented in Table 7.4.

A number of tests based on the TPTP Library have been performed also for this extension of the prover. All results are collected in the next chapter, where the global performance of the prover is evaluated and this version compared against the others.

System	Output	System	Output
Bliksem 1.12	Unsatisfiable (0.00 sec.)	Mace2 2.2	Timeout (301.55 sec.)
Darwin 1.4.5	Unsatisfiable (0.01 sec.)	Mace4 0908	Timeout (300.27 sec.)
DCTP 1.31	Unsatisfiable (0.01 sec.)	Otter 3.3	Unsatisfiable (0.27 sec.)
E 1.2	Unsatisfiable (0.02 sec.)	Paradox 4.0	Timeout (300.40 sec.)
E-Darwin 1.3	Unsatisfiable (0.02 sec.)	Prover9 0908	Unsatisfiable (0.02 sec.)
iProver 0.8	Unsatisfiable (0.04 sec.)	S-SETHEO 0.0	Unsatisfiable (0.01 sec.)
iProver SAT 0.8	Unsatisfiable (0.11 sec.)	SETHEO 3.3	Unsatisfiable (0.04 sec.)
IsabelleM 2009-2	Inappropriate (-)	Theo 2006	Unsatisfiable (0.05 sec.)
IsabelleN 2009-2	Inappropriate (-)	Vampire 0.6	Unsatisfiable (0.02 sec.)
IsabelleP 2009-2	Inappropriate (-)	Vampire 9.0	Inappropriate (-)
leanCoP 2.2	Unsatisfiable (4.46 sec.)	Vampire SUMO	Inappropriate (-)
LeanTAP 2.3	Error (-)	Waldmeister 710	Inappropriate (-)

Table 7.2: Solving the Steamroller problem with different automated theorem provers

7.5 Implementation

To finish this chapter, after having settled the theoretical basis of *specific lemmas* and studied an interesting problem, we will describe briefly the minor modifications needed in our code to incorporate the new feature as a version of the basic prover. Once we have decided how to compute and store the corresponding language $\mathcal{P}_I$, we need to replace the distinction between negative and positive literals by a predicate that tests whether a literal L belongs to $\mathcal{P}_I$.

First, to store $\mathcal{P}_I$ we have opted for a fact in the database of the form:

`syntax_lemmas([·p1(_,...,_),·p2(_,...,_),...,·pn(_,...,_)]).`

where `p1,...,pn` correspond to all predicate symbols $P_1,\dots,P_n$ occurring in S and $\cdot$ can be either `+` or `-`, depending on the result of the calculation described in Section 7.3 and implemented by the predicate `construct_syntax/2`, not detailed here. For example, the language $\mathcal{P}_I = \langle R(x), P(x), \neg S(x), Q(x), \neg T(x)\rangle$ would be stored as `syntax_lemmas([+r(_),+p(_),-s(_),+q(_),-t(_)]).`

Furthermore, we have implemented a predicate `is_uniform(+Set)` that checks if the current language stored in the database is uniform with respect to the set passed as parameter. In this way, we allow the user to define their own language for lemma and conclusion literals.

In order to check if a given literal *Lit* belongs to $\mathcal{P}_I = \langle \mathit{Lang}\rangle$, it is enough to find a maximally general literal $K \in \mathit{Lang}$ such that K subsumes *Lit*. This is basically what the predicate `matches(+Lit)` does. The code of the core predicate `prove/9` in the basic prover is simply modified as follows:

```
prove([Lit|Clause],Branch,...,Skip,...) :-
      \+ matches(Lit), (*1)
      comp(Lit,CompLit), (*2)
            ...
            cla([CompLit],B,L), (*3)
            prove(B,[Lit|Branch],...,Skip1,...),
          ...
      prove(Clause,Branch,...,Skip2,...),
      combine_lemma_literals(Skip1,Skip2,Skip),
      ...
```

In line (*1) it is tested whether `Lit` is a lemma/conclusion literal and if not, an extension step is attempted. The predicate `comp(+Lit,-CompLit)` in line (*2) returns the complimentary literal of `Lit`, that is used to index the database in line (*3) in order to find a suitable clause. The rest of the predicate is essentially the same as in the basic version.

```
prove([Lit|Clause],Branch,...,Skip,...) :-
      matches(Lit), (*1)
      (comp(Lit,CompLit), (*2)
      member(CompLit,Branch), (*3)
      prove(Clause,Branch,...,Skip,...),
      ;
      prove(Clause,Branch,...,Skip1,...)),
      combine_lemma_literals([Lit],Skip1,Skip),
      ...
```

Similarly to the previous case for non-lemma/conclusion literals, line (*1) verifies that `Lit` belongs to the language. The complimentary literal computed in line (*2) is used to explore the current branch in line (*3) searching for a candidate for a reduction step. If this fails, the skip rule is applied as in the basic prover.

Regarding the implementation of the further refinement described at the end of last section, the least general language for a set of clauses S,

$$\mathcal{G}_I = \langle\{[\neg]A_1, \ldots, [\neg]A_n\}\rangle$$

(where $A_1, \ldots, A_n \in atoms(S)$ and $[\neg]A_i$ denotes A_i or its complement) have been computed using the following algorithm:

Input: A set of clauses S

Output: A set $\{A_1, \ldots, A_n\}$ of unsigned atoms from $atoms(S)$ such that $\mathcal{G}_I = \langle\{[\neg]A_1, \ldots, [\neg]A_n\}\rangle$ is a least general language for S.

1. $Language = \{\}$
2. Let L be a set of tuples $(P, A, Literals)$, where P is a predicate symbol of arity A that appears in S and $Literals$ is a set with all *unsigned* literals in S in which P occurs. For example, for

   $$S = \{Q(a,b) \vee \neg P(x), P(a) \vee \neg R(y), \neg Q(z,c), R(a)\}$$

   we would have

   $$L = \{(Q, 2, \{Q(a,b), Q(z,c)\}), (P, 1, \{P(x), P(a)\}), (R, 1, \{R(y), R(a)\})\}$$

```
3. For each tuple $(P, A, Literals) \in L$ do:
   (a) $Current = Literals, Previous = \{\}$
   (b) While $Current \neq Previous$ :
           Delete all literals $L \in Current$ that are subsumed by $K \in Current$, $K \neq L$.
           $Previous = Current$
           Let $L$ be a literal in $Current$, $Generalised = \{L\}$
           For each $K \in Current$, $K \neq L$ do:
             • If $K$ and $L$ are not unifiable, $Generalised = Generalised \cup \{K\}$ (they do not
               have common instances).
             • Else let $G$ be the most specific generalisation of $K$ and $L$,
               $Generalised = Generalised \cup \{G\}$
           $Current = Generalised$
   (c) $Language = Language \cup Current$
4. Return $Language$
```

This algorithm has been implemented in a predicate `least_general_language(+S,-L)`.

Chapter 8

Performance of the prover

This chapter is dedicated to the evaluation and testing of the different versions of our prover. This allows us to assess the capabilities of the system with respect to other theorem provers and the performance effect of the pruning methods and equality built-in techniques developed through this report.

The evaluation has been carried out using The TPTP Problem Library v4.0.1. for Automated Theorem Proving [51]. Its characteristics are described in Section 8.1, together with some aspects that we have considered when employing this Library to test our system. In Section 8.2 we provide a collection of the tests results obtained, as well as a brief analysis of them. Finally, in Section 8.3 a small comparison against other different automatic theorem proving systems is presented.

All tests have been performed in a 2600 MHz Six-Core AMD Opteron^{TM}with 8GB of RAM running GNU/Linux (Kernel version 2.6.24-19-generic) and SWI-Prolog [3]. Since not all the tests have been carried out in the same conditions, more details about execution and time limits are provided along with the results.

8.1 Using the TPTP Library

The TPTP (Thousands of Problems for Theorem Provers) is "the facto standard set of test problems for classical first-order Automated Theorem Proving (ATP) systems" [51]. It provides a comprehensive library of test problems in classical logic with equality used as a simple and unambiguous reference mechanism for evaluation of different ATP systems. One of the decisive advantages and reasons of its success is the consistent use and adoption of the TPTP language for problems and solutions. Apart from the extensive collection of problems, the TPTP Library supplies an utility, the `tptpt2X` tool, to convert the problems from the TPTP format to formats used by existing ATP systems and to apply transformations such as conversion into clausal form or addition of the equality axioms among others.

The problems in the TPTP Library are classified into *domains* (reflecting scientific domains) and are presented divided into 6 fields: logic, mathematics, computer science, science and engineering, social sciences, and other. Each domain is identified by a three-letter mnemonic. The full classification scheme as well as a brief description of each domain can be found in the TPTP technical manual [48]. Each physical file stores a *version* of an underlying *abstract* problem, represented by the collection of corresponding problem versions. In the TPTP, the main differentiation between problem versions is whether the problem is presented in first-order clausal form (CNF), as a set of first-order logic formulas (FOF), typed-higher order form (THF) or typed first-order form (TFF). For the nature of our theorem prover, the tests are restricted to the first two types, CNF and FOF, and we will refer only to them in the rest of this section. Annotations in the header differentiates a problem file from other versions. Axiomatisations are kept in separate axioms files and are used by several problem versions. The use of different axiomatisations is the main reason for having different versions of the same abstract problem.

The release v4.0.1. of the TPTP used for the benchmarks contains 10076 abstract problems, which result in 16512 ATP problems. There are 4297 FOF abstract problems, which result in 6983 FOF ATP problems, and 5047 CNF abstract problems, which result in 6800 CNF ATP problems. The naming scheme developed for the files of the TPTP Library reflects the division in domains, types and versions. It provides an unambiguous denomination for each abstract problem, problem version and axiomatisation.

The difficulty of the problems in the TPTP Library ranges from very simple problems to open problems. One of the most important features of the library is the *rating* scheme [49], that aims to measure the difficulty of the problems and the effectiveness of ATP systems with respect to different types of problems. For rating purposes, problems are divided in *specialist problem clases* (SPCs), which are syntactically identifiable groups of problems for which some ATP techniques have been proved to be well suited, for example, "CNF unsatisfiable unit equality problems". For each SPC, performance data of contemporary theorem proving systems is analysed and a partial order established between them according whether a system solves a strict superset of the problems solved by another system. If this happens, the first system is said to *subsume* the second system. Problems solved by all non-subsumed systems get a rating of 0.0 (*easy* problems) and unsolved problems get a rating of 1.0 (*very difficult*). Problems that are solved by just some of the non-subsumed systems get a rating between 0.0 and 1.0 and are considered to be *difficult*.

TPTP problems and solutions are written in the TPTP Language [15], introduced in release v.3.0.0. It has a Prolog-like syntax, to facilitate the development of reasoning software for TPTP data in Prolog. As we have mentioned before, the `tptp2X` tool includes a functionality to translate problems in the TPTP Language to any other format specified by output formatting files `format.<ATP>` implemented in Prolog. The current version of the `tptp2X` tool comes along with 41 formatting files including some well-known theorem provers such as Otter [26], Isabelle [35], Setheo [36] or leanCoP [31] (see also Section 2.3.4).

It is possible to add new output formats by creating new format files and it is advised in the TPTP technical manual to do that by modifying a copy of one the existing files, to guarantee that all files follow the same structure. We have written our own formatting file, named `format.ilemma`, to translate TPTP files to the problem file format described in Section A.1 for input files of our prover. Since the format is essentially the same as leanCoP's, we have written it by modifying `format.leancop`. The main changes have been switching from a positive to a negative representation, translating the equality symbol as = instead of a normal predicate `eq/2` [1], adding support for the new connectives ~`|` (nor), ~`&` (nand) and `<~>` (xor) and fixing some minor issues for the v.4.0.1 version. This formatting file is used to translate problems in the TPTP in order to evaluate our system. The results of such evaluation are detailed in the next section.

8.2 Benchmark results

In order to run the tests in our prover, CNF and FOF problems in the TPTP Library have been translated using the formatting file `format.ilemma` mentioned in the previous section. We have excluded 7 FOF problem files for which the `tptp2X` tool failed to generate the proper format due to their huge size. Then, for the benchmarks 6976 FOF problems and 6800 CNF problems (13776 in

[1] leanCoP employs this notation because they make use of the `tptp2X` tool to add the equality axioms to the problem file, whereas we add them as part of our input preprocessing (Section 5.1)

total) are considered.

To ensure the accuracy of the results obtained, we follow the use guidelines of the TPTP Library and provide the following information:

- The TPTP release used is v.4.0.1. At the time of writing this report, a new version v.4.1.0. released on 15th June 2010 is available, but our development was started before that date.
- The only change that we make to the problem formulas is the transformation to clausal form when required. No re-ordering, removal, reformatting or addition is carried out.
- No information other than the set of formulas is provided to the system neither the header information is used during the tests.

When running this kind of tests, we have to decide which time limit is set for the prover to give up with a concrete problem. The TSTP (Thousands of Solutions from Theorem Provers) solution library [52, 50], a corpus of solutions to TPTP problems, contain the results of running 44 ATP systems and some of their variants on all the problems in the TPTP Library. The TSTP Library is regularly updated with the most recent versions of those ATP systems and it can be used not only to study how particular problems can be attacked in order to improve a particular system, but also for comparisons and relative evaluation of a concrete system. In 2009, after the release of TPTP v.3.5.0. in July, all the runs were done on 2.80 GHz computers with 1 GB of RAM and running Linux as operating system, with a 600 seconds CPU limit. It is not unusual to choose similar system settings and time limit when performing benchmarks for a theorem prover, in order to increase the accuracy of the results. This is the case, for example, of the performance results of leanCoP presented in [32], obtained with a time limit of 600 seconds on a 3 GHz Xeon system with 4 GB of RAM. However, due to the time restrictions of this project and the limited availability of resources, using such settings in order to evaluate our four different system versions over 13776 distinct problems was unfeasible. We decided to restrict the time limit and the machine capabilities by running 6 tests in parallel in order to get an estimation of the performance over the whole set of problems, and then repeat some tests over concrete, small subsets of problems according to different criteria. The following results are obtained using a time limit of 120 seconds, running 6 tests in parallel in a shared server with 2600 MHz Six-Core AMD Opteron$^{\text{TM}}$with 8GB of RAM.

Table 8.2 shows the results of the four versions described through this report for 13776 FOF and CNF problems in the TPTP Library. The columns correspond to the simple version (with only regularity and re-use), the version with built-in equality, the version enhanced with local failure caching and the rules of the SOL Tableau Calculus and finally the version presented as case study with the specific language for lemma/conclusion literals. The rows of the table contain the total number and percentage of solved problems, the average time taken to solve those, the number and percentage of problems proved within different rating intervals and the number and percentage of problems solved without equality, containing equality and containing only equality (pure equality problems, not included in the row with equality). The last column collects the total of problems of each type as reference. Finally, we provide the number of problems refuted (i.e., counter satisfiable for formulas or satisfiable for set of clauses), the number and percentage of problems for which the time limit of 120 seconds was exceeded (time out) and for which an error was produced (stack overflow or memory allocation errors). In the case of FOF problems, the time taken to perform the clausal transformation is included in the time limit of 120 seconds.

We can see that the best results are achieved by the version presented in the previous chapter as a case study, based on the automatically generated language for conclusion and lemma literals. Remarkably, this version is able to refute 167 more problems than the other three. It does not seem to be particularly good for the problems that contains equality, getting the lower results in that category than the simple ersion. These results suggest that a combination of built-in equality mechanism with the specific

	Simple	RUE built-in equality	SOL local failure caching	Specific lemmas	Total
Proved [%]	2350 [17%]	2289 [17%]	2171 [16%]	2418 [18%]	13776
Average time	6.24 seconds	5.79 seconds	5.72 seconds	6.61 seconds	
Rating 0.0 [%]	905 [44%]	883 [43%]	850 [42%]	898 [44%]	2022
Rating > 0.0 [%]	1445 [12%]	1406 [11%]	1321 [11%]	1520 [12%]	11754
Rating 0.00 – 0.24 [%]	1889 [40%]	1829 [39%]	1775 [38%]	1904 [41%]	4610
Rating 0.25 – 0.50 [%]	297 [14%]	285 [14%]	267 [13%]	329 [16%]	2024
Rating 0.50 – 0.75 [%]	134 [6%]	140 [6%]	102 [4%]	158 [7%]	2086
Rating 0.75 – 1.00 [%]	30 [0%]	35 [0%]	27 [0%]	27 [0%]	5056
No equality [%]	1293 [33%]	1278 [33%]	1198 [31%]	1422 [37%]	3819
With equality [%]	893 [11%]	877 [11%]	838 [10%]	834 [10%]	7843
Pure equality [%]	164 [7%]	134 [6%]	135 [6%]	162 [7%]	2114
Refuted	7	7	7	174	
Time out [%]	11062 [80%]	11129 [80%]	11239 [81%]	10910 [79%]	
Error [%]	368 [3%]	363 [3%]	366 [3%]	447 [3%]	

Table 8.1: Overall TPTP benchmarks for the four different versions

language for lemma and conclusion literals, or even a different treatment in terms of language for equality literals, might work well for a prover of this nature. The average time is also higher, as a fixed amount of time has to be spent computing the language before the proof search is started, and because the more complex distinction between lemma/conclusion literals and goal literals.

The lowest results are obtained by the SOL Tableau Calculus version, as we advanced in Section 6.3 and although the version with the built-in equality mechanism performs below the simple version, the difference is not very marked and it can still solve 5 more problems with rating between 0.75 and 1.00, being the best in that category.

The simple version is faster than the specific version, which makes sense if we think that this involves less computation and less applicable rules at each step. The version with equality and the version with local failure caching are faster but this average time is computed for a lower number of problems. Although the local failure caching procedure should make the SOL Tableau version more effective, our implementation counteracts its effect. Recall all the codification and identification mechanisms that we had to add to the simple version to be able to store substitutions and failure substitutions as literals. It is clear that in a different kind of implementation, probably in a different programming language and following a completely different approach this would perform much better. It is the case of SOLAR [17], a high-performance deduction system based on the SOL Tableau Calculus and incorporating this pruning technique, which is implemented in Java.

For a deeper analysis, we study the performance on the different versions on the FOF and CNF problems separately and within each type we provide the results classified by domain in Tables 8.2 and 8.2. Each row contains the number and percentage of problems solved and the last column contains the total number of problems.

The clausal transformation is included in the time allowed for the FOF problems. The time required by this transformation can vary considerably from some domains to others. To complement the information, we provide the number and percentages of problems for which the clausal transformation cannot be completed within 10 seconds in Table 8.2. For those domains with 0% the clausal transformation can be regarded as insignificant for comparison purposes.

	Simple	RUE built-in equality	SOL local failure caching	Specific lemmas	Total
Proved [%]	1091 [16%]	1049 [15%]	1006 [14%]	1045 [15%]	6800
ALG	3 [11%]	0 [0%]	0 [0%]	1 [3%]	27
ANA	22 [24%]	20 [21%]	22 [24%]	22 [24%]	91
BOO	4 [2%]	8 [5%]	4 [2%]	4 [2%]	139
CAT	29 [46%]	20 [32%]	25 [40%]	24 [38%]	62
COL	91 [38%]	49 [20%]	78 [32%]	90 [37%]	239
COM	7 [50%]	8 [57%]	7 [50%]	5 [35%]	14
FLD	52 [18%]	49 [17%]	44 [15%]	52 [18%]	279
GEO	30 [11%]	34 [13%]	20 [7%]	30 [11%]	253
GRA	1 [100%]	1 [100%]	1 [100%]	1 [100%]	1
GRP	91 [10%]	85 [9%]	72 [8%]	96 [10%]	877
HEN	12 [17%]	20 [29%]	11 [16%]	12 [17%]	67
HWC	0 [0%]	0 [0%]	0 [0%]	0 [0%]	6
HWV	13 [15%]	15 [18%]	15 [18%]	5 [6%]	83
KRS	8 [47%]	8 [47%]	8 [47%]	8 [47%]	17
LAT	35 [11%]	36 [11%]	35 [11%]	35 [11%]	314
LCL	119 [21%]	109 [19%]	102 [18%]	99 [17%]	566
LDA	2 [8%]	0 [0%]	0 [0%]	2 [8%]	23
MGT	20 [25%]	21 [26%]	20 [25%]	21 [26%]	78
MSC	7 [33%]	8 [38%]	7 [33%]	10 [47%]	21
NLP	3 [1%]	5 [1%]	3 [1%]	5 [1%]	258
NUM	17 [5%]	17 [5%]	15 [4%]	16 [4%]	321
PLA	30 [75%]	30 [75%]	28 [70%]	18 [45%]	40
PUZ	32 [31%]	31 [30%]	29 [28%]	31 [30%]	102
REL	1 [0%]	6 [5%]	1 [0%]	1 [0%]	108
RNG	10 [9%]	10 [9%]	10 [9%]	10 [9%]	104
ROB	6 [15%]	4 [10%]	6 [15%]	6 [15%]	40
SET	87 [10%]	83 [10%]	98 [12%]	83 [10%]	800
SWC	2 [0%]	22 [5%]	2 [0%]	2 [0%]	423
SWV	102 [17%]	100 [17%]	106 [18%]	111 [19%]	579
SYN	251 [29%]	246 [29%]	233 [27%]	240 [28%]	844
TOP	4 [16%]	4 [16%]	4 [16%]	5 [20%]	24

Table 8.2: TPTP benchmarks for the four versions: CNF problems

We can observe that the four versions prove more FOF problems than CNF problems, and for both types the solved problems are not evenly distributed between domains. Some domains are clearly much more difficult than others. Although the results achieved by the specific lemma version are the highest globally, this is not so for CNF problems, where the simple version overcomes the other three. It is the best for FOF problems and also the best in some particular domains. To have more information, we have checked which automatically generated languages are in fact different from the *all positive* language, $\langle \mathcal{L}^+ \rangle$ used by the simple version. Table 8.5 contains the number and percentage of problems for each domain with an automatically generated language different from $\langle \mathcal{L}^+ \rangle$, divided in FOF (only those for which the clausal transformation takes less than 10 seconds are included) and

	Simple	RUE built-in equality	SOL local failure caching	Specific lemmas	Total
Proved [%]	1259 [18%]	1240 [17%]	1165 [16%]	1373 [19%]	6976
AGT	14 [26%]	14 [26%]	13 [25%]	9 [17%]	52
ALG	1 [0%]	0 [0%]	0 [0%]	0 [0%]	285
BOO	0 [0%]	0 [0%]	0 [0%]	0 [0%]	1
CAT	2 [2%]	2 [2%]	2 [2%]	2 [2%]	68
COM	6 [19%]	5 [16%]	8 [25%]	1 [3%]	31
CSR	193 [25%]	195 [25%]	165 [21%]	198 [26%]	756
GEG	0 [0%]	0 [0%]	0 [0%]	0 [0%]	1
GEO	238 [70%]	234 [69%]	219 [65%]	235 [69%]	336
GRA	4 [12%]	3 [9%]	2 [6%]	1 [3%]	32
GRP	0 [0%]	0 [0%]	0 [0%]	2 [1%]	195
HAL	0 [0%]	0 [0%]	0 [0%]	1 [11%]	9
KLE	17 [7%]	23 [10%]	15 [6%]	17 [7%]	225
KRS	65 [25%]	58 [22%]	59 [23%]	66 [25%]	254
LAT	6 [1%]	6 [1%]	6 [1%]	5 [1%]	413
LCL	18 [4%]	17 [4%]	18 [4%]	22 [5%]	406
MED	0 [0%]	0 [0%]	0 [0%]	0 [0%]	10
MGT	22 [28%]	22 [28%]	21 [26%]	21 [26%]	78
MSC	4 [40%]	3 [30%]	3 [30%]	3 [30%]	10
NLP	0 [0%]	0 [0%]	0 [0%]	86 [33%]	257
NUM	75 [13%]	78 [13%]	71 [12%]	61 [10%]	554
PLA	0 [0%]	0 [0%]	0 [0%]	1 [16%]	6
PRO	8 [11%]	8 [11%]	7 [9%]	3 [4%]	72
PUZ	4 [13%]	4 [13%]	4 [13%]	4 [13%]	30
REL	1 [0%]	6 [5%]	0 [0%]	1 [0%]	110
RNG	26 [16%]	31 [19%]	23 [14%]	22 [14%]	157
SET	165 [35%]	153 [32%]	143 [30%]	155 [33%]	468
SEU	106 [11%]	98 [10%]	101 [11%]	108 [11%]	906
SWC	18 [4%]	22 [5%]	15 [3%]	14 [3%]	423
SWV	163 [47%]	159 [45%]	170 [49%]	162 [46%]	346
SYN	99 [26%]	95 [25%]	96 [25%]	169 [45%]	374
TOP	4 [3%]	4 [3%]	4 [3%]	4 [3%]	107

Table 8.3: TPTP benchmarks for the four versions: FOF problems

CNF problems.

For example, domains NLP and SYN of type FOF, in which the specific lemma version is much better than the simple version, have respectively 39.53% and 46.9% of the problems with a language different from $\langle \mathcal{L}^+ \rangle$, whereas AGT or NUM of type FOF, where the specific lemma version performs slightly worse, have almost all of their problems with positive lemma and conclusion literals.

Domain	Number [%]	Total	Domain	Number [%]	Total	Domain	Number [%]	Total
AGT	0 [0.0%]	52	KLE	0 [0.0%]	225	PRO	0 [0.0%]	72
ALG	158 [55.44%]	285	KRS	0 [0.0%]	254	PUZ	17 [56.67%]	30
BOO	0 [0.0%]	1	LAT	259 [62.71%]	413	REL	0 [0.0%]	110
CAT	51 [75.0%]	68	LCL	91 [22.41%]	406	RNG	0 [0.0%]	157
COM	0 [0.0%]	31	MED	0 [0.0%]	10	SET	0 [0.0%]	468
CSR	407 [53.84%]	756	MGT	0 [0.0%]	78	SEU	159 [17.55%]	906
GEG	0 [0.0%]	1	MSC	1 [10.0%]	10	SWC	0 [0.0%]	423
GEO	0 [0.0%]	336	NLP	0 [0.0%]	258	SWV	10 [2.89%]	346
GRA	0 [0.0%]	32	NUM	7 [1.25%]	558	SYN	2 [0.54%]	373
GRP	108 [55.38%]	195	PLA	0 [0.0%]	6	TOP	72 [67.29%]	107
HAL	0 [0.0%]	9						

Table 8.4: FOF problems for which the clausal transformation requires more than 10 seconds

CNF problems				FOF problems			
ALG	24 [88.89%]	LDA	9 [39.13%]	AGT	0 [0.0%]	MGT	21 [26.92%]
ANA	44 [48.35%]	MGT	28 [35.9%]	ALG	96 [67.61%]	MSC	3 [33.33%]
BOO	139 [100.0%]	MSC	18 [85.71%]	BOO	1 [100.0%]	NLP	102 [39.53%]
CAT	56 [90.32%]	NLP	88 [34.11%]	CAT	0 [0.0%]	NUM	178 [31.9%]
COL	201 [84.1%]	NUM	292 [90.97%]	COM	2 [6.45%]	PLA	0 [0.0%]
COM	4 [28.57%]	PLA	4 [10.0%]	CSR	0 [0.0%]	PRO	0 [0.0%]
FLD	279 [100.0%]	PUZ	60 [58.82%]	GEG	0 [0.0%]	PUZ	11 [44.0%]
GEO	143 [56.52%]	REL	108 [100.0%]	GEO	0 [0.0%]	REL	110 [100.0%]
GRA	1 [100.0%]	RNG	104 [100.0%]	GRA	0 [0.0%]	RNG	82 [52.23%]
GRP	822 [93.73%]	ROB	37 [92.5%]	GRP	44 [50.57%]	SET	173 [36.97%]
HEN	67 [100.0%]	SET	646 [80.75%]	HAL	0 [0.0%]	SEU	22 [2.97%]
HWC	5 [83.33%]	SWC	0 [0.0%]	KLE	158 [70.22%]	SWC	0 [0.0%]
HWV	42 [50.6%]	SWV	326 [56.7%]	KRS	13 [5.12%]	SWV	53 [15.73%]
KRS	1 [5.88%]	SYN	149 [19.05%]	LAT	1 [0.86%]	SYN	174 [46.9%]
LAT	288 [91.72%]	TOP	1 [4.17%]	LCL	28 [8.33%]	TOP	1 [3.45%]
LCL	279 [49.29%]			MED	0 [0.0%]		

Table 8.5: CNF and FOF problems with automatically generated languages other than $\langle \mathcal{L}^{+} \rangle$

8.3 Comparison with other theorem provers

Due to the limited time scope of this project we have not been able to run benchmarks for other ATP system in the same conditions of system settings and time as for our prover, so for comparison purposes we have to rely in results already published, even if they are obtained with a different release of the TPTP Library. We combine in Table 8.3 the results obtained by T. Raths and J. Ottens for FOF and CNF problems of the TPTP v.3.7.0 provided in [32], for lean$T^{A}P$ 2.3 [5], leanCoP 1.0 [33], SETHEO 3.3 [36], Prover9 2009-02A [22], leanCoP 2.0 [31] and E 1.0 [41]. The last column contains the results for the simple version of our prover, based on the TPTP Library v.4.0.1, from Table 8.2.

	leanTAP	leanCoP	SETHEO	Otter	Prover9	leanCoP	E	Simple
	2.3	1.0	3.3	3.3	2009-02A	2.0	1.0	(TPTP v.4.0.1)
Proved	692	2496	3139	3295	4299	4763	6510	2350
FOF	405	1105	1296	1389	1664	1797	2541	1259
CNF	287	1391	1843	1906	2635	2966	3969	1091
Rating 0.0	477	1432	1672	1716	2045	2137	2276	905
Rating > 0.0	206	1064	1467	1579	2254	2626	4234	1445
No equality	596	1464	1607	1617	1616	1761	2109	1293
With equality	87	1032	1532	1678	2683	3002	4401	1057
Pure equality	13	149	214	213	696	862	1122	164
Refuted	0	7	40	109	0	35	795	7
Time out	9160	7973	6855	5009	1502	3477	4094	11062
Error	1556	923	1355	2986	5598	3124	0	368

Table 8.6: Overall TPTP benchmarks for the four different versions

The error row includes the problems for which a theorem prover gave up without reaching the time limit, and that is the reason of the high numbers in the columns of Otter and Prover9, that often gave up because of an empty set-of-support. There is a newer version of Prover9 (2009-11A) and leanCoP (2.1), for which we have not been able to find benchmarks results. For E 1.2 the authors provide performance results on the TPTP Library v.4.0.1 in [41], that shows how the performance of the prover has been increased to make it one of the "most powerful theorem provers for pure first order logic currently available". It solves 63.2% of the 13783 problems in the TPTP Library. The only benchmark results from the TPTP Library for SOLAR provided in [17] are referred to 6346 problems in the release v.3.5.0. From those, it can solve 1949 (31%) within a time limit of 60 seconds.

Since the global results reached by the four versions of our prover are below those achieved by provers as leanCoP (versions 1.0 and 2.0), with which our prover has many aspects in common, we have repeated some of the tests with a time limit of 600 seconds and the simple version of the prover to check if the lower time limit has been the cause. We have chosen the domains GEO, KRS, SET and SYN from the FOF problems. None of them present any problems regarding the clausal transformation, as we observe in the above table, and leanCoP seems to achieve good results for them. We can see in Table 8.7 that the time limit is clearly not the cause of the poor results, since the number of solved problems does not increase significantly with the increase from 120 to 600 seconds.

Domain	Simple	Simple	leanCoP 1.0	leanCoP 2.0	Total
	(120 s.)	(600 s.)	(600 s.)	(600 s.)	
GEO	238	239	143	171	336
KRS	65	67	70	105	254
SET	165	168	160	318	468
SYN	99	99	200	217	373

Table 8.7: TPTP benchmarks for the simple version and leanCoP: FOF problems

In the intermediate lemma extension approach, a big search space dedicated to find a deep, closed tableau is replaced by smaller search spaces that lead to a series of more shallow tableaux. It could be that the lack of control in the lemma generation of our implementation increases considerably the global search space. If a lemma can be derived in several ways during the search, this would cause that many lemmas are forward subsumed and all the time invested in generating them is wasted. To get a better understanding of what is behind the low performance, we have analysed the number of lemmas generated during the search performed by our simple version and the number of lemmas that are forward subsumed for solved problems in different domains. For some problems no lemma is generated, as they only contain Horn clauses. Table 8.8 summarises this information for the domains AGT, COL, KRS and RNG (both CNF and FOF problems) and confirms that most of the lemmas generated are actually useless.

Domain	Solved problems	Generated lemmas	Forward-subsumed lemmas [%]
AGT	14	62	54 [87.1%]
COL	91	19	13 [68.42%]
KRS	73	22325	22065 [98.84%]
RNG	36	723	702 [97.1%]

Table 8.8: Generated and subsumed lemmas in problems solved by the simple version

We have experimented with three small variants of the simple prover, in an attempt of improving the performance. Two of them implement two simple approaches to tackle directly this issue of many forward subsumed lemmas. First, only allowing the application of the reduction rule with the immediate ancestor and second, checking for subsumption of a partially generated lemma at each skip step. With the first approach, lemmas would have generally more literals and some time is saved by not checking the whole branch for matching negative literals. The second approach aims to reduce the number of generated lemmas that are forward-subsumed.

The third variant automatically backtrack when a tableau from which a tautology can be derived is constructed during the search. In this tableau, we encounter a negative subgoal $\neg L$, having previously skipped the literal L, that is, L and $\neg L$ are leaves that might not be at the same depth (the negative literal is always encountered after the positive, since with the intermediate lemma extension tableaux are developed from left to right and negative literals are never skipped as leaves). Generally such a pattern is unnecessary in a connection tableau and the procedure can backtrack when it arises. However, this is not the case when using the intermediate lemma extension and introducing this restriction makes the calculus incomplete[2]. This is demonstrated in Example 41.

Example 41. Consider the unsatisfiable set of clauses

$$S = \{P(a) \vee P(b), \underline{P(b)} \vee \neg P(a), \underline{P(a)} \vee \neg P(b), \underline{P(a)} \vee P(b)\}$$

in which we have underlined the literals chosen as conclusion.

Tableaux T_1 and T_2 in Figure 8.1 show what happens when trying to derive a closed tableau with the intermediate lemma extension and the additional restriction on tautology patterns mentioned before. If clause $\underline{P(a)} \vee \neg P(b)$ is used for the extension step at K_1 in T_1, the only choice to close $\neg P(b)$ is to use clause $\underline{P(b)} \vee \neg P(a)$. This either fails because the regularity condition is violated at node K_2, or if regularity is not implemented, it leads to an infinite tableau. If clause $\underline{P(a)} \vee P(b)$ is used instead for the extension step at K_1 in T_2, the literal $P(b)$ is skipped and then, when encountering node K_3 the procedure backtracks due to the tautology pattern. As the conclusion literal in clause $\underline{P(a)} \vee P(b)$ is fixed, the closed tableau T_3 can never be constructed.

[2]It is actually the same kind of problem that arises when introducing a locking refinement in resolution. Sometimes it is necessary to derive tautologies in order to enable some literals to match and produce the empty clause

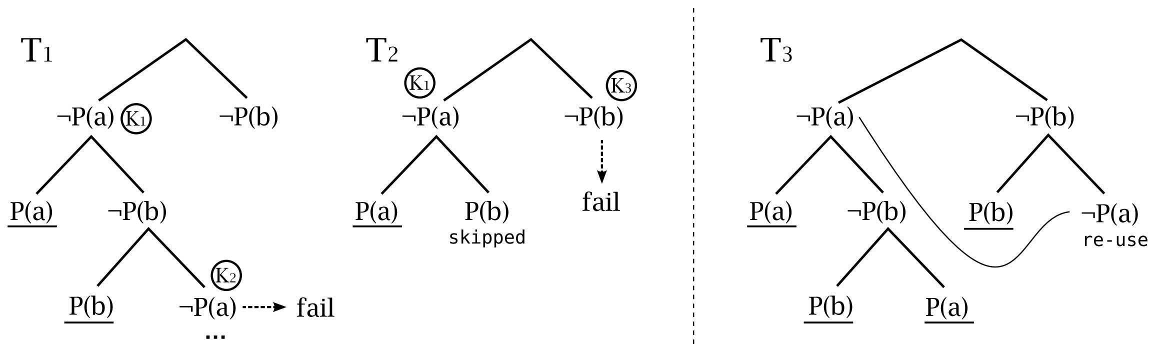


Figure 8.1: Incompleteness introduced by restricting a tautology pattern

Despite being incomplete, it could improve performance as it happened with the restricted backtracking version of leanCoP 2.0 explained in Section 3.3.2, that performs better than the regular version.

We present the benchmark results for the three variants in Table 8.3, following the same format as Table 8.2, for 13776 FOF and CNF problems in the TPTP Library, run with the same computer settings and time limit (120 seconds) as for the other versions. We have included the total number of CNF and FOF problems solved separately in the table but without detailing that total for each domain. The distribution of solved problems by domains is very similar to those shown in Tables 8.2 and 8.2.

	Restricted reduction	Subsumption check at each skip step	No tautology pattern	Total Total
Proved [%]	2332 [17%]	2320 [17%]	2337 [17%]	13776
Average time	5.68 seconds	5.56 seconds	5.55 seconds	
CNF	1096 [16%]	1083 [15%]	1081 [15%]	6800
FOF	1236 [17%]	1237 [17%]	1256 [18%]	6976
Rating 0.0 [%]	906 [44%]	901 [44%]	900 [44%]	2022
Rating > 0.0 [%]	1426 [12%]	1419 [12%]	1437 [12%]	11754
Rating 0.00 – 0.24 [%]	1876 [40%]	1874 [40%]	1879 [40%]	4610
Rating 0.25 – 0.50 [%]	295 [14%]	291 [14%]	293 [14%]	2024
Rating 0.50 – 0.75 [%]	133 [6%]	130 [6%]	133 [6%]	2086
Rating 0.75 – 1.00 [%]	28 [0%]	25 [0%]	32 [0%]	5056
No equality [%]	1289 [33%]	1281 [33%]	1284 [33%]	3819
With equality [%]	877 [11%]	877 [11%]	890 [11%]	7843
Pure equality [%]	166 [7%]	162 [7%]	163 [7%]	2114
Refuted	7	7	7	
Time out [%]	11087 [80%]	11101 [80%]	11073 [80%]	
Error [%]	360 [2%]	359 [2%]	365 [2%]	

Table 8.9: Overall TPTP benchmarks for restricted reduction rule and subsumption check at each skip step

The three variants are faster than the simple prover, with very similar average times. We can observe that although a small increase in the number of solved CNF problems is achieved in the case of restricting the reduction rule to the immediate ancestor, it does not convey an improvement in the total number (it solves 18 problems less than the simple prover). When checking for subsumption at each step, the performance is slightly worse than for the simple prover, probably caused by the amount of time spent in those checks. The variant that backtracks when the tautology pattern arises, despite being incomplete, presents a very similar performance to the simple prover, just slightly below, and it can solve 5 problems more than the restricted reduction version.

Chapter 9

Conclusions and future work

In this project we have investigated the use of Connection Tableau Calculus with the Intermediate Lemma Extension for automated theorem proving. We have implemented a Prolog technology theorem prover based on this framework with very basic shortcuts and have tested it. Although a large search space dedicated to find a deep, closed tableau is replaced by smaller search spaces that lead to a series of more shallow tableaux, the lack of control in the lemma generation increases considerably the global search space. Some drawbacks of the approach, like the fact that for most problems a lemma can be derived in several ways or that many lemmas are forward subsumed or simply not used in the final proof, can justify that a theorem prover based in this framework can be surpassed by the regular Connection Tableau Calculus implemented without refinements in the simplest version of leanCoP 2.0 [31], for example.

In order to improve the simple version, we have attempted to equip it with a built-in equality mechanism based on the RUE resolution. We have defined an extension of the calculus wit rules to deal with equality literals and have proved that it is sound and complete. The new rules have been incorporated in an alternative version of the simple prover, which has turned out not to be better in practise than using explicitly the equality axioms combined with some refinements to reduce the search space. We have adapted to our particular setting a sophisticated pruning technique, the local failure caching method, originally proposed for the SOL Tableau Calculus for consequence finding. The efficiency flaws of our adaptation and implementation have become evident at the view of the benchmark results collected in the previous chapter for this specific version. We believe that this is consequence of our own design and implementation, rather than the unsuitableness of local failure caching for refutation finding. The reason, as we advanced in Section 6.2, may be that we have not implemented the SOL Tableau Calculus itself, but have embedded its rules and pruning techniques into our previously implemented simple version, in a relatively forced manner.

More fruitful has been surprisingly the idea of production fields for defining characteristic clauses in consequence finding, transformed and combined with a feature of hyper-resolution to define languages that specify lemma and conclusion literals. This language definition, developed as a case study, has resulted from a thorough analysis of what requirements are needed for defining this kind of language in order to preserve soundness and completeness of the method. As a result, we have proposed a generalisation of Connection Tableau Calculus with the Intermediate Lemma Extension and have proved that it is sound and complete. Moreover, we have devised a simple heuristic to automatically generate a language tailored to a particular problem that have turned out to be effective despite its apparent simplicity. It has been the fourth version of the prover that incorporates the definition and automatic generation of languages the one that has achieved the best results in the tests performed in the TPTP Library, especially in the number of refuted problems.

A good understanding of the general flaws of the Intermediate Lemma Extension has been gained through this project. According to Table 8.8 presented in the previous chapter, a bit amount of the search space is dedicated to generate lemmas that are forward subsumed. This usually happens because the same lemma is derived several times during the search. A control mechanism in the lemma generation is indispensable to attain a reasonable level of performance. Yet it has to be designed very carefully. The fixed structure imposed by the choice of an unique conclusion literal on each clause makes the framework quite inflexible to incorporate some good pruning techniques like restricted backtracking [32], discussed in Section 3.3 and below. Even simple restrictions such as not allowing two complimentary literals part of a tautology as leaves turns the method into incomplete, as we demonstrated in the previous chapter in Section 8.3. This inflexibility is easily noticed when thinking from the point of view of possible connections between literals. In Figure 9.1 the possible connections between literals of clauses $\{\neg P(a), Q(e), \underline{P(y)} \vee \neg Q(z) \vee P(z), Q(a)\}$ have been drawn. Connection (3) is irrelevant with respect to the satisfiability of this set and connection (1) is initially unavailable due to the choice of $P(y)$ as conclusion literal. Any pruning method that interferes with connections (2) and (4) would cause that a closed tableau cannot be found.

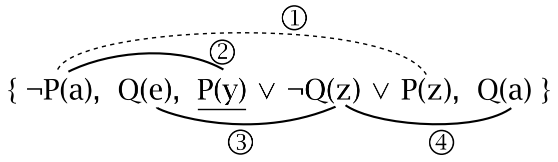


Figure 9.1: Connections between literals

Regarding future research directions, apart from restricting and controlling effectively the lemma generation, this project leaves diverse paths for improvement and extension possibilities. Several have already been suggested in various parts of the report. We collect them here, together with few other ideas. Some of them are relatively simple add-ons to the prover already implemented whereas others are small projects in themselves.

- Restricting backtracking within the intermediate lemma extension framework. We introduced this topic in Section 3.3, where we adverted to the difficulties that a restricted version of backtracking involves when adding the restrictions introduced by the intermediate lemma extension. It seems that a distinction between essential and non-essential backtracking made in the same way introduced by J. Ottens in [32] is not possible, but we have not proved that. It would be interesting to study this in more depth, looking at which connections between literals are essential to find a proof, which ones are eliminated by fixing an unique conclusion literal in each clause and which ones are possible when performing only essential backtracking steps. As part of this deeper study, a possible relation with the Connection Graphs introduced by R. Kowalski ([24]) could be also investigated. Independently, the definition of essential backtracking could be relaxed to some extent that allows us to distinguish between those essential and non-essential backtracking in the context of intermediate lemma generation without loosing so much completeness as it apparently happens with the given definition. Relaxing the choice of conclusion literals (that is, allowing any positive literal to be chosen as conclusion dynamically, during the search) would be an alternative to consider with the current definition of essential backtracking. Particularly relevant would be the possibility of pruning paths of the search space likely to conclude in a forward subsumed lemma.

- Introduction of ordering on terms and orientation of equations with the disagreement extension and decomposition rules. After setting the basis of the disconnection calculus, we described two approaches for dealing with equality in that framework. The first one was based on the equality linking rule and reflexivity linking and the second one relied on the concepts of disagreement sets from RUE resolution. For both of them, a partial ordering on the terms was considered in order to control the symmetry property of equality, so that overlapping was only allowed with maximal sides of equations. Our adaptation of RUE resolution to connection tableau calculus applies symmetry in an unrestricted way. Then a clear extension to this calculus is to introduce an ordering to change this. It would be necessary to study which characteristics should the ordering have to be compatible with the skip and lemma derivation rules and also investigate the impact on the performance.

- Universal variables in the re-use shortcut. When we presented re-use among other shortcuts in Section 2.2.2, and formalised it within the rules of the connection tableau calculus in Table 2.2.2, we did not mention any bindings made to the variables or substitutions. Further on the report, in Section 5.4 we described how re-use has been implemented in our prover and explained that the shortcut was only applicable when a subgoal was *syntactically* identical to a previously closed literal with which it shares all its ancestors. No bindings are allowed, resulting in a very restrictive application of the shortcut. However, if a subgoal $K = \neg L(x)$ (variable x occurs in L) is closed and variable x is not bound in the closure, it would be as if the branch could be closed for any x. This would allows us to close nodes of the form $\neg L(x/t)$ (occurrences of x in L replaced by term t) in branches to the right of K that are known to share the necessary ancestors, as if $\forall x L(x)$ was added to the tableau. This extension could be implemented in all versions of the prover, first for the case in which no ancestors are involved in the closure of K, and $\forall x L(x)$ can be added. More sophisticate implementations could consider also the case in which some ancestors are involved in the closure beneath K, but variables shared by K and its ancestors do not appear in any siblings of K.

- Restriction of the reduction rule. A simple variant of the intermediate lemma extension that could be analysed and evaluated is that in which reduction steps are forbidden. We have already observed that restricting it to the most immediate parent improves slightly the unrestricted version for CNF problems and it is faster. Entirely suppressing its application has the additional advantage that a compiled implementation (of the likes of that presented in Section 5.6) could be developed, as there is no need of keeping the literals in the current branch. More lemmas would be generated, but that could be worthwhile with this faster kind of implementation and not having to store and search for suitable literals that match with the current positive goal on the branch.

- Application of local failure caching to positive literals and possible extension to specific lemmas. Regarding the SOL tableau calculus pruning methods, it is necessary to improve the current implementation of local failure caching, to make it more efficient and avoid the poor performance results obtained by this version of the prover. Once this is achieved, an obvious extension that was pointed out in our description of the implementation in Section 6.2 is to extend it to positive literals. The purpose of this would be to avoid exploring an alternative application of the reduction rule after having applied the reduction rule before and having obtained a more general substitution, even though this is not a very frequent situation. It would be also interesting to explore the possibility of combining the local failure caching technique with the specific lemmas extension.

Appendix A

Brief user guide

This appendix provides some guidelines to invoke the different versions of the prover and it is complemented by the `Readme` file supplied with the Prolog code. That file contains further installation instructions. SWI-Prolog is required.

Execution of the prover:

```
./prover.sh <problem> <prover> <time limit>
```

where `<problem>` is the name of the problem file, `<prover>` is the version of the prover (`simple, disagreement, sol, specific`) and `<time limit>` is the time limit in seconds

Example:

```
./prover.sh tests/prueba1.p simple 10
```

A.1 Problem file format

The problem file has to contain a Prolog term of the form

```
f(<formula>).
```

in which `<formula>` is a first-order formula built from Prolog terms (atomic formulas), the logical connectives ∼ (negation), `;` (disjunction), `,` (conjunction), `=>` (implication), `<=>` (equivalence), and the logical quantifiers `all X:` (universal) and `ex X:` (existential) where `X` is a Prolog variable. Equality is represented as `=`.

For example, `f( ((p , all X:(p=>q(X))) => q(a)) ).` represents the formula

$$(P \land \forall x\, (P \to Q(x)) \to Q(a)$$

The formula can appear in the form of *Axioms* $\Rightarrow$ *Conjecture*, where the task of the prover is to show that *Conjecture* follows from *Axioms*. In this case, the unsatisfiability of the set $\{Axioms, \neg Conjecture\}$ has to be shown and the clausal transformation has to be made in accordance. To activate this negation of the conjecture, the file has to contain a Prolog term of the form:

```
type(conjecture).
```

Alternatively, `<formula>` can be a set of clauses built from Prolog atoms. The set is delimited by `[,]`, and every clause is a set of literals enclosed into `[,]` and separated by `,`.

For example, `f([[-r(b,W),-r(h(W),W)],[-r(Z,h(Z))],[r(X,Y),r(Y,X)]]).` represents the set of clauses

$$\{\neg R(b,w) \vee R(h(w),w), \neg R(z,h(z)), R(x,y) \vee R(y,x)\}$$

If the file is not in the right format or does not exist, an error message informing of this will appear. A format file, `format.ilemma`, is provided for translation of the TPTP Library problem files to this format through the `tptp2X` tool [48].

A.2 Output

The output of the proof is controlled with the parameter `PRINT_PROOF` that can be set in the file `prover.sh`. The result can be either `Theorem` or `Unsatisfiable`, for valid first order formulas or unsatisfiable set of clauses respectively, or `CounterSatisfiable` or `Satisfiable`, for invalid first order formulas or satisfiable set of clauses respectively. If the time limit is reached before a closed tableau can be constructed, the result will be `Timeout`. When a proof is found the result and the time needed in seconds is printed. If the option to print the proof is enabled with `PRINT_PROOF=yes`, the proof found is also shown as a sequence of tableaux. Each tree is written in the order of application of extension steps. Each line corresponds to one clause, with its level written in the right. Each tableau in the sequence except for the last leads to a derived lemma. The following example clarifies this.

Example 42. Consider the following unsatisfiable set of clauses as input:

$$S = \{\neg R(b,w) \vee \neg R(h(w),w), \neg R(z,h(z)), R(x,y) \vee R(y,x)\}$$

- Example of output with `PRINT_PROOF=no`:

  ```
  Unsatisfiable (0.01)
  ```

- Example of output with `PRINT_PROOF=yes`:

  ```
  Unsatisfiable (0.01)
  [] [-r(b,b),-r(h(b),b)] (1)
  [r(b,b)] [r(b,b)] (1)
  [r(h(b),b)] [r(b,h(b))] (1)
  ---> Lemma [r(b,h(b))] generated with level 2

  [] [-r(b,_G1602),-r(h(_G1602),_G1602)] (1)
  [r(b,_G1602)] [r(_G1602,b)] (1)
  [r(h(_G1602),_G1602)] [r(_G1602,h(_G1602))] (1)
  ---> Lemma [r(_G1602,b),r(_G1602,h(_G1602))] generated with level 2

  [] [-r(_G1746,h(_G1746))] (1)
  [r(_G1746,h(_G1746))] [r(h(_G1746),_G1746)] (1)
  ---> Lemma [r(h(_G1746),_G1746)] generated with level 2

  [] [-r(b,b),-r(h(b),b)] (1)
  [r(b,b)] [r(b,b)] (1)
  [r(h(b),b)] [r(h(b),h(h(b)))] (2)
  ```

```
---> Lemma [r(h(b),h(h(b)))] generated with level 3

[] [-r(b,b),-r(h(b),b)] (1)
[r(b,b)] [r(b,b)] (1)
[r(h(b),b)] [] (2)
End of the proof.
```

The printed proof corresponds to the tableau sequence represented in Figure A.1.

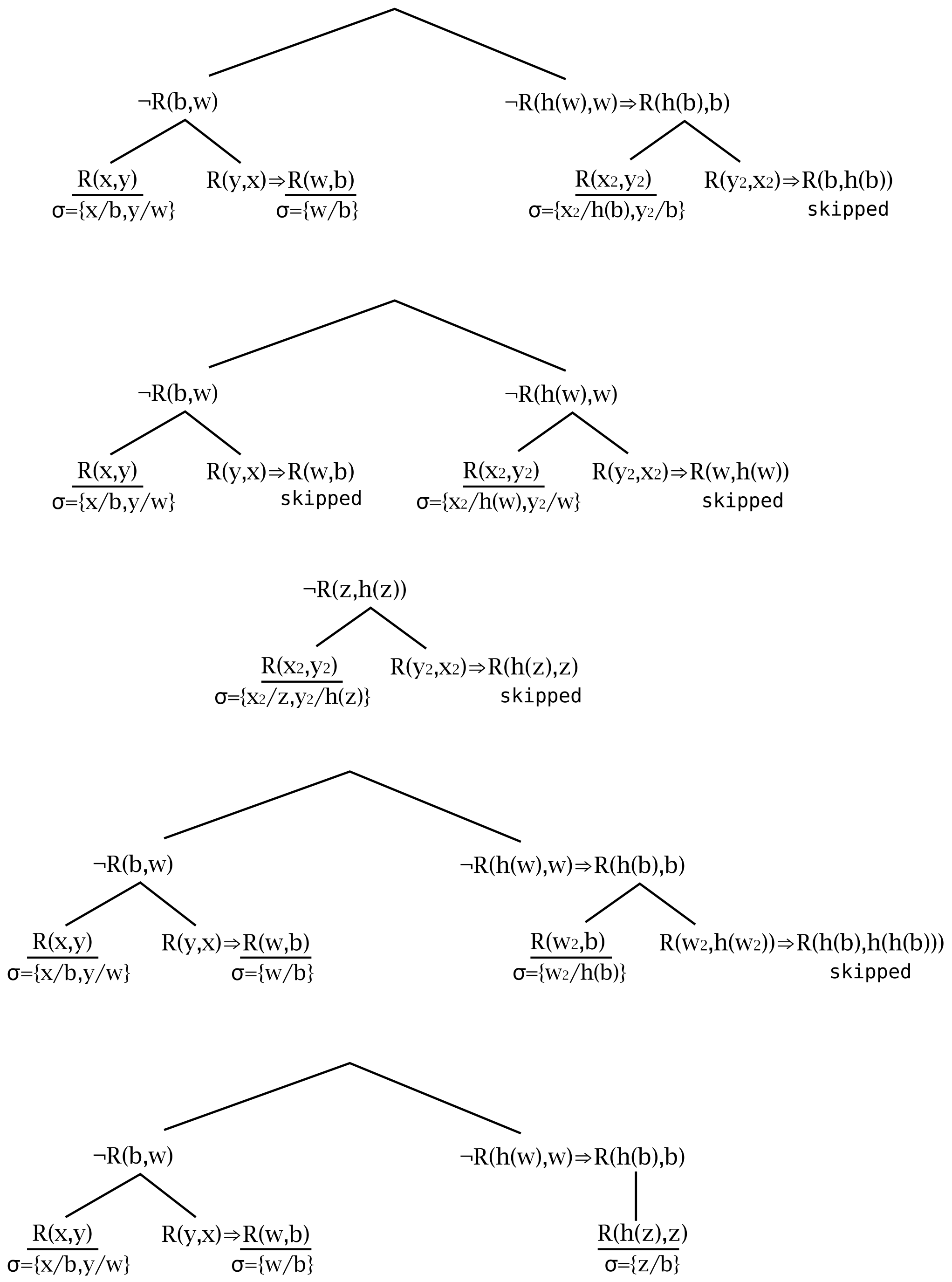


Figure A.1: Example of proof representation

Bibliography

[1] Acl2. http://userweb.cs.utexas.edu/users/moore/acl2/.

[2] The eclipse constraint programming system. http://eclipseclp.org/.

[3] Swi-prolog. http://www.swi-prolog.org/.

[4] Bachmair, L., Ganzinger, H., and Voronkov, A. Elimination of equality via transformation with ordering constraints. Technical Report MPI-I-97-2-012, 1997.

[5] Beckert, B., and Posegga, J. leanT^AP: Lean tableau-based deduction. *Journal of Automated Reasoning 15*, 3, 339–358.

[6] Beckert, B., and Posegga, J. leanT^AP. http://www.uni-koblenz.de/ beckert/leantap/.

[7] Bibel, W. Matings in matrices. *Communications of the ACM 26*, 11 (1983), 844–852.

[8] Bibel, W. *Automated Theorem Proving*, 2nd ed. Vieweg Verlag, Braunschweig, 1987.

[9] Billon, J.-P. The disconnection method - a confluent integration of unification in the analytic framework. In *TABLEAUX* (1996), pp. 110–126.

[10] Brand, D. Proving theorems with the modification method. *SIAM Journal of Computing 4*, 4 (1975), 412–430.

[11] D'Agostino, M., G. D. H. R., and Posegga, J., Eds. *Handbook of Tableau Methods*. Springer, 1999.

[12] Degtyarev, A., and Voronkov, A. Equality elimination for the tableau method. In *Design and Implementation of Symbolic Computation Systems*, J. Calmet and C. Limongelli, Eds., vol. 1128 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1996, pp. 46–60. 10.1007/3-540-61697-7_4.

[13] Digricoli, V. J., and Harrison, M. C. Equality-based binary resolution. *Journal of the ACM (JACM) 33*, 2 (1986), 253–289.

[14] Fitting, M. *First-Order Logic and Automated Theorem Proving*, 2nd ed. Springer, 1996.

[15] Geoff Sutcliffe, Stephan Schulz, K. C., and Gelder, A. V. Using the tptp language for writing derivations and finite interpretations. In *Automated Reasoning*, U. Furbach and N. Shankar, Eds., vol. 4130 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 67–81. 10.1007/11814771_7.

[16] Gordon, M. J. C. "hol". http://hol.sourceforge.net/.

[17] Hidetomo Nabeshima, Koji Iwanuma, K. I., and Ray, O. SOLAR: An automated deduction system for consequence finding. *AI Communications 23*, 2-3 (2010), 183–203.

[18] Inoue, K. Linear resolution for consequence finding. *Artificial Intelligence 56*, 2-3 (1992), 301 – 353.

[19] Inoue, K. Linear resolution for consequence finding. *Artificial Intelligence 56*, 2-3 (1992), 301–353.

[20] Inoue, K. Automated abduction. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II* (London, UK, 2002), Springer-Verlag, pp. 311–341.

[21] Iwanuma, K., Inoue, K., and Satoh, K. Completeness of pruning methods for consequence finding procedure sol. In *In Proceedings of the Third International Workshop on First-Order Theorem Proving (FTP'2000* (2000), pp. 89–100.

[22] Kaufmann, M., and Moore, J. S. Prover 9 and mace 4. http://www.cs.unm.edu/ mccune/prover9/.

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

[24] Kowalski, R. A proof procedure using connection graphs. *Journal of the ACM (JACM) 22*, 4 (1975), 572–595.

[25] Kowalski, R., and Kuehner, D. Linear resolution with selection function. In *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, J. Siekmann and G. Wrightson, Eds. Springer, Berlin, Heidelberg, 1983, pp. 542–577.

[26] L. Wos, W. M. Otter. http://www.cs.unm.edu/ mccune/otter/.

[27] Letz, R. *First-order calculi and proof procedures for automated deduction.* PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, 1999.

[28] Letz, R., and Stenz, G. The disconnection tableau calculus. *Journal of Automated Reasoning 38*, 1-3 (2007), 79–126.

[29] Letz, R., and Stenzt, G. Integration of equality reasoning into the disconnection calculus. In *TABLEAUX '02: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods* (London, UK, 2002), Springer-Verlag, pp. 176–190.

[30] Loveland, D. W. Mechanical theorem-proving by model elimination. *Journal of the ACM (JACM) 15*, 2 (1968), 236–251.

[31] Otten, J. leancop 2.0 and ileancop 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 283–291.

[32] Otten, J. Restricting backtracking in connection calculi. *AI Communications 23*, 2-3 (2010), 159–182.

[33] Otten, J., and Bibel, W. leancop: lean connection-based theorem proving. *Journal of Symbolic Computation 36*, 1-2 (2003), 139–161.

[34] Paskevich, A. Connection tableaux with lazy paramodulation. *Journal of Automated Reasoning 40*, 2-3 (2008), 179–194.

[35] Paulson, L., and Nipkow, T. Isabelle. http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[36] R. Letz, S. Bayerl, J. S., and Bibel, W. Setheo: A high-performance theorem prover. *Journal of Automated Reasoning*, 8 (1992).

[37] R. Letz, K. M., and Goller, C. Cotrolled integration of the cut rule into connection tableaux calculi. *Journal of Automated Reasoning 13*, 3 (1994), 297–337.

[38] Ray, O., and Inoue, K. A consequence finding approach for full clausal abduction. In *DS'07: Proceedings of the 10th international conference on Discovery science* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 173–184.

[39] Robinson, J. A. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics 1* (1965), 227–234.

[40] Robinson, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM) 12*, 1 (1965), 23–41.

[41] Schulz, S. E. http://www4.informatik.tu-muenchen.de/ schulz/WORK/details.html.

[42] Shie-Jue, L., and Plaisted, D. A. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning 1*, 9 (1992), 25–42.

[43] Smullyan, R. M. *First-Order Logic.* Springer-Verlag, 1968. Republished by Dover, New York, in 1995.

[44] Stenz, G., and Wolf, A. E-setheo: An automated3 theorem prover. In *TABLEAUX '00: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods* (London, UK, 2000), Springer-Verlag, pp. 436–440.

[45] Stickel, M. E. A prolog technology theorem prover: implementation by an extended prolog computer. *Journal of Automatic Reasoning 4*, 4 (1988), 353–380.

[46] Stickel, M. E. A prolog technology theorem prover: a new exposition and implementation in prolog. *Theoretical Computer Science 104*, 1 (1992), 109–128.

[47] Sutcliffe, G. System on tptp. http://www.cs.miami.edu/ tptp/cgi-bin/SystemOnTPTP.

[48] Sutcliffe, G. The tptp problem library technical manual. http://www.cs.miami.edu/ tptp/TPTP/TR/TPTPTR.shtml.

[49] Sutcliffe, G. Evaluating general purpose automated theorem proving systems. *Artificial Intelligence 131*, 1-2 (2001), 39–54.

[50] Sutcliffe, G. TPTP, TSTP, CASC, etc. In *Proceedings of the 2nd International Computer Science Symposium in Russia* (2007), V. Diekert, M. Volkov, and A. Voronkov, Eds., no. 4649 in Lecture Notes in Computer Science, Springer-Verlag, pp. 7–23.

[51] Sutcliffe, G. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning 43*, 4 (2009), 337–362.

[52] Sutcliffe, G. The TPTP World - Infrastructure for Automated Reasoning. In *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning* (2010), E. Clarke and A. Voronkov, Eds., no. 6355 in Lecture Notes in Artificial Intelligence, Springer-Verlag, p. To appear.

[53] Suttner, C. B. Sps-parallelism + setheo = sptheo. *Journal of Automated Reasoning 22*, 4 (1999), 397–431.

[54] Voronkov, A. Vampire. http://www.voronkov.com/vampire.cgi.

[55] Walther, C. A mechanical solution of schubert's steamroller by many-sorted resolution. *Artificial Intelligence 26*, 2 (1985), 217 – 224.

[56] Wolf, A. p-setheo: Strategy parallelism in automated theorem proving. In *TABLEAUX '98: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods* (London, UK, 1998), Springer-Verlag, pp. 320–324.