Proof Search Without Backtracking for Free Variable Tableaux

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Martin Giese

aus Berlin

Tag der mündlichen Prüfung: 12. Juli 2002

Erster Gutachter: Prof. Dr. Reiner Hähnle, Chalmers University of Technology, Göteborg (Schweden)

Zweiter Gutachter: Prof. Dr. Wolfram Menzel, Universität Karlsruhe

Acknowledgments

I would like to take this opportunity to express my gratitude to a number of people without whom I could never have completed this thesis.

First and foremost, I sincerely thank my supervisor, Prof. Dr. Reiner Hähnle. Without his encouragement, advice, and support, I would have neither begun nor finished this thesis. I am deeply impressed with the thoroughness with which he read through the submitted version of this thesis in a matter of days, providing valuable comments at a very detailed level.

I am also grateful to Prof. Dr. Wolfram Menzel, who not only acted as 'Korreferent', but was always available for discussions. He has supported and guided me in my studies ever since I started as a student assistent in 1995.

I am indebted to my colleagues, Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, and Elmar Habermalz for the pleasant working atmosphere and the numerous discussions which provided the substratum from which grew the results presented here. I also thank Richard Bubel, who helped me with parts of the implementation.

Finally, a very special Thank You goes to my wife Simone. How would I have managed the final months of work on this thesis without your constant love and support? I owe you more than I can put into words.

Rücksetzungsfreie Beweissuche in Tableaukalkülen

Diese Dissertation befaßt sich mit dem automatischen Beweisen von Formeln der Prädikatenlogik erster Stufe im Umfeld des KeY-Projekts [ABB$^+$00]. Die Deduktionskomponente des KeY-Systems wird verwendet, um Aussagen über Programme zu beweisen, die in einer dynamischen Logik [Bec01] ausgedrückt sind. Um die Probleme zu vermeiden, die sich beim Zusammenschluß von separaten automatischen und interaktiven Beweissystemen ergeben, wird ein Beweiser entwickelt, der in einem einzigen System und unter Benutzung eines einzigen Kalküls sowohl interaktives als auch automatisches Beweisen zuläßt. Zum interaktiven Beweisen wurden in bisherigen Systemen entweder *Sequenzenkalküle* oder *natürliches Schließen* verwendet. Um eine möglichst nahtlose Integration der automatischen Beweissuche zu ermöglichen haben wir uns im Vorfeld für eine Version des Sequenzenkalküls entschieden, der freie Variablen im Sinne des Tableaukalküls benutzt [Gie98]. Da für die dynamische Logik keine Klauselnormalform existiert, muß ein Nicht-Normalform-Kalkül verwendet werden.

Eine Schwierigkeit bei tableaubasierten Verfahren ist allerdings deren Verwendung von destruktiven Regeln, in denen die Beweisprozedur eine bestimmte Instantiierung auswählt, die vorhandene Formeln im Beweisbaum zerstört. Solche Regeln machen es schwer, eine vollständige Beweisprozedur zu entwerfen, die ohne Rücksetzung (*backtracking*) arbeitet. In fast allen bisherigen Tableaubeweisern wird dieses Problem durch die sogenannte iterative Tiefensuche gelöst, die einen beschränkten Teil des Suchraums durch Tiefensuche durch Rücksetzung exploriert, und bei Mißerfolg die Schranke erhöht.

Diese rücksetzungsbasierten Verfahren sind für das beschriebene Szenario ungeeignet: Zum einen kann aus einem bestimmten Beweiszustand nicht abgelesen werden, welche Teile des Suchbaums bereits durchsucht wurden. Wenn die automatische Prozedur innerhalb einer gewissen Zeit keinen Beweis findet, besteht also für den Benutzer des Systems keine Möglichkeit, nachzuvollziehen, warum der Versuch fehlgeschlagen ist. Zweitens liegt es in der Natur von Rücksetzungsschritten, daß alle Information, die nach dem Rücksetzpunkt hergeleitet wurde, verlorengeht, da sie von der zurückzusetzenden Entscheidung abhängen kann. Dadurch muß aber eventuell auch Information, die von dieser Entscheidung unabhängig ist neu berechnet werden, was in der Praxis zu erheblichen Effizienzeinbußen führt. Drittens schließlich ist es im Bereich der Verifikation nicht ungewöhnlich, daß eine zu beweisende Aussage gar nicht wahr ist, etwa weil die Implementierung oder Spezifikation fehlerhaft ist. In diesem Fall möchte man aus dem fehlgeschlagenen Beweisanfang ein Gegenbeispiel ableiten [Ahr02], was in einer rücksetzungsbasierten Beweisprozedur nicht möglich ist.

Der Hauptbeitrag dieser Abhandlung besteht in einem Verfahren, das die rücksetzungsfreie Beweissuche für Tableaux mit freien Variablen ermöglicht. Um Rücksetzungsschritte zu vermeiden, wird beim Beweisaufbau auf destruktive Schritte völlig verzichtet. Der wesentliche destruktive Schritt im Tableaukalkül mit freien Variablen ist der Astabschluß, in welchem durch Unifikation von komplementären Literalen einige der freien

Variablen global instantiiert werden. In dem erarbeiteten Verfahren werden diese Abschlüsse nicht durchgeführt. Stattdessen wird nach jedem Beweisschritt überprüft, ob es eine einzige *schließende Substitution* für die freien Variablen gibt, die sämtliche Äste des Tableaus gleichzeitig abschließt.

Dieses Prinzip ist an sich nicht neu, es wird gelegentlich als *delayed closure* bezeichnet. Da der immer wieder auszuführende Abschlußtest aber als zu aufwendig galt, wurde es nie in einer ernsthaften Implementierung verwendet. Hier wird gezeigt, daß es möglich ist, die Berechnungen für den globalen Abschlußtest *inkrementell* durchzuführen, so daß für jede hinzukommende Abschlußmöglichkeit mit in der Praxis relativ geringem Aufwand feststellbar ist, ob der gesamte Beweis geschlossen werden kann. Die Praktikabilität des Verfahrens wird durch einen experimentellen Vergleich mit einem auf Rücksetzung basierenden Beweiser belegt.

Eine Stärke dieser Methode des *inkrementellen Abschlusses* von Tableaux ist die Verträglichkeit mit vielen Verfeinerungen und Suchraumeinschränkungen, die von Beweisern mit iterativer Tiefensuche bekannt sind. Die Verwendung solcher Verfeinerungen ist für eine effiziente Implementierung von großer Bedeutung. Außer solchen von bestehenden Beweisprozeduren übernommenen Verfeinerungen werden einige neue Möglichkeiten vorgestellt, die direkt mit dem Verfahren des inkrementellen Abschlusses zusammenhängen.

Ferner wird eine Familie von Simplifikationsregeln für den Tableaukalkül vorgestellt, die es erlauben, einige der erfolgreichsten Techniken aus modernen automatischen Beweisern im Rahmen eines mit inkrementellem Abschluß arbeitenden Systems für ein Nicht-Normalform-Kalkül zu verwenden. Die vorgestellten Regeln sind auch für einen gewöhnlichen rücksetzungsbasierten Beweiser anwendbar.

Ein besonders wichtiger Aspekt beim automatischen Beweisen ist die Behandlung der Gleichheit. Um zu einem effizienten Verfahren zu gelangen, muß eine auf Rewriting basierende Gleichheitsbehandlung in den Kalkül eingebaut werden. In dieser Arbeit wird ein Tableaukalkül beschrieben, der auf dem bei Resolutionsbeweisern bewährten *basic superposition* Kalkül mit syntaktischen Ordnungsconstraints beruht. Es wird eine Methode vorgestellt, die Vollständigkeit eines solchen Kalküls nachzuweisen, die zu erheblich einfacheren Vollständigkeitsbeweisen als bisherige Ansätze führt, und die es erleichtert, bekannte Techniken für Resolutionsbasierte Saturierungsverfahren auf Tableaukalküle zu übertragen. Wie für die erwähnten Simplifikationsregeln, sind auch diese Resultate ebenso relevant für rücksetzungsbasierte Beweisprozeduren.

Contents

List of Figures

1 Introduction

This is a Work on automated theorem proving. The goal of this discipline is the construction of software which takes statements expressed in some logical formalism as input, and eventually gives an answer indicating whether that statement is logically valid with respect to a given semantics of the input language. One talks of a 'prover' because such software usually proceeds by attempting to construct some sort of proof of the statement.

We shall specifically be talking about automated theorem proving for classical first order predicate logic. The validity of first order formulae is only semi-decidable. Accordingly, an automated theorem prover will in general not terminate if the input is not valid. The minimal behaviour of a first order prover is thus simply to terminate for all valid statements—this is called completeness—and run indefinitely otherwise—this is soundness.

What, then, is a good theorem prover? Well, one that terminates as quickly as possible, if it terminates at all. To find the best theorem prover, one takes a collection of valid statements, and measures the execution time required by various programs.

Unfortunately, this approach does not work: Typically, every theorem prover will be faster than other provers for some problems and slower for others. The lesson to be learnt from this is that which theorem prover is best depends on what problems must be solved. In other words, one should be clear about the application domain before one chooses an automated theorem proving technology.[1]

This observation obviously applies to any piece of software. For instance, which is the best sorting algorithm depends on factors like the number of data sets to be sorted, the range of values of the sort key, the amount of main memory available, and the possibilities of accessing the data to be sorted. But for automated theorem provers, this truth is sometimes neglected.

The automated theorem proving method we are going to present has been developed in the context of the KeY project [ABB$^+$00, ABB$^+$02]. This project was started in November 1998 at the University of Karlsruhe, Germany. It is now a joint project of the University of Karlsruhe and Chalmers University of Technology, Göteborg, Sweden. The aim of the KeY project is to integrate formal software specification and verification into industrial software engineering processes. While a large part of this project is concerned with the development of formal *specifications* for object oriented software, it also important to be able to formally *verify* parts of a software system with respect to the given specifications.

[1]This is a one-way implication: even within a given application domain, problems might have very different structure. The point is that we can't say *anything* without knowing the application domain.

There are two ways in which theorem proving plays a role in this context:

- One can reason about properties of a specification, independently of an implementation. For instance, one might want to prove that a given invariant of some class implies that of a superclass. Another application might be the detection of specification errors by proving specifications to be inconsistent.

- One can reason about implementations. In particular, one could prove that an implementation guarantees a post-condition given in the specification, or that it preserves a class invariant.

The theorem prover to be used in such a context has to meet a number of requirements:

1. It has to support the *interactive* construction of proofs. First, proving properties of programs includes reasoning about numbers and recursive data structures, and most proofs about these require induction. Automated theorem proving over inductive theories would require guessing induction hypotheses, which is feasible only for very simple proofs. Second, reasoning about programs containing loops or recursion also requires induction. Third, even when induction is not needed, proofs of program properties are often so large that an automated theorem prover cannot find them without help from the user.

2. It must support *automated proof search* whenever possible. If parts of a proof can be found automatically, the system should support this. Ideally, user interaction should only be needed for the 'creative' parts of the proof. The 'trivial', 'boring' or 'standard' parts should be done automatically.

3. The system has to use a *specialized logic* which is suitable for reasoning about programs. As parts of the proofs about programs will have to be constructed interactively, it is important to choose a formalism which is transparent to a human user. In the KeY project, the JAVADL logic is used, which is a dynamic logic for reasoning about JAVA programs [Bec01]. All we need to know about that logic here is that it is a kind of modal logic, where the box and diamond operators are decorated with programs.

4. The construction of *counter examples* should be supported, in case the user tries to prove a statement which is not valid [Ahr01, Ahr02]. This is particularly relevant for the context of program verification, as the system should help the user in detecting mistakes in the specification or implementation.

5. The previous requirements should be met by a uniform logical framework, and implemented in a single system. In particular, interactive and automated proof construction should be based on the same calculus and data structures. Previous experiments [ABH$^+$98] with the integration of interactive and automated theorem proving in a program verification context have shown that a hybrid system consisting of separate interactive and automated provers is problematic.

This wish list shows how inadequate our initial termination-based description of an automated theorem prover was. Although this work is only concerned with the aspect of automated theorem proving for first order logic, these requirements restrict the possible choices of calculus and proof procedure.

First of all, we cannot use a calculus that requires problems to be given in clausal form. Transforming proof obligations into a normal form is not acceptable for an interactive system, because the user needs to recognize the structure of formulae coming from the specification. There is also no clausal form for the formulae of dynamic logic.

This essentially excludes resolution based calculi from the start. Although there are non-clausal versions of resolution (see e.g. [BG01]), it is not clear whether these could be applied for a dynamic logic, and resolution is also not very well suited for interactive use, as it is too different from the way humans construct proofs.

On the other hand, a sequent calculus for dynamic logic has successfully been used for interactive verification in a previous project [Rei92]. The non-clausal free variable tableau calculus (see eg. [Häh01, Fit96]) is very close to a sequent calculus, and tableau methods have been developed for modal logics. It has been shown [Gie98] that a synthesis between a non-clausal free variable tableau and a sequent calculus is possible which is suitable for interactive *and* automated theorem proving.

Automated deduction for the KeY system is thus going to be based on a non-clausal free variable tableau calculus. Virtually all existing automated theorem provers for such a calculus work with a technique known as *iterative deepening*. This technique uses backtracking to explore a finite part of the search space in the hope of finding a proof. The explored portion of the search space is bounded by some limit on the complexity of considered proofs. If no proof with the given maximal complexity is found, the limit is raised, and proof search is restarted.

This backtracking based method is problematic in our context, because information about the part of the search space already explored is discarded every time the procedure backtracks. This means that if the procedure does not find a proof within a certain time, the current state of the prover permits no analysis of the possible reasons for the failure. It is not possible for the user to find out what went wrong by looking at a partial proof. And, more importantly, the current state of the backtracking procedure provides no basis for a counter example search which could automatically provide this information to the user.

This observation leads us to the topic of the present work, namely to develop a proof procedure for a free variable tableau calculus which does not require backtracking.

There is another incentive to study backtracking-free proof procedures for free variable tableau calculi, which is independent of the requirements of the KeY project. The standard non-clausal free variable tableau calculus is *proof confluent*. This means that any partial proof of a theorem can be extended to a complete proof. In other words, there are no ‘dead ends’ in the search space. The proof procedure cannot make any wrong decisions, in the sense that some work would need to be undone in order to find a proof. Backtracking in an iterative deepening process is only needed because of dead ends artificially introduced by the limit on the complexity of proofs. Connection tableaux [LS01b] for instance are *not* proof confluent. There are ways to start proof construction for a valid theorem, which will never lead to a closed proof. In that context, backtracking seems a natural choice. But for a proof confluent calculus, it should not be necessary.

An analysis of incremental closure proof search for the standard proof confluent tableau calculus shows that backtracking not only seems unnatural, but that it is also inefficient. Backtracking means that previously collected information is discarded. In a typical proof confluent calculus, most rule applications are independent of each other. This means that when a step of the proof construction is undone via backtracking, most of the discarded information will later be regenerated in exactly the same way. This repeated application of the same proof steps makes the backtracking procedure inefficient.

We have now presented two reasons to investigate proof search without backtracking for free variable tableaux. At this point, we should become a little more precise. We are interested in proof search for classical first order predicate logic. Validity of formulae in this logic is only semi-decidable. If an automated theorem prover terminates, establishing the validity of a formula, we can consider it to have found a proof of that formula, whatever the internal mechanisms of the prover. If it does not terminate, it goes through an infinite sequence of internal states, which can be interpreted as an *enumeration of proof attempts*. The best any prover for first order logic can do is to enumerate proof attempts, until it eventually finds a valid proof for its input formula. Backtracking is just one possible technique of organizing an enumeration. There is nothing inherently wrong with backtracking. We thus reformulate our goal as follows:

> Find a procedure to enumerate partial tableau proofs without discarding any information that could later be useful.

By being useful, we mean both useful for the procedure itself, and useful for an analysis if the procedure is interrupted.

We will present our results only from the perspective of automated proof search. We rely on the work in [Gie98] for the integration of the proposed methods with an interactive prover.

1.1 Scientific Contributions

The central scientific contribution of this thesis is the *incremental closure* technique, which is used to define a proof procedure for free variable tableaux without backtracking. We show the practicality of our approach by an experimental comparison with a procedure using backtracking and iterative deepening. We also show that many of the refinements previously known for backtracking procedures are compatible with our approach.

A further contribution consists in a family of powerful simplification rules which can be used to subsume some of the most successful redundancy elimination techniques employed in state-of-the-art theorem provers. These rules can be applied in the incremental closure framework, but also in a backtracking prover.

Finally, a method for handling equality in free variable tableaux, based on ordered rewriting is presented. We introduce a method for proving completeness of such calculi which leads to shorter completeness proofs than previous approaches, and which permits the adaptation of results from resolution saturation procedures. Like for the simplification rules, our results are also applicable for backtracking proof procedures.

1.2 Structure

The remainder of this work is structured as follows.

- We will need some standard terminology and notation, which is introduced in Chapter 2.
- There are some existing approaches to proof search without backtracking. We will review these in Chapter 3.
- In Chapter 4, we introduce the incremental closure technique, which is the main contribution of this thesis.
- Chapter 5 describes a number of possible refinements, some of them specific to the incremental closure approach, and some adopted from backtracking procedures.
- A family of simplification rules is presented in Chapter 6.
- We show how to integrate equality handling into our framework in Chapter 7.
- Some areas for future research are pointed out in Chapter 8.
- We conclude with a short summary of our results in Chapter 9.

2 Preliminaries

Before we discuss existing approaches to our problem of finding a backtracking-free tableau proof procedure, we introduce some standard terminology and notations. We will define the *syntactic* entities needed for our discussion. We shall not give a definition of the *semantics* of first order logic, as this definition is standard, and we are not going to use it explicitly in our proofs.[1] We will also assume that the reader has a basic knowledge of analytic tableaux, and standard backtracking proof procedures. Fitting's textbook [Fit96] is a good introduction to the field.

Definition 2.1 *A* first order signature *is a quadruple* $(\mathrm{Prd}, \mathrm{Fun}, \mathrm{Var}, \alpha)$*, where* Prd *is a set of* predicate symbols*,* Fun *a set of* function symbols*,* Var *a set of* variables*, and* $\alpha : (\mathrm{Prd} \cup \mathrm{Fun}) \to \mathbb{N}$ *a function assigning an* arity *to each function and predicate symbol.* Prd*,* Fun*, and* Var *are required to be pairwise disjoint. Function symbols with arity* 0 *are also called* constant symbols*, and predicate symbols with arity* 0 *are called* propositional constants*.*

In the following, we assume a fixed first order signature with *an infinite set of variables* Var. These will be needed, because our calculi can require the introduction of new variables. Having an infinite supply, we won't need to extend the signature later.

Definition 2.2 *The set* $\mathrm{Trm}(V)$ *of* terms with variables in V *over a given signature, with* $V \subseteq \mathrm{Var}$ *is defined to be the least set with* $V \subseteq \mathrm{Trm}(V)$*, and* $f(t_1, \ldots, t_{\alpha(f)}) \in \mathrm{Trm}(V)$ *for any* $f \in \mathrm{Fun}$ *and* $t_1, \ldots, t_{\alpha(f)} \in \mathrm{Trm}(V)$*. The set* Trm *of all terms is defined as* $\mathrm{Trm} := \mathrm{Trm}(\mathrm{Var})$*. The set* Trm^0 *of* ground terms *is defined as* $\mathrm{Trm}^0 := \mathrm{Trm}(\emptyset)$*.*

The set $\mathrm{Fml}(V)$ *of* formulae with free variables in V*, with* $V \subseteq \mathrm{Var}$ *is defined to be the least set with*

- $\mathit{true}, \mathit{false} \in \mathrm{Fml}(V)$*,*
- $p(t_1, \ldots, t_{\alpha(p)}) \in \mathrm{Fml}(V)$ *for any* $p \in \mathrm{Prd}$ *and* $t_1, \ldots, t_{\alpha(p)} \in \mathrm{Trm}(V)$*,*
- $\neg\phi \in \mathrm{Fml}(V)$ *for any* $\phi \in \mathrm{Fml}(V)$*,*
- $\phi \wedge \psi$*,* $\phi \vee \psi \in \mathrm{Fml}(V)$ *for any* $\phi, \psi \in \mathrm{Fml}(V)$*,*
- $\forall x.\phi$*,* $\exists x.\phi \in \mathrm{Fml}(V)$ *for any* $\phi \in \mathrm{Fml}(V \cup \{x\})$*, and* $x \in \mathrm{Var}$*.*

The set Fml *of all formulae is defined as* $\mathrm{Fml} := \mathrm{Fml}(\mathrm{Var})$*. The set* Fml^0 *of* closed formulae *is defined as* $\mathrm{Fml}^0 := \mathrm{Fml}(\emptyset)$*.*

[1] We *will* need some slightly non-standard semantic notions in Chapter 7, so appropriate definitions will be given there.

Note that the set Trm^0 is non-empty only if the signature contains at least one constant symbol. We shall thus assume that this is the case for our fixed signature.

Definition 2.3 *A* substitution *is a mapping* $\mu : \mathrm{Var} \to \mathrm{Trm}$*, such that* $\mu(x) = x$ *for all but finitely many* x*. We define the* domain *of a substitution as* $\mathrm{dom}(\mu) := \{x \in \mathrm{Var} \mid \mu(x) \neq x\}$*. Substitutions are extended to homomorphisms of terms and formulae in the usual way. Composition of substitutions is the same as functional composition and is denoted by* $\circ$*:* $(\mu \circ \nu)(t) = \mu(\nu(t))$*. A substitution* μ *is* idempotent *if* $\mu \circ \mu = \mu$*. A* ground substitution *is a substitution* μ *with* $\mu(x) \in \mathrm{Trm}^0$ *for all* $x \in \mathrm{dom}(\mu)$*. We shall use the notation* $\mu = [x_1/t_1, \ldots, x_n/t_n]$ *to describe the concrete unification with* $\mathrm{dom}(\mu) = \{x_1, \ldots, x_n\}$ *and* $\mu(x_i) = t_i$*.*

In most of the following work, we are going to simplify the presentation by *not* using arbitrary first order formulae, but working on a certain normal form instead.

Definition 2.4 *A formula is in* negation normal form *(NNF), iff negation signs appear only in front of atomic formulae* $p(t_1, \ldots, t_n)$*. A formula is in* skolemized negation normal form *(SNNF), iff it is in NNF and does not contain* $\exists$ *quantifiers.*

We rely on the following well-known result:

Proposition 2.5 *Any closed formula* ϕ *can be transformed by application of de Morgan's rules and skolemization into a formula* ϕ' *in SNNF that is satisfiable iff* ϕ *is satisfiable.*

This fact alone is not a sufficient justification for our restriction to a normal form, given the context of our work outlined in Chapter 1. After all, the same is true for clause normal form. We use SNNF because results for SNNF can easily be generalized to full first-order logic. Full first order logic only adds arbitrary use of negation and existential quantifiers. Negation is invariably handled in proofs and calculi for full first order formulae using duality: The part of a proof or calculus for handling $\neg(\phi \vee \psi)$ for instance, corresponds exactly to that for $(\neg\phi)\wedge(\neg\psi)$, and similarly for the other connectives. Using SNNF means that such applications of de Morgan's rules are performed in advance, and don't clutter our proofs. Existential quantification may be handled in tableau calculi by one of various skolemization rules [HS94, BHS93, BF95, CNA98, GA99]. Although skolemization is usually formalized as a rule to be applied in the course of tableau construction, it is possible in general to factor out skolemization from proof search, and do it in advance, like for negation. This removes further ballast from our proofs. To summarize, presenting our results for SNNF only makes definitions and proofs simpler without any loss in generality.

These are all the standard syntactic notions and notations we need for the time being. Any further terminology will be introduced later on, as and when it is needed.

3 Existing Approaches

We shall describe some existing ideas for backtracking-free tableau proof procedures in this chapter. In particular, we shall point out for each of these, why they are not suitable for our application domain, as described in Chapter 1. It cannot be the aim of this chapter to give a detailed presentation of the various approaches, so the reader will have to refer to the given literature for the exact definitions. In particular, we have to assume that the reader is familiar with the standard free variable tableau calculus as presented e.g. in [Fit96].

3.1 Smullyan Style Tableaux

One of the first formulations of the tableaux method by Smullyan [Smu68] directly supports proof search without backtracking. This is due to the use of a different rule for universal quantifiers than that of free variable tableaux.

In a Smullyan style tableau, a formula $\forall x.\phi$ gets handled by putting formulae $[x/t]\phi$ on the tableau for ground terms t. By contrast, in free variable tableaux, one introduces a formula $[x/X]\phi$, where X is a new free variable. This will be covered in detail in the following chapter, see Def. 4.2. The main point is that the introduced variable is later going to be instantiated by a ground term. But the instantiation is delayed until a useful instantiation is found. It is essentially the process of finding the right instantiations which introduces backtracking into free variable tableau procedures.

The problem with Smullyan style tableaux is that the term t has to be guessed. As an automated theorem prover cannot be expected to guess the right ground term, it would have to enumerate all ground terms and apply the tableau extension for each of them. This can be done, and it yields a proof procedure that works without backtracking. But that proof procedure is very inefficient in practice, because the enumeration might take a long time until the correct instantiation is found.

This is of course the reason why we insisted on a backtracking-free proof procedure for *free variable* tableaux in Chapter 1. Tableaux without free variables are just too inefficient.

3.2 Disconnection and Related Calculi

Billon [Bil96] proposed a method for proof search without backtracking in tableau-like calculi, known as the *disconnection calculus.* Billon gives a very generic description, which can also be applied to matrix based methods. For a tableau calculus the main idea is as follows. A (clausal) tableau is built as usual, introducing free variables, until a branch closure becomes possible under some instantiation for the free variables. Instead of setting a backtracking choice point and applying a unification for the free variables, one generates copies of the clauses the complementary literals come from, and applies the unifier on *them*. These instantiated clauses are then used to expand the proof tree. In other words, free variables on the tableau are never instantiated, but one generates the required instances. A proof is found, when all branches of the tableau could be closed by instantiating all free variables with the same (arbitrary) constant symbol. Recently, Letz and Stenz [LS01a, LS02, Ste02] have further investigated the disunification calculus and implemented an automated theorem prover which uses it.

Baumgartner, Eisinger and Furbach [BEF99, BEF00] have defined the *Confluent Connection Calculus (CCC)*, which is in many ways similar to the disconnection calculus. The criteria for branch closure and the organization of clause copying are slightly different, but the principle of constructing copies and instantiating them is the same. More recently, Baumgartner [Bau00] and Baumgartner and Tinelli [BT03] have extended these ideas to create first order versions of the Davis-Putnam-Logemann-Loveland procedure.

Although these calculi are a promising basis for an automated theorem prover, they do not fit our requirements, as laid down in Chapter 1. First, they are only formulated for problems given in clausal form. It would probably be possible, but not easy, to develop non-clausal versions. More importantly, the way instantiation works is quite far from the way humans construct proofs. While an integration with a calculus suitable for human use is feasible for the standard free variable tableau calculus [Gie98], it is not clear how this could be done for the disconnection calculus or CCC. In particular, if one considers equality handling, which is important in our application domain, the necessary modifications for the disconnection calculus [Bil96, LS02] are rather unintuitive. For instance, an application of a rewriting step can lead to the generation of new branches. Finally, it is not clear how the calculi would have to be changed to accommodate problems in which modal or dynamic logic formulae might occur.

To summarize, the calculi described in this section, though suitable for pure automated theorem proving, are inadequate for our goals, because they are not compatible with dynamic logic, and because they are not suitable for interactive use.

3.3 Strong Fairness Conditions

A proof procedure has been described by Beckert [Bec98, Bec03] which, in a sense, takes the opposite approach of that described in the previous section. A free variable tableau is again extended until a branch closure becomes possible. But now, the corresponding instantiation is applied, changing some of the formulae on the tableau. A *reconstruction step* follows to reintroduce literals which were destroyed by the instantiation. To prevent the procedure from going into cycles, Beckert defines a tableau subsumption relation. Any proof step that leads to a tableau which is subsumed by an earlier tableau is forbidden. Together with some more fairness restrictions, one obtains a complete depth-first search procedure.

This procedure works on a standard free variable calculus. Closing branches actually leads to the instantiation of free variables. Also, Beckert has defined the procedure for full first order logic, and not only for clause tableaux. This makes it a good candidate for the application scenario outlined in Chapter 1.

We chose not to use this approach, mainly because of its conceptual complexity. In particular, the tableau subsumption relation is not easy to understand. There are unexpected and not well-understood phenomena, like the need to apply reconstruction steps on branches which are already closed. No implementation of the procedure has yet been attempted, but it is clear that the efficiency of an implementation depends on how efficiently the tableau subsumption relation can be tested. One should also not forget that an implementation of a new proof procedure is normally not very powerful. Various refinements have to be incorporated into the procedure to reduce redundancy. If the basic procedure is already hard to understand, it will be very difficult to extend it. We shall see in the following chapters that our approach is easily adapted to use various refinements known from backtracking procedures.

3.4 Delayed Closure Approaches

In this section, we shall mention two approaches which are close to the incremental closure technique we are going to present in the next chapter. Both are based on the idea of expanding a tableau without applying instantiations to close branches. Instead, after a number of expansions steps, one tests whether an instantiation can be found that closes all branches simultaneously. We call this approach 'delayed closure', because closure of single branches is delayed to the point where all branches can be closed. We will give a detailed introduction to tableaux with delayed closure in Sections 4.1 and 4.2.

A proof procedure based on delayed closure, including a Prolog implementation, is given in Sect. 7.5 of Fitting's textbook [Fit96]. That implementation is only of didactic value however, as the way the global closure test is implemented leads to severe inefficiency, see the footnote on page 21.

Another method based on delayed closure has been presented by Konev and Jebelean [KJ00]. Their ideas are very close to ours, so we will defer a detailed comparison to Sect. 4.8, when we have presented our approach.

This concludes our general discussion of existing approaches for proof search without backtracking in tableau procedures. Our work is of course related to that of other researchers in many more ways than could be covered here. The following chapters will contain sections describing further related work.

4 The Incremental Closure Approach

4.1 Block Tableaux with Delayed Closure

Although this chapter is mainly concerned with the definition of a *proof procedure* and not a *calculus*, it is useful to use a slightly non-standard formulation as a basis for the proof procedure.

Free variable tableaux are usually defined as trees, the nodes being labeled with formulae. One then talks about closing branches of the tableau by unifying literals that lie one some branch, that is on some path between a leaf and the root of the tableau.

However, it is sometimes useful to change a formula on a branch or even to delete it. Such an operation is usually local to one branch and should not be visible on the other ones. This cannot be expressed in the usual formulation of tableaux. For that reason, we shall use what Smullyan [Smu68] calls *block tableaux*: nodes of the tableau are labeled with finite sets of formulae and only the formulae at a leaf node are considered for tableau expansion and closure.

The other deviation from the usual formulation is that we shall use *no closure rule*. This rule normally states that a branch of a tableau may be closed by applying the most general unifier of some complementary pair of literals on that branch to the whole tableau. Instead, we shall try to find a single unifier that closes all branches of a tableau simultaneously, for reasons which will become apparent later in this chapter. We call this approach *delayed closure*.

We now formally define our concept of free variable tableaux for formulae in skolemized negation normal form (SNNF), see Def. 2.4.

Definition 4.1 *An* instantiation *is a mapping* $\sigma : \mathrm{Var} \to \mathrm{Trm}^0$ *from the set of all variables to ground terms. Let* $\mathcal{I}$ *denote the set of all instantiations.*

This differs from the usual concept of a ground substitution, see Def. 2.3, in that we require *all*, i.e. potentially infinitely many variables to be mapped. Although any given tableau can contain only a finite number of free variables at a time, this concept will simplify some of our definitions because we do not need to keep track of them.

Definition 4.2 *A* goal *is a finite set of formulae. A* tableau *is a finite tree where each node is labeled with a goal. A* leaf *is a node with no children. The* leaf goals *of a tableau are the goals that label its leaves.*

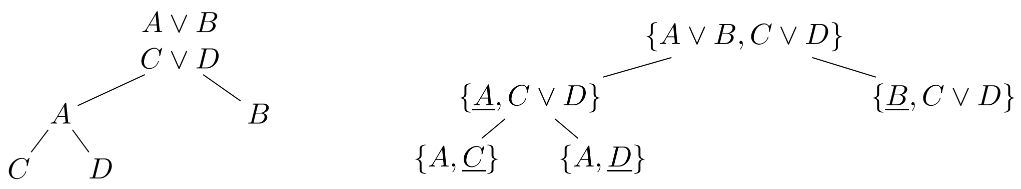


Figure 4.1: Standard Tableaux vs. Block Tableaux.

A node n is said to be above *a node n' in a given tableau, if n is on the path between n' and the root, but $n \neq n'$. Conversely, n is* below *n' if n' is above n.*

A tableau for a finite set of SNNF formulae *S is defined inductively as follows:*

1. *The tableau consisting of the root node labeled with the goal S is a tableau for S, called the* initial tableau*.*

2. *If there is a tableau for S that has a leaf n with goal $\{\alpha_1 \land \alpha_2\} \cup G$, then the tableau obtained by adding a new child n' with goal $\{\alpha_1, \alpha_2\} \cup G$ to n is also a tableau for S. (α-expansion)*

3. *If there is a tableau for S that has a leaf n with goal $\{\beta_1 \lor \beta_2\} \cup G$, then the tableau obtained by adding two new children n', resp. n'' with goals $\{\beta_1\} \cup G$, resp. $\{\beta_2\} \cup G$ to n is also a tableau for S. (β-expansion)*

4. *If there is a tableau for S that has a leaf n with goal $\{\forall x.\gamma_1\} \cup G$, then the tableau obtained by adding a new child n' with goal $\{[x/X]\gamma_1, \forall x.\gamma_1\} \cup G$ to n, where X did not previously occur in the tableau, is also a tableau for S. (γ-expansion)*

A complementary pair *is a pair ϕ, $\neg\psi$, where ϕ and ψ are unifiable atomic formulae. A goal G is* closed under an instantiation *σ, iff there is a complementary pair $\{\phi, \neg\psi\} \subseteq G$ with $\sigma(\phi) = \sigma(\psi)$. A tableau T is* closed under an instantiation *σ, iff each leaf goal of T is closed under σ. A tableau is* closable *iff it is closed under some instantiation.*

Fig. 4.1 illustrates the difference between standard and block tableaux. In the block tableau, the information that the formula $C \lor D$ is consumed on the two leftmost branches but not on the right one is made explicit. Obviously, block tableaux are a bit bulkier to write down, so we shall underline the new formula in each goal. Also, we shall sometimes give examples in standard tableau notation, if the added precision of the block tableau formulation is not needed.

Example 4.3 As an example for a first order tableau, see Fig. 4.2. We follow the convention of denoting free variables like X by capital letters. The left leaf goal is obviously closed under an instantiation σ exactly if $\sigma(X) = a$, while the right leaf goal is closed if $\sigma(X) = b$. Both conditions cannot be met simultaneously, so the tableau is not closed.

$$\begin{tikzcd}
 & \{\forall x.(px \vee qx), \neg pa, \neg qb\} \arrow[d, dash] & \\
 & \{\underline{pX \vee qX}, \neg pa, \neg qb, \forall x.(px \vee qx)\} \arrow[dl, dash] \arrow[dr, dash] & \\
\underline{pX}, \neg pa, \neg qb, \forall x.(px \vee qx) & & \underline{qX}, \neg pa, \neg qb, \forall x.(px \vee qx)
\end{tikzcd}$$

Figure 4.2: A first order tableau with two branches.

It is obvious that the usual soundness and completeness proofs for free variable tableaux are also applicable to this formulation.

Proposition 4.4 *Let S be a set of closed formulae in SNNF. S is unsatisfiable iff there is a closable tableau for S.*

At this point, we should say a few words about the 'free variables' of the free variable tableau calculus. The distinction between free and bound variables is usually needed in the definition of model based semantics for first order logic. The interpretation of a first order formula depends on a carrier set $\mathcal{D}$, an interpretation function $\mathcal{I}$ for the function and predicate symbols of the signature, and a *variable assignment* β. The latter is a function which maps every variable to an element of the domain. It is used to interpret formulae and terms containing free variables. The need for this variable assignment arises when one defines the validity of quantified formulae in a model. For instance, one defines:

> A formula $\forall x.\phi$ is valid under $\mathcal{D}$, $\mathcal{I}$, β, iff ϕ is valid under $\mathcal{D}$, $\mathcal{I}$, β_x^a for every $a \in \mathcal{D}$, where
> $$\beta_x^a(y) := \begin{cases} a & \text{if } x = y, \\ \beta(y) & \text{otherwise.} \end{cases}$$

In a deduction context, it is sufficient to restrict oneself to problems formulated as closed formulae, i.e. formulae in which all variables are bound by quantifiers. Although variable assignments permit the definition of semantics of formulae with free variables, their main use is in the recursive definition of the semantics of quantified formulae. In this context, it is natural to evaluate variables by mapping them to elements of the domain.

In free variable tableau calculi, new free variables are introduced by γ rules. In contrast to their use in the definition of model semantics, these free variables are not only free in some currently considered subformula. They are free in the formulae on the tableau branch. Also, as these variables are new, they will *never* be quantified in any context. During tableau construction, the free variables are instantiated by unification when one or more branches are closed. This, as part of a calculus, is a syntactic manipulation. The variables are instantiated to *terms*, and not to elements of the domain.

It is customary to use variable assignments for the free variables of the tableau when one shows soundness of the calculus. It turns out however that the soundness proof actually becomes simpler if one works with ground substitutions instead of variable assignments. Such soundness proofs have been used successfully in [Gie98], for instance. This means that it is never necessary to instantiate the free variables introduced by γ rules using a variable assignment.

Due to this fact, it is possible to introduce a new syntactic category for the free variables of the tableau calculus, giving a formal basis for our convention of using capital letters for the free variables. This was done in [Gie98], where the free variables of the calculus are called *metavariables*, while the usual variables are referred to as *logic variables*. Logic variables are always bound, and they are interpreted using variable assignments; metavariables are never bound, and they are interpreted using ground substitutions or instantiations. This distinction helps to make soundness and completeness proofs simpler and more transparent.

For this work however, we have decided to use the usual name of ‘free variables’. The reason for this is that the reader accustomed to the usual nomenclature would probably be confused by the use of a new name, and this work contains no proofs where the distinction would be of much value. In nearly every reference in this book to a ‘free variable’, we mean a free variable in the sense of the free variable tableau calculus. The reader might notice however that we will *always* instantiate free variables to ground terms before interpreting them.

4.2 A Proof Procedure for Delayed Closure

From Prop. 4.4, it is easy to derive a complete proof procedure:

```
T := initial tableau for S
while ( not closable(T) ) do
   if expandable(T) then
      select possible expansion of T
      expand T
   else
      answer ' satisfiable '
   end
end
answer ' unsatisfiable '
```

This is a complete proof procedure, provided the selection of tableau expansions is *fair*. Being fair means the same thing as in standard tableaux, namely that if the procedure does not terminate, any extension step possible on a goal will at some point be applied on that goal or one of its descendants. In particular, in a non-terminating run, infinitely many instances of each γ-formula will ultimately be produced on each branch.

The main problem with this proof procedure is the test closable(T): In general, there may be several complimentary literals in each leaf goal, leading to different closing instantiations. The right combination of complementary literals has to be found in the leaf goals to close all goals simultaneously.

Theorem 4.5 *Deciding whether a tableau is closable is NP-complete in the total size of the literals in the leaf goals.*

Proof. Unifiability can be decided in linear time [PW78], so with indeterministic selection of complementary pairs, closable(T) is in NP. On the other hand, SAT can be reduced to this problem as follows. We take a signature containing two constant symbols T and F and a number of predicate symbols: For every propositional symbol A of the SAT problem, introduce a predicate symbol p_A, as well as a free variable X_A. Translate every clause C of the SAT problem to one leaf goal G of the tableau by

$$G := \bigcup_{A \in C} \{\neg p_A(T), p_A(X_A)\} \cup \bigcup_{\neg A \in C} \{\neg p_A(F), p_A(X_A)\} \quad ,$$

e.g., $A \vee \neg B$ is translated to $\{\neg p_A(T), p_A(X_A), \neg p_B(F), p_B(X_B)\}$. Closing instantiations of these goals are easily seen to correspond to truth value assignments of the propositional symbols of the SAT problem that make the corresponding clauses true. Accordingly, the set of goals generated from a set of clauses can be closed simultaneously if and only if the set of clauses is satisfiable. □

This means that with the proof procedure outlined above, an NP-complete problem has to be solved after each proof step. Of course, such an asymptotic complexity may or may not be a problem in practice, depending on the nature of the problems and the possibilities for optimizing a solver. But if the closure test is implemented in the straightforward fashion of going through every combination of complementary literals and checking for simultaneous unifiability, the complexity does turn out to be prohibitive in practice. Even for small problems, a tableau might easily have several dozen branches, and there are often several different combinations of complementary literals on each branch, leading to an exponential behaviour of the outlined algorithm.[1]

Does this mean that the given proof procedure is unpractical? Not necessarily. The problem of finding the right complementary pairs has to be solved in any free variable tableau proof procedure, backtracking or not. In a backtracking prover, this search is implicit in the proof procedure. If the global closure test is made explicit, one has the opportunity to make it more efficient in practical cases.

Our technique for doing this is the *Incremental Closure* approach, which was first introduced in [Gie01], and which shall be presented here in greater detail. It is based on the given proof procedure, but it uses a better closure test: We compute closable(T) in an incremental fashion, based on the following observations:

- If a pair of complementary literals present on a branch is unifiable, it will stay unifiable after any extension of that branch. Even if an extension step introduces new branches, they can still all be closed by taking those complementary literals. For instance, a goal

$$\{pX, \neg pY, q \vee r, \ldots\}$$

[1] This is the problem of the proof procedure given by Fitting, which we mentioned in Sect. 3.4. That procedure fully expands a tableau up to a given complexity limit, and then tries to close all branches by going through every combination of complementary literals.

may be closed by any instantiation that assigns the same ground term to X and Y. After application of a β-expansion, we obtain

$$\begin{tikzcd}
 & \{pX, \neg pY, q \vee r, \ldots\} \arrow[dl, dash] \arrow[dr, dash] & \\
\{pX, \neg pY, q, \ldots\} & & \{pX, \neg pY, r, \ldots\}
\end{tikzcd}$$

and both goals are still closable by those instantiations. This is due to the fact that Def. 4.2 ensures that literals are never removed from a goal by a tableau expansion. It is this monotonicity property that makes an incremental closure test seem worthwhile.

- The closure test has to find an instantiation for the free variables introduced by γ-expansions. But a free variable occurs in the proof tree only below the node in which it has been introduced. In general, this does not mean that the instantiations of free variables X, Y introduced in disjoint subtableaux are independent: both might be linked to the instantiation of a third variable Z present in both subtableaux. But still, free variables enjoy a certain locality which should be exploited in our algorithm. To do this, the closure computation shall follow the structure of the proof tree.

4.3 Abstract View

In this section, we will explain the Incremental Closure approach on an abstract level. In particular, we shall abstract away from concrete representations of instantiations, and assume that we can perform calculations on (potentially infinite) sets of instantiations. How to represent these in an actual implementation is discussed in Sect. 4.4.2.

Definition 4.6 *Let*

$$\operatorname{unif}(\phi, \psi) := \{\sigma \in \mathcal{I} \mid \sigma(\phi) = \sigma(\psi)\}$$

be the set of instantiations that unify two atomic formulae. We define the closer set of G

$$\operatorname{cl}_0(G) := \bigcup_{\phi, \neg\psi \in G} \operatorname{unif}(\phi, \psi)$$

to be the set of instantiations under which a goal G is closed. For a node n of a tableau, let $lg(n)$ be the set of leaf goals associated with the leaves that are descendants of n. Use this to define the closer set of n

$$\operatorname{cl}_0(n) := \bigcap_{G \in lg(n)} \operatorname{cl}_0(G)$$

to be the set of instantiations under which all leaves below n are closed.

It is obvious from the definitions that closability of tableaux may be characterized as follows:

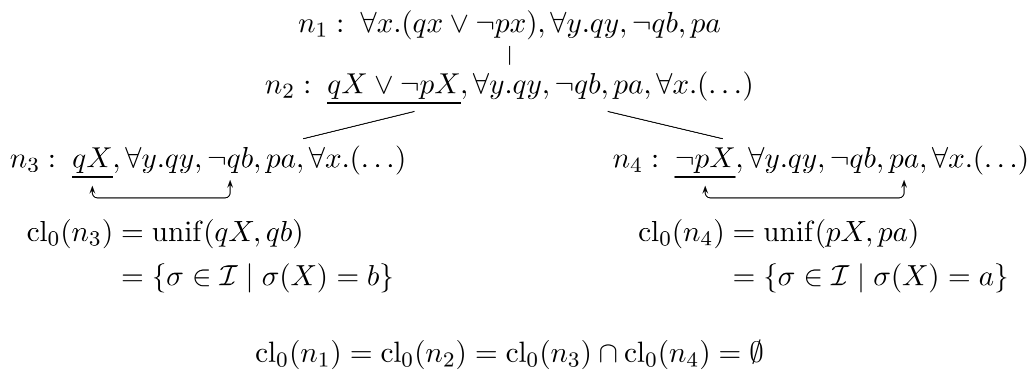

$n_1 : \forall x.(qx \vee \neg px), \forall y.qy, \neg qb, pa$
$n_2 : \underline{qX \vee \neg pX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$
$n_3 : \underline{qX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$
$n_4 : \underline{\neg pX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$
$\mathrm{cl}_0(n_3) = \mathrm{unif}(qX, qb) = \{\sigma \in \mathcal{I} \mid \sigma(X) = b\}$
$\mathrm{cl}_0(n_4) = \mathrm{unif}(pX, pa) = \{\sigma \in \mathcal{I} \mid \sigma(X) = a\}$
$\mathrm{cl}_0(n_1) = \mathrm{cl}_0(n_2) = \mathrm{cl}_0(n_3) \cap \mathrm{cl}_0(n_4) = \emptyset$

Figure 4.3: An open tableau with closer sets $\mathrm{cl}_0(n)$.

Proposition 4.7 *If root is the root node of the tableau, then* $\mathrm{cl}_0(\mathit{root})$ *is the set of instantiations that close the whole tableau. A tableau is closable iff* $\mathrm{cl}_0(\mathit{root}) \neq \emptyset$.

Example 4.8 We shall clarify these notions using the tableau in figure Fig. 4.3. n_2 was constructed by applying a γ-expansion at n_1, and n_3, n_4 were introduced by a β-expansion at n_2. The newly introduced formulae are underlined in each goal.

The goal at n_3 contains only one complementary pair $qX, \neg qb$. So $\mathrm{cl}_0(n_3) = \mathrm{unif}(qX, qb) = \{\sigma \in \mathcal{I} \mid \sigma(X) = b\}$, the set of instantiations that map X to b. Similarly, $\mathrm{cl}_0(n_4) = \{\sigma \in \mathcal{I} \mid \sigma(X) = a\}$, because of the complementary pair $\neg pX, pa$. For $\mathrm{cl}_0(n_2)$ we have to find instantiations that close both leaf goals, $\mathrm{cl}_0(n_2) = \mathrm{cl}_0(n_3) \cap \mathrm{cl}_0(n_4)$. There are obviously no such instantiations, $\mathrm{cl}_0(n_2) = \emptyset$. The same holds for the root n_1, of course.

The instantiations contained in the sets $\mathrm{cl}_0(n)$ refer to all free variables, including those that are introduced only below n in the tableau. It is sufficient to restrict the instantiations to variables introduced above n. We need a few notions to formalize this.

Definition 4.9 *An* instantiation for a set of variables V *is a mapping from* V *to the set of ground terms. Let* $\mathcal{I}(V)$ *denote the set of all instantiations for* V*. In particular,* $\mathcal{I} = \mathcal{I}(\mathrm{Var})$.

Let $\square$ *be the* empty instantiation*, which is the single instantiation for an empty set of variables,* $\{\square\} = \mathcal{I}(\emptyset)$.

Given an instantiation $\sigma \in \mathcal{I}(V)$ *and a set of variables* $W \subseteq V$*, let* $\sigma|_W \in \mathcal{I}(W)$ *be the restriction of* σ *to* W*. For a variable* $X \in V$*, we shall write* $\sigma|^X := \sigma|_{V \setminus \{X\}}$ *to exclude* X *from the domain of* σ.

We extend restriction to sets of instantiations as follows: For a set $I \subseteq \mathcal{I}(V)$ *and* $W \subseteq V$*, let* $I|_W := \{\sigma|_W \mid \sigma \in I\}$*, and similarly* $I|^X := \{\sigma|^X \mid \sigma \in I\}$ *for* $X \in V$.

The empty instantiation $\square$, which is the only element of $\mathcal{I}(\emptyset)$, is a somewhat peculiar object. We will use elements of $\mathcal{I}(V)$ to denote closing instantiations in situations, where only the variables in V are relevant, in the sense that we are not interested in the instantiation of other variables. In particular, this means that an implementation could discard the information about the actual instantiations of irrelevant variables to save space. Now for the closure of the *whole* tableau, none of the free variables is relevant. It is sufficient that there is some instantiation for them. Hence our interest in the set $\mathcal{I}(\emptyset)$. One should keep in mind that there are exactly two *sets* of instantiations for an empty set of variables, namely $\emptyset$ meaning that there is no closing instantiation (yet), and $\{\Box\}$, meaning that there is one.

We can now follow up on Def. 4.6 by restricting the sets of closing instantiations to the free variables of interest.

Definition 4.10 *Let* $\mathrm{fv}(n)$ *be the set of free variables introduced at* γ*-expansions on nodes above some node* n.[2]

This set is empty for the root of the tableau.

We use this to define the restricted closer set of a node n *as*

$$\mathrm{cl}(n) := \mathrm{cl}_0(n)|_{\mathrm{fv}(n)} \quad .$$

The following fact is an immediate consequence of this definition and Prop. 4.7:

Proposition 4.11 *If root is the root node of the tableau, then a tableau is closable iff* $\mathrm{cl}(\mathit{root}) \neq \emptyset$*, or equivalently iff* $\mathrm{cl}(\mathit{root}) = \{\Box\}$.

In order to take the structure of the tableau into account, as announced in the previous section, we will now sate how $\mathrm{cl}(n)$ may be calculated recursively from the values for the nodes below n.

Lemma 4.12 *Let* n *be a node of a tableau.*

- *If* n *is a leaf labeled with goal* G*, then*

$$\mathrm{cl}(n) = \mathrm{cl}_0(G)|_{\mathrm{fv}(n)} \quad .$$

- *If an* α*-expansion is applied on* n*, leading to a child* n'*, then*

$$\mathrm{cl}(n) = \mathrm{cl}(n') \quad .$$

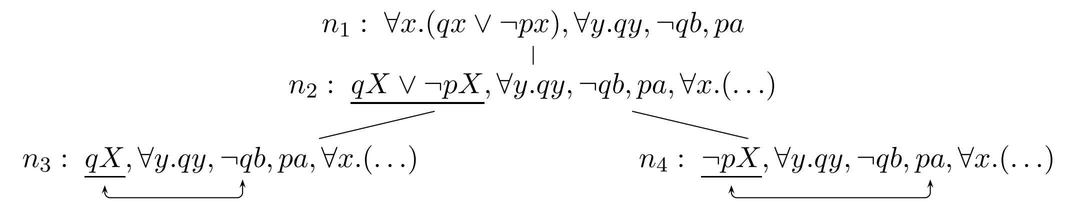
- *If a* β*-expansion is applied on* n*, leading to children* n' *and* n''*, then*

$$\mathrm{cl}(n) = \mathrm{cl}(n') \cap \mathrm{cl}(n'') \quad .$$

- *If a* γ*-expansion is applied on* n*, leading to a child* n' *with a new free variable* X*, then*

$$\mathrm{cl}(n) = \mathrm{cl}(n')|^{X} \quad .$$

[2]This implies that a free variable introduced by a γ-expansion *on* the node n will not be included in $\mathrm{fv}(n)$.


$n_1 :\ \forall x.(qx \vee \neg px), \forall y.qy, \neg qb, pa$
$n_2 :\ qX \vee \neg pX, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$
$n_3 :\ qX, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$
$n_4 :\ \neg pX, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$

$$\begin{aligned}\mathrm{cl}(n_3) &= \mathrm{unif}(qX, qb)|_{\{X\}} \\ &= \{\sigma \in \mathcal{I}(\{X\}) \mid \sigma(X) = b\}\end{aligned} \qquad \begin{aligned}\mathrm{cl}(n_4) &= \mathrm{unif}(pX, pa)|_{\{X\}} \\ &= \{\sigma \in \mathcal{I}(\{X\}) \mid \sigma(X) = a\}\end{aligned}$$

$$\mathrm{cl}(n_2) = \mathrm{cl}_0(n_3) \cap \mathrm{cl}_0(n_4) = \emptyset$$

$$\mathrm{cl}_0(n_1) = \mathrm{cl}_0(n_2)|^X = \emptyset$$

Figure 4.4: Recursive computation of restricted closer sets $\mathrm{cl}(n)$.

Proof. The cases for leaf goals and α-expansions follow immediately from the definitions.

For the β case, let $V := \mathrm{fv}(n) = \mathrm{fv}(n') = \mathrm{fv}(n'')$. To show the '$\subseteq$' direction, let $\sigma \in \mathrm{cl}(n) = \mathrm{cl}_0(n)|_V$. Then there exists $\sigma_0 \in \mathrm{cl}_0(n)$ with $\sigma_0|_V = \sigma$. Now as $\mathrm{lg}(n) = \mathrm{lg}(n') \cup \mathrm{lg}(n'')$, we have $\sigma_0 \in \mathrm{cl}_0(n) = \mathrm{cl}_0(n') \cap \mathrm{cl}_0(n'')$, so $\sigma \in \mathrm{cl}_0(n')|_V = \mathrm{cl}(n')$, and similarly $\sigma \in \mathrm{cl}(n'')$. For the '$\supseteq$' direction, let $\sigma \in \mathrm{cl}(n') \cap \mathrm{cl}(n'') = \mathrm{cl}_0(n')|_V \cap \mathrm{cl}_0(n'')|_V$. Then there are $\sigma_1 \in \mathrm{cl}_0(n')$ and $\sigma_2 \in \mathrm{cl}_0(n'')$ with $\sigma_1|_V = \sigma_2|_V = \sigma$. Define σ_0 by

$$\sigma_0(X) := \begin{cases} \sigma(X) & \text{if } X \in V, \\ \sigma_1(X) & \text{if } X \text{ is introduced at or below } n', \\ \sigma_2(X) & \text{if } X \text{ is introduced at or below } n'', \\ \text{don't care} & \text{if } X \text{ doesn't occur in the tableau.} \end{cases}$$

Then σ_0 coincides with σ_1 for all free variables occurring in leaves under n', so $\sigma_0 \in \mathrm{cl}_0(n')$, and similarly $\sigma_0 \in \mathrm{cl}_0(n'')$, and so $\sigma_0 \in \mathrm{cl}_0(n)$. It follows that $\sigma \in \mathrm{cl}(n)$, since $\sigma_0|_V = \sigma$.

Finally, for a γ-expansion at n, we have $\mathrm{fv}(n') = \mathrm{fv}(n) \,\dot\cup\, \{X\}$ if X is the new free variable. Then,

$$\mathrm{cl}(n) = \mathrm{cl}_0(n)|_{\mathrm{fv}(n)} = (\mathrm{cl}_0(n)|_{\mathrm{fv}(n')})|_{\mathrm{fv}(n)} = \mathrm{cl}(n')|_{\mathrm{fv}(n)} = \mathrm{cl}(n')|^X \quad ,$$

as is easily verified. □

Example 4.13 Fig. 4.4 continues the previous example, giving values of $\mathrm{cl}(n)$ computed recursively according to Lemma 4.12.

To get an incremental algorithm, we now need to examine how the values of cl change when a tableau expansion produces new complementary pairs. In general, one expansion step might lead to several new complementary pairs in one goal, or there might be two new goals, each of which can contain new complementary pairs. We shall examine the changes to cl induced by *one* new complementary pair $\phi, \neg\psi$ at *one* leaf l, called the *focused leaf.* If there are several new complementary pairs, these changes must be applied consecutively for each of them.

For each node n, let $\mathrm{cl}(n)^{\mathrm{old}}$ denote the value of $\mathrm{cl}(n)$ before taking into account the new complementary pair $\phi, \neg\psi$, while $\mathrm{cl}(n)^{\mathrm{new}}$ is the updated value. As noted in the previous section, possible closing instantiations are never destroyed by an expansion step, so the sets cl can only grow when the tableau is expanded, i.e. $\mathrm{cl}(n)^{\mathrm{new}} \supseteq \mathrm{cl}(n)^{\mathrm{old}}$ for all nodes of the tableau.

Definition 4.14 *Define*

$$\delta(n) := \mathrm{cl}(n)^{\mathrm{new}} \setminus \mathrm{cl}(n)^{\mathrm{old}}$$

to be the set of new *closing instantiations for a node n that arise by taking into account a new complementary pair $\phi, \neg\psi$ at a focused leaf l.*

This δ can be calculated recursively as stated in the following Lemma.

Lemma 4.15 *Let $\delta(n)$ be a set of new closing instantiations for a node n as in Def. 4.14.*

- *If $n \neq l$, and n is not above l in the tableau, then*
$$\delta(n) = \emptyset$$
- *If $n = l$ is the focused leaf, then*
$$\delta(n) = \mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(n)} \setminus \mathrm{cl}(l)^{\mathrm{old}}$$
- *If an α-expansion is applied on n, leading to a child n', then*
$$\delta(n) = \delta(n')$$
- *If a β-expansion is applied on n, leading to children n' and n'', and l is equal to, or below n', then*
$$\delta(n) = \delta(n') \cap \mathrm{cl}(n'')^{\mathrm{old}}$$
- *If a γ-expansion is applied on n, leading to a child n' with a new free variable X, then*
$$\delta(n) = \delta(n')|^{X} \setminus \mathrm{cl}(n)^{\mathrm{old}}$$

Proof. Obviously, if the n is not at or above the focused leaf l, then $\mathrm{cl}(n)^{\mathrm{new}} = \mathrm{cl}(n)^{\mathrm{old}}$, so $\delta(n) = \emptyset$. In other words, $\delta(n)$ is non-empty at most for nodes n on the path between l and the root of the tableau. This takes care of the first case. For the focused leaf l, δ is given by

$$\delta(l) = (\mathrm{cl}(l)^{\mathrm{old}} \cup \mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(l)}) \setminus \mathrm{cl}(l)^{\mathrm{old}} \quad = \mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(l)} \setminus \mathrm{cl}(l)^{\mathrm{old}} \quad ,$$

showing the second case.

The other cases show how to 'propagate' this change up the branch towards the root. They follow from the recursive expressions for $\mathrm{cl}(n)$ given in Lemma 4.12. For α-expansions, we trivially have

$$\delta(n) = \mathrm{cl}(n)^{\mathrm{new}} \setminus \mathrm{cl}(n)^{\mathrm{old}} = \mathrm{cl}(n')^{\mathrm{new}} \setminus \mathrm{cl}(n')^{\mathrm{old}} = \delta(n') \quad .$$

For a β node with children n' and n'', we assume that n' is the one at or below which l lies. This implies that $\mathrm{cl}(n'')^{\mathrm{new}} = \mathrm{cl}(n'')^{\mathrm{old}}$, so we have

$$\begin{aligned}
\delta(n) &= \mathrm{cl}(n)^{\mathrm{new}} \setminus \mathrm{cl}(n)^{\mathrm{old}} \\
&= (\mathrm{cl}(n')^{\mathrm{new}} \cap \mathrm{cl}(n'')^{\mathrm{new}}) \setminus (\mathrm{cl}(n')^{\mathrm{old}} \cap \mathrm{cl}(n'')^{\mathrm{old}}) \\
&= (\mathrm{cl}(n')^{\mathrm{new}} \cap \mathrm{cl}(n'')^{\mathrm{old}}) \setminus (\mathrm{cl}(n')^{\mathrm{old}} \cap \mathrm{cl}(n'')^{\mathrm{old}}) \\
&= (\mathrm{cl}(n')^{\mathrm{new}} \setminus \mathrm{cl}(n')^{\mathrm{old}}) \cap \mathrm{cl}(n'')^{\mathrm{old}} \\
&= \delta(n') \cap \mathrm{cl}(n'')^{\mathrm{old}}
\end{aligned}$$

Finally, for γ nodes, we have to show that

$$\mathrm{cl}(n)^{\mathrm{new}} \setminus \mathrm{cl}(n)^{\mathrm{old}} = \delta(n')|^X \setminus \mathrm{cl}(n)^{\mathrm{old}} \quad .$$

For this, it suffices that $\sigma \in \mathrm{cl}(n)^{\mathrm{new}}$ iff $\sigma \in \delta(n')|^X$ under the side-condition that $\sigma \notin \mathrm{cl}(n)^{\mathrm{old}}$. But under that side-condition,

$$\begin{aligned}
\sigma \in \mathrm{cl}(n)^{\mathrm{new}} \quad &\text{iff} \quad \exists\sigma_0 \in \mathrm{cl}(n')^{\mathrm{new}} \text{ with } \sigma_0|^X = \sigma \\
&\text{iff} \quad \exists\sigma_0 \in \mathrm{cl}(n')^{\mathrm{new}} \setminus \mathrm{cl}(n')^{\mathrm{old}} \text{ with } \sigma_0|^X = \sigma \\
&\text{iff} \quad \exists\sigma_0 \in \delta(n') \text{ with } \sigma_0|^X = \sigma \\
&\text{iff} \quad \sigma \in \delta(n')|^X.
\end{aligned}$$

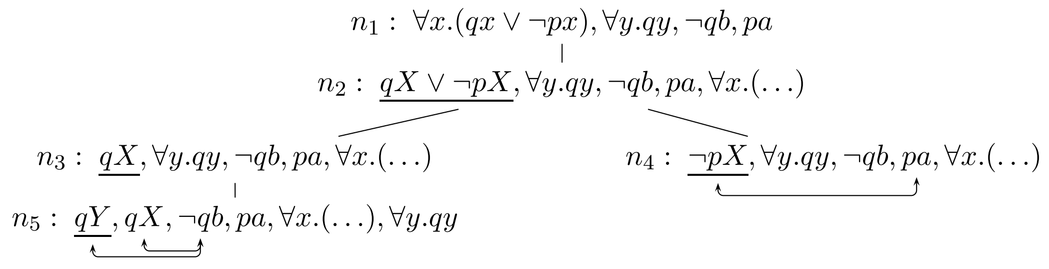
□

According to Prop. 4.11, the tableau is closable when $\mathrm{cl}(\mathit{root}) \neq \emptyset$. This means that if it was not previously closable, it becomes closable if $\delta(\mathit{root}) \neq \emptyset$, or equivalently, if $\delta(\mathit{root}) = \{\Box\}$. The central idea of the incremental closure procedure is to keep track of the sets $\mathrm{cl}(n)$ and update them by propagating the additional closures $\delta(n)$ up the branch using the equations of Lemma 4.15.

Example 4.16 Let us continue the previous example to demonstrate the propagation of δ values. In Fig. 4.5, there is a new node n_5 stemming from a γ-expansion at n_3. This leads to the new complementary pair $qY, \neg qb$. Before this new complementary pair is taken into account, the values of cl are as before. The value $\mathrm{cl}(n_5)^{\mathrm{old}}$ is inherited from n_3, so

$$\mathrm{cl}(n_5)^{\mathrm{old}} = \{\sigma \in \mathcal{I}(\{X, Y\}) \mid \sigma(X) = b\} \quad .$$

Now, taking n_5 as focused leaf and including $qY, \neg qb$, we can calculate δ for all nodes between n_5 and the root as shown in the figure. A non-empty δ value is calculated for the root, so the tableau is closed.


$n_1 : \forall x.(qx \vee \neg px), \forall y.qy, \neg qb, pa$
$n_2 : \underline{qX \vee \neg pX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$
$n_3 : \underline{qX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$
$n_4 : \underline{\neg pX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots)$
$n_5 : \underline{qY}, qX, \neg qb, pa, \forall x.(\ldots), \forall y.qy$

$$
\begin{aligned}
\delta(n_5) &= \mathrm{unif}(qY, qb)|_{\{X,Y\}} \setminus \mathrm{cl}(n_5)^{\mathrm{old}} \\
&= \{\sigma \in \mathcal{I}(\{X,Y\}) \mid \sigma(Y) = b \text{ and } \sigma(X) \neq b\}
\end{aligned}
$$

$$
\begin{aligned}
\delta(n_3) &= \delta(n_5)|^{Y} \\
&= \{\sigma \in \mathcal{I}(\{X\}) \mid \sigma(X) \neq b\}
\end{aligned}
$$

$$\delta(n_4) = \emptyset$$

$$
\begin{aligned}
\delta(n_2) &= \delta(n_3) \cap \mathrm{cl}(n_4)^{\mathrm{old}} \\
&= \{\sigma \in \mathcal{I}(\{X\}) \mid \sigma(X) = a\}
\end{aligned}
$$

$$
\begin{aligned}
\delta(n_1) &= \delta(n_2)|^{X} \setminus \mathrm{cl}(n_1)^{\mathrm{old}} \\
&= \{\Box\} \setminus \emptyset \\
&= \{\Box\}
\end{aligned}
$$

Figure 4.5: Closing the tableau of Fig. 4.4.

Note that the computation needs the old values of cl for some of the nodes (n_5, n_4 and n_1 in the example), so an implementation of this algorithm will have to *store* these values between consecutive closure tests. Also, these stored values will have to be *updated*, when a new complementary pair is taken into account.

The idea of storing previous sets of closing instantiations and updating them by propagating δ sets toward the root is the core of the Incremental Closure technique. The following sections and chapters discuss various modifications and refinements, but the basic principle is the one that was described in this section.

4.4 Implementation Issues

While the last section concentrated on *what* we wish to compute, we shall now discuss *how* this computation may be organized in an actual implementation. The presented structure closely corresponds to that of the the prototypical implementation PrInS.[3]

We shall start by giving an idea of how the incremental closure technique may be implemented, based on its simplest form as outlined in the previous section. We shall also give some experimental results to compare the technique to a simple backtracking prover.

4.4.1 Infrastructure of an Implementation

Still assuming we could calculate with infinite sets of instantiations, we shall now show how the computation and propagation of the δ values is organized. As storing and modifying the sets of closing instantiations $\mathrm{cl}(n)$ is a central part of the technique, it makes sense to describe the procedure in a state-based way. On the other hand, it turns out that operations on the state are typically of very local nature—to calculate a new δ, one only has to look at one previously calculated δ, and possibly the remembered set cl of a neighbouring node. For that reason, we shall take an object-oriented (OO) view, which supports local state-based calculations very well. In fact, the prototypical implementation PrInS is written in the JAVA programming language [GJS97].

Alternatively, it is also viable to describe this approach in a way that is closer to lazy functional programming. This view is described in Sect. 4.7, along with a number of arguments in favor of the OO view.

A UML class diagram [UML01] for an incremental closure prover is depicted in Fig. 4.6. The prover keeps track of a set of leaf goals, each of which contains a set of formulae. Rule applications modify the set of leaf goals. For a β-expansion, one of the goals is replaced by two new ones.

Every leaf goal has an associated Sink object. A sink is an object capable of receiving a set of instantiations and performing some computation on it. One can say that the Sink is a *consumer* for sets of instantiations. This is realized by giving a put method to the Sink objects that takes a set of instantiations as parameter. In JAVA terms, Sink is

[3]That acronym stands for 'Proving with Instance Streams', an old name for the incremental closure approach which stems from a different view of the approach, explained in 4.7.

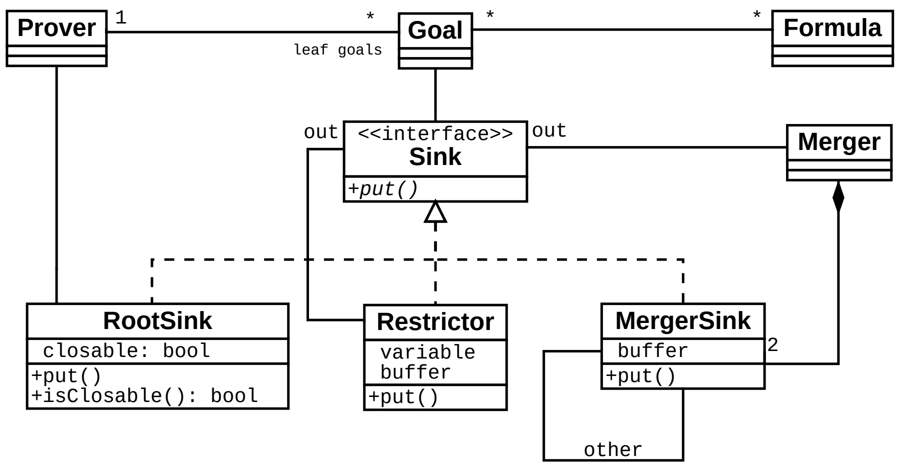
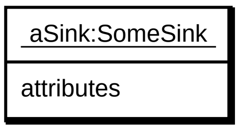


Figure 4.6: A UML diagram for an incremental closure prover.

an *interface* consisting of that one method put. Different objects may implement this interface and do different things with the parameter, as appropriate. In object diagrams, we will give Sink objects a more conspicuous border to make them easier to distinguish from the other kinds of objects:



After any expansion step that leads to a new complementary pair, the proof procedure will pass the set $\delta(n)$ of new closing instantiations to the associated sink by calling the put method. In an OO notation that is:

$$\mathsf{goal.sink.put}(\mathrm{unif}(\phi,\psi)|_{\mathrm{fv}(n)} \setminus \mathrm{cl}(n)^{\mathrm{old}})$$

Of course, to do this calculation, we have to keep the set $\mathrm{cl}(n)^{\mathrm{old}}$ in memory for each goal. We shall assume, for the time being, that we actually do keep them in some buffer associated with each goal. In Sect. 4.5, we will describe a more efficient alternative.

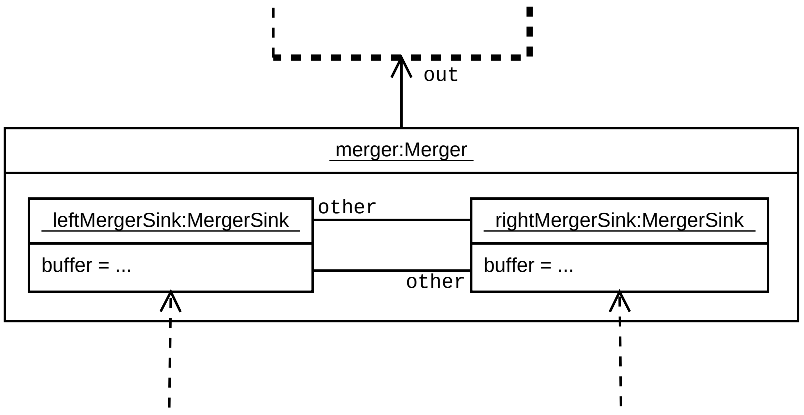
There are three kinds of objects that act as sinks, defined by three classes implementing the Sink interface. One is the RootSink, which will receive $\delta(root)$. This contains a flag closable that records whether a non-empty set of instantiations has yet been received:

```
RootSink::put(S) is
  if S nonempty then
    closable := true
  end
end
```

The prover holds a reference to the unique RootSink and queries the closable attribute after each proof step to determine whether the tableau constructed so far is closable.

The next flavour of sink is provided by Merger objects, which correspond to the splits in the tableau and are responsible for calculating the intersections $\delta(n) = \delta(n') \cap \mathrm{cl}(n'')^{\mathrm{old}}$.

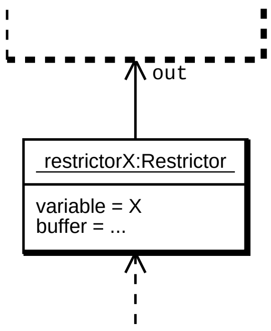
The structure of a Merger is shown in the following diagram:



It consists of two MergerSink objects, one to receive $\delta(n')$ and one for $\delta(n'')$. The current set $\mathrm{cl}(n')$, resp. $\mathrm{cl}(n'')$ is stored in a buffer B (attribute buffer in Fig. 4.6) in the corresponding input sink. Furthermore there is a reference out to an output sink, to which $\delta(n)$ will be passed on. The two sinks are mutually connected by an association other, so they can access each others buffers via other.B. Accordingly, the put method of the MergerSink object works as follows:

```
MergerSink::put(S) is
  J := S ∩ other.buffer      // $\delta(n) = \delta(n') \cap \mathrm{cl}(n'')^{\mathrm{old}}$
  buffer := buffer ∪ S       // $\mathrm{cl}(n')^{\mathrm{new}} = \mathrm{cl}(n')^{\mathrm{old}} \cup \delta(n')$
  out.put(J)
end
```

The last kind of sink is the Restrictor. It corresponds to the γ-expansions of the tableau, and according to Lemma 4.15, its put method must restrict the domain of each incoming instantiation and send the result to an output sink out, if it was not previously known. Thus, a Restrictor looks as follows:



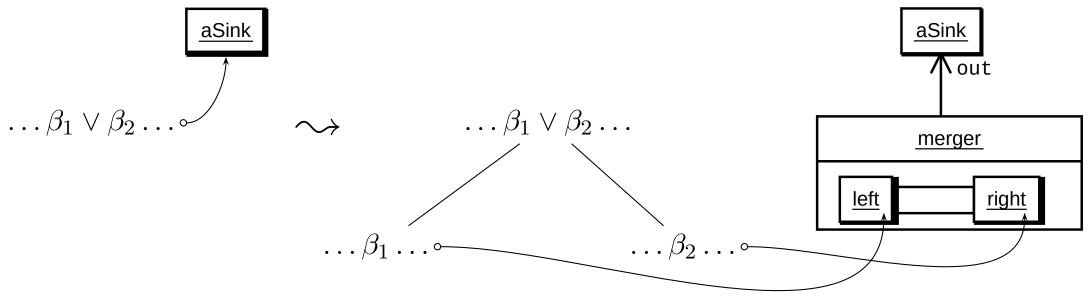


Figure 4.7: Changing the sink structure for a β-expansion.

Here is the implementation of the put method for a Restrictor:

```
Restrictor :: put(S) is
  R := S. restrict ( variable ) \ buffer  // $\delta(n) = \delta(n')|^X \setminus \mathrm{cl}(n)^{\mathrm{old}}$
  buffer := buffer ∪ R                 // $\mathrm{cl}(n)^{\mathrm{new}} = \mathrm{cl}(n)^{\mathrm{old}} \cup \delta(n)$
  out.put(R)
end
```

Note that this setup requires keeping a lot of buffers in memory. In Sect. 4.5 we will show how this can be amended.

The proof procedure can now be stated thus:

```
T := initial tableau for S
r := new RootSink
associate r with goal of T
while ( not r. closable ) do
   if expandable(T) then
      select possible expansion of T
      expand T
      possibly generate new Merger or Restrictor
      handle new complementary pairs
   else
      answer ' satisfiable '
   end
end
answer ' unsatisfiable '
```

At the initialization, a RootSink object is associated with the single goal of the tableau.

In the case of a β-expansion, i.e. a new split in the tableau, the step ‘possibly generate new Merger or Restrictor’ creates a new Merger object. The buffers are initialized with the current value of cl of the parent node. The output of the merger object is sent to the sink s of the parent node, and the new child nodes are associated with the input sinks of the merger, as shown in Fig. 4.7. Analogously, for a γ-expansion, a new Restrictor object is created and connected to the Sink structure.

After the sinks have been updated, the procedure checks for new complementary pairs introduced by the expansion, calculates $\delta(n) = \mathrm{unif}(\phi, \psi) \setminus \mathrm{cl}(n)^{\mathrm{old}}$ for each of them, and sends $\delta(n)$ into the associated sink of the goal.

After all new closing instantiations have been passed to the sinks, the tableau is closable, if the closable flag of the root sink has been set.

4.4.2 Representation of Instantiation Sets

We have so far assumed that we can compute with infinite sets of instantiations. In a concrete implementation, we have to represent these with finite data structures. We shall now describe the representations used in the prototypical prover PrInS.

We use *syntactic constraints* to denote sets of instantiations: These are first-order formulae with a certain fixed signature, endowed with a fixed interpretation over the domain consisting of all ground terms. Free variables in constraints are identified with the free variables of the tableau. For the proof procedure as it has been presented so far, we need only one predicate symbol $\equiv$, interpreted as syntactic equality. Conjunction, disjunction and negation in constraints will respectively be written as $C \,\&\, D$, $C \mid D$ and $!\,C$, to make constraints more easily distinguishable from ordinary formulae. We need existential quantification, which will be written $[X]C$. We also use the symbols $\top$ to denote the unsatisfiable constraint and $\bot$ for the constraint satisfiable by every instantiation. [4] A constraint represents the set of instantiations that satisfy it.

Example 4.17 The constraint

$$X \equiv a \,\&\, Y \equiv f(X)$$

is satisfied by all instantiations that map X to a and Y to $f(a)$.

$$[Y]X \equiv f(Y)$$

is satisfied by any instantiation that maps X to a term with top function symbol f. The constraint

$$X \equiv f(X)$$

is unsatisfiable, since $f(t)$ cannot be syntactically equal to t for any term t.

As the interesting question about sets of instantiations was whether they were empty, we are now interested in the satisfiability of constraints. Satisfiability checking for syntactic constraints is a well-researched field, see e.g. [Com91].

We can now modify the procedure to use constraints wherever we were previously working with sets of instantiations. E.g. $\mathrm{unif}(p(s), p(t))$ yields a constraint $s \equiv t$. The intersection of sets of instantiations required in the Mergers corresponds to the conjunction of constraints. In the updates of the MergerSinks' buffers, set union is required, which can be represented as disjunction of constraints. The domain restriction for instantiations in Restrictor objects corresponds to existential quantification. Finally, the set difference operation needed in Restrictor objects, and to represent $\mathrm{unif}(\phi, \psi) \setminus \mathrm{cl}(n)^{\mathrm{old}}$ for a new complementary pair can be modeled by taking negation into the constraint

[4] A more in-depth treatment of constraints will follow in Sect. 6.3.

language: $A \setminus B$ becomes $C \,\&\, !\, D$ for constraints C and D corresponding to instantiation sets A and B.

There is however a problem with this direct translation: each additional connective in the constraint language makes the satisfiability check more expensive. This is particularly the case for negation. It is therefore worthwhile to look closely at the kinds of constraints that really occur, and how they are combined, to optimize the constraint handling. We shall show that it is possible to modify our procedure so that it can be implemented using only satisfiability and subsumption checking for constraints that consist of conjunctions of syntactic equalities.

Our uses for negation of constraints come from set difference operations, which are needed in two places: For leaves

$$\delta(l) = \mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(l)} \setminus \mathrm{cl}(l)^{\mathrm{old}}$$

and for γ-expansions

$$\delta(n) = \delta(n')|^{X} \setminus \mathrm{cl}(n)^{\mathrm{old}} \quad .$$

In both cases the effect is that of preventing the propagation of instantiations that were previously known to close a subtableau. If one just used

$$\delta(l) = \mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(l)}$$

and

$$\delta(n) = \delta(n')|^{X}$$

instead, the procedure would remain correct, and one could do without negation in the constraint language. But this would lead to inefficiency because of the propagation of redundant constraints. In practice, one has to find a compromise between the two extremes, in other words

$$\mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(l)} \setminus \mathrm{cl}(l)^{\mathrm{old}} \subseteq \delta(l) \subseteq \mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(l)}$$

and

$$\delta(n')|^{X} \setminus \mathrm{cl}(n)^{\mathrm{old}} \subseteq \delta(n) \subseteq \delta(n')|^{X} \quad ,$$

which can be easily computed but still catches enough previously known closing instantiations.

The compromise we shall adopt here is to refrain from propagation of a δ, whenever it is *subsumed* by $\mathrm{cl}(n)^{\mathrm{old}}$, that is when *all* the new instantiations are already known. We can then do without negation in the constraint language.

Definition 4.18 *A constraint C is* subsumed by *a constraint D, if any instantiation that satisfies C also satisfies D.*

Example 4.19 In our previous examples (see e.g. 4.16) we had a leaf l with a new complementary pair $qY, \neg qb$, which would lead to a constraint $Y \equiv b$. The set of previously known closing instantiations $\mathrm{cl}(l)^{\mathrm{old}}$ is represented by the constraint

$X \equiv b$. Using negation, we would propagate a δ of $Y \equiv b \,\&\, !\, X \equiv b$. Using the subsumption variant, we would determine that $Y \equiv b$ is not subsumed by $X \equiv b$, because there are instantiations that map Y to b but not X to b. Thus, a δ of $Y \equiv b$ would be propagated. By contrast, a new complementary pair $r(X, Y), \neg r(b, b)$ would lead to a constraint $X \equiv b \,\&\, Y \equiv b$, which would not be propagated in either case, because this constraint *is* subsumed by $X \equiv b$.

If the constraint language does not contain negation, there is another possibility for simplification: We can shift existential quantifiers out of the constraint, possibly renaming bound variables (as in the usual computation of a conjunctive normal form). We can then leave out the quantifiers, preserving constraint satisfiability.

Example 4.20 The constraint

$$([Y]X \equiv fY) \,\&\, ([Y]Z \equiv gY)$$

is equivalent to

$$[Y_1][Y_2](X \equiv fY_1 \,\&\, Z \equiv gY_2) \quad ,$$

which is satisfiable iff

$$X \equiv fY_1 \,\&\, Z \equiv gY_2$$

is satisfiable.

Of course, existential quantifiers do make a difference for subsumption. But in a clever implementation, one does not need to store the existentially quantified variables explicitly: At any node n, all variables not in $\mathrm{fv}(n)$, i.e. all variables introduced at or below n, can be considered to be existentially quantified.

If we forbid negation and quantification in our constraints (see below), we are left with only positive boolean combinations of syntactic equality constraints. These can be transformed into a disjunction of conjunctions of equations. Such a constraint

$$(s_{11} \equiv t_{11} \,\&\, \ldots \,\&\, s_{1k_1} \equiv t_{1k_1}) \mid \ldots \mid (s_{l1} \equiv t_{l1} \,\&\, \ldots \,\&\, s_{lk_l} \equiv t_{lk_l})$$

is satisfiable if and only if one of the disjuncts is satisfiable, and each of those amounts to a simultaneous unification problem. In other words, the constraint satisfaction problems we have to handle are essentially unification problems.

There is however no need to handle arbitrary disjunctive constraints: Disjunction is used to represent the union of instantiation sets, and the only place where we require set union in the implementations of the last section is to add new closing instantiations to the buffers. But then, the buffers can be implemented as lists or sets of conjunctive constraints.

Our approach of replacing negation by subsumption can be carried one step further: We no longer check whether a new δ is subsumed by the disjunction of all the constraints in a buffer. Instead, we check whether it is subsumed by any *one* of those constraints. Again, we obtain a weaker subsumption test, but one that can be checked more efficiently.

Example 4.21 Assume that we have a signature containing only two constants a and b. Let a buffer contain two constraints $X \equiv a$ and $X \equiv b$. The disjunction of these constraints is satisfied by any instantiation, as X has to be instantiated either with a or with b. Thus the disjunction also subsumes any other constraint. By contrast, the constraint $Z \equiv a$ is not subsumed by either of the two constraints in the buffer.

Situations like the one in the example occur rarely enough in practice to make this further simplification worthwhile.

Summing up, the put method of a MergerSink object now looks as follows:

```
MergerSink::put(C) is
  foreach D in other.buffer do
    J := C & D
    if J satisfiable then            // propagate only satisfiable constraints
      out.put(J)
    end
  end
  add C to B
end
```

For a Restrictor, we have:

```
Restrictor::put(C) is
  R := C.restrict(variable)
  foreach D in buffer do
    if R subsumed by D               // with respect to $\mathrm{fv}(n)$
      return
    end
  end
  out.put(R)
  add C to B
end
```

The constraints our program needs to handle are now purely conjunctive.

Example 4.22 Fig. 4.8 contains a UML object diagram for the state of a prover that corresponds to the tableau in Fig. 4.5, but before propagation of the δ for the new complementary pair. The two Goal objects have already been updated.[5] There is a Merger object containing two MergerSinks. Each of the buffers contains one constraint, corresponding to the sets $\mathrm{cl}(n_{3/4})$ of Fig. 4.4.

Now the prover initiates δ propagation for the new complementary pair $qY, \neg qb$ in the left goal. The sequence diagram in Fig. 4.9 demonstrates what happens. The left MergerSink receives the unification constraint $Y \equiv b$. It joins this to the constraint $X \equiv a$ from the other MergerSink's buffer. The conjunction is passed upwards and reaches the RootSink, which sets its closable flag. When the whole call chain terminates, the prover queries the RootSink and knows that the tableau is closable.

[5]We have listed the formulae of the goals in the objects for simplicity, although that is not legal UML.

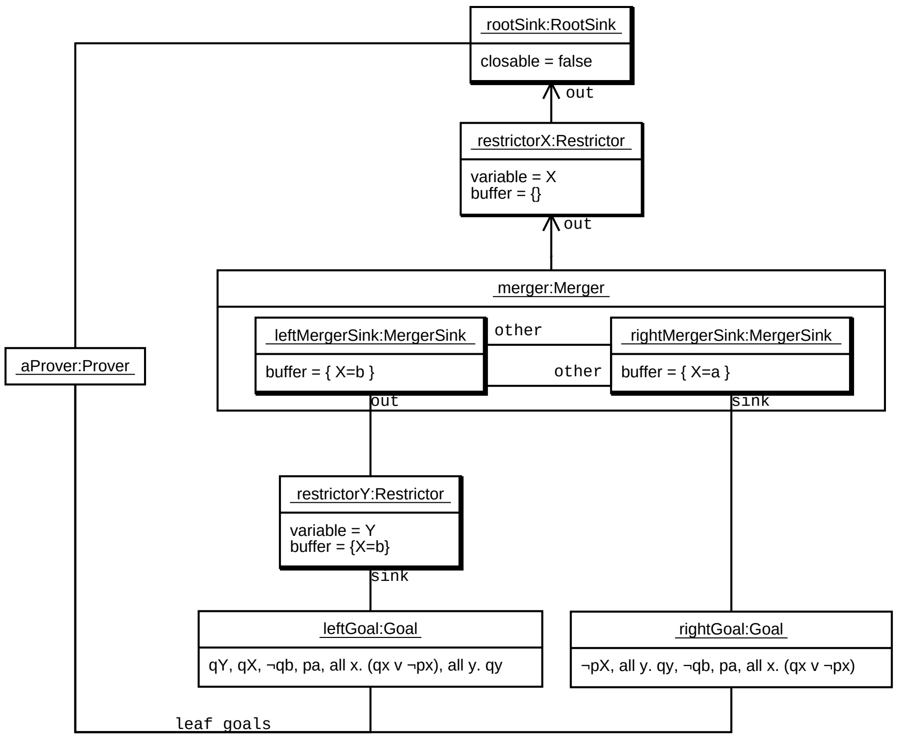


Figure 4.8: An object diagram for a state of the prover.

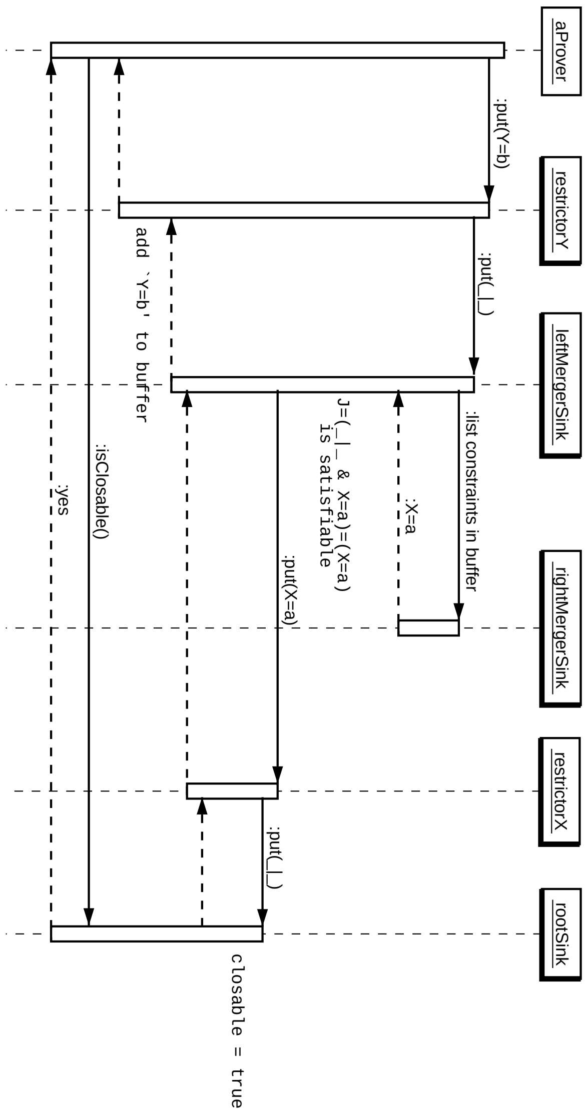


Figure 4.9: A sequence diagram for δ propagation.

4.5 Combination of Restrictors

As it has been presented so far, our architecture requires keeping a lot of buffers for previously known closing instantiations in memory. In this section, we will look at an optimization that reduces the number of needed buffers.

Consider a scenario where n consecutive γ-expansions have taken place on a branch. There would be a chain of k Restrictor objects, where each would keep a buffer of previously known instantiations. Any new instantiation is domain-restricted and checked for containment in a buffer k times. It is more efficient to combine these k steps to remove all k variables in a single step and only then check for containment in a buffer. If n is the top-most γ-expansion node in the chain, n' its child, and n'' is the child of the bottom-most one, we can use a single Restrictor object to calculate

$$\delta(n) = \delta(n'')|_{\mathrm{fv}(n')} \setminus \mathrm{cl}(n)^{\mathrm{old}} \quad .$$

But we don't even have to buffer $\mathrm{cl}(n)^{\mathrm{old}}$ for this single Restrictor. As chains of Restrictor objects have been eliminated, the output of a Restrictor will be passed either into a RootSink or a MergerSink. For the RootSink, the buffer must be empty, because otherwise a closing instantiation for the tableau has already been found. So the new δ can be passed on without any subsumption checking. Otherwise, the Restrictor is connected to a MergerSink, which contains a buffer itself. Clearly, the contents of that buffer are identical to those of the Restrictor. So one can eliminate the buffer from the Restrictor objects and let the MergerSinks do the subsumption checking. This gives us the following implementation for the put method of a MergerSink:

```
MergerSink::put(C) is
  foreach D in buffer do
    if C subsumed by D                    // with respect to $\mathrm{fv}(n)$
      return
    end
  end
  foreach D in other.buffer do
    J := C & D
    if J satisfiable then
      out.put(J)
    end
  end
  add C to B
end
```

The put method for a Restrictor is simplified to:

```
Restrictor :: put(C) is
  R = C.restrictTo($\mathrm{fv}(n')$)
  out.put(R)
end
```

At this point, one could also consider removing the Restrictor objects altogether and adding a restriction to $\mathrm{fv}(n')$, resp. $\mathrm{fv}(n'')$ to the MergerSinks. In fact this shall be done in Sect. 5.7, but for the time being, that modification has no real advantage.

4.6 Experimental Results

In this section we shall report some results obtained with the prototypical implementation PrInS.

Experimental comparisons between theorem proving methods are very problematic. With a sufficiently large benchmark library, like the TPTP [SS97], one can compare entire theorem provers. But the result will depend not only on the underlying principles, e.g. whether a resolution or tableau style calculus was used, but also strongly on aspects like the set of refinements (preprocessing, subsumption, etc.) used, indexing techniques, the implementation language, and last but not least whether the theorem prover was optimized with respect to the benchmark library in question.

To cleanly separate the effects of incremental closure from those of various refinements, the technique was first tested with a very simple implementation: None of the various refinements introduced in the following sections were used.[6] Formulae in goals are kept in a list and the first formula in this list is used for expansion. On the other hand, an equally simple backtracking prover was implemented using the same data structures. Iterative deepening was applied on the number of γ-expansions per branch. The proof search of this backtracking prover is practically identical to that of lean*TAP* [BP94].

Of course, this approach has a number of drawbacks. The most prominent one is that it can be based only on comparatively simple benchmarks. Without suitable refinements, neither a backtracking prover nor an incremental closure prover can handle more difficult problems. This provokes the question whether results obtained in such a comparison 'scale well', whether they still apply if both provers are refined. We believe that this is the case, because there will always be problems that require heavy backtracking in a backtracking prover, however sophisticated it may be, and an approach that can avoid this behaviour will have an advantage. It is however important that the various known refinements for a backtracking prover *can* actually be incorporated into the incremental closure prover. The following chapters will show that this is possible.

Table 4.1 summarizes the results of running both provers on some of the 'Pelletier Problems' introduced in [Pel86].[7] Each prover was run 50 times on each of the problems on a PC with a 2.53 GHz Intel Pentium$^{\text{TM}}$ 4 processor running Linux. Sun's HotSpot$^{\text{TM}}$ Client VM was used to execute the compiled JAVA programs.

The given times represent elapsed real time, averaged over the 50 runs. For problems 34, 38, and 43 the backtracking prover did not find a proof within a time limit of 300 seconds. JAVA provides no means of measuring the pure CPU time consumed by a thread or process, so measuring 'wall clock time' was the only alternative. An advantage of this approach is that the given times include time needed for garbage collection, swapping, etc. The drawback is that there is a certain variation in run times due to system activity

[6] To be quite precise, the goal selection strategy from Sect. 5.2 *was* used. But as a backtracking prover will work on each subtree at least until one instantiation has been found that closes the whole subtree, the behaviour of Sect. 5.2 is automatic for backtracking provers. Using that refinement in the incremental closure procedure does thus not introduce a bias in the comparison.

[7] These simple problems were chosen because lean*TAP* cannot handle anything more complicated.

problem	leanT^AP clone				Incremental Closure			
	time [s]	std. dev.	expand	unify	time [s]	std. dev.	expand	unify
1 : SYN040+1	0.0004	0.0005	16	16	0.0009	0.0015	16	23
2 : SYN001+1	0.0003	0.0005	4	4	0.0004	0.0006	4	7
3 : SYN041+1	0.0001	0.0002	3	2	0.0002	0.0005	3	2
4 : LCL181+1	0.0004	0.0007	16	16	0.0011	0.0025	16	23
5 : LCL230+1	0.0010	0.0026	9	12	0.0001	0.0004	9	14
6 : SYN387+1	0.0002	0.0004	1	1	0.0002	0.0004	1	1
7 : SYN388+1	0.0004	0.0017	1	1	0.0002	0.0004	1	1
8 : SYN389+1	0.0002	0.0004	3	4	0.0002	0.0004	3	5
9 : SYN391+1	0.0002	0.0004	11	19	0.0006	0.0011	11	27
10 : SYN044+1	0.0004	0.0008	18	42	0.0003	0.0005	19	72
11 : SYN390+1	0.0001	0.0004	4	4	0.0002	0.0004	4	7
12 : SYN393+1	0.0015	0.0008	192	465	0.0024	0.0005	192	598
13 : SYN045+1	0.0003	0.0005	40	52	0.0006	0.0014	40	71
14 : SYN392+1	0.0003	0.0006	38	60	0.0006	0.0005	38	83
15 : SYN046+1	0.0002	0.0005	16	16	0.0005	0.0005	16	23
16 : SYN416+1	0.0001	0.0003	3	2	0.0001	0.0003	3	2
17 : SYN047+1	0.0007	0.0011	116	167	0.0008	0.0005	116	208
18 : SYN048+1	0.0002	0.0004	8	3	0.0003	0.0005	4	2
19 : SYN049+1	0.0006	0.0007	25	71	0.0006	0.0008	15	58
20 : SYN050+1	0.0012	0.0011	68	26	0.0007	0.0008	20	16
21 : SYN051+1	0.0004	0.0005	41	74	0.0007	0.0012	21	64
22 : SYN052+1	0.0002	0.0004	27	27	0.0004	0.0005	14	25
23 : SYN053+1	0.0003	0.0005	42	32	0.0006	0.0006	21	30
24 : SYN054+1	0.0040	0.0007	446	1049	0.0031	0.0015	128	634
25 : SYN055+1	0.0006	0.0005	66	99	0.0003	0.0004	9	19
26 : SYN056+1	0.4299	0.0275	30082	59121	0.0091	0.0027	275	2137
27 : SYN057+1	0.0010	0.0012	71	203	0.0009	0.0004	39	282
28 : SYN058+1	0.0004	0.0005	40	71	0.0007	0.0013	35	83
29 : SYN059+1	0.3772	0.0198	31547	161583	0.0015	0.0005	83	422
30 : SYN060+1	0.0000	0.0002	14	15	0.0005	0.0005	11	17
31 : SYN061+1	0.0003	0.0008	28	40	0.0005	0.0005	9	26
32 : SYN062+1	0.0004	0.0005	32	58	0.0008	0.0006	51	137
33 : SYN063+1	0.0006	0.0006	100	151	0.0010	0.0005	97	219
34 : SYN036+2		Timeout			2.4367	0.1874	3938	64201
35 : SYN064+1	0.0003	0.0009	18	4	0.0002	0.0004	6	2
36 : SYN065+1	0.0004	0.0005	34	21	0.0008	0.0022	10	15
37 : SYN066+1	0.0017	0.0007	146	175	0.0007	0.0010	27	37
38 : SYN067+1		Timeout			0.2293	0.0198	1330	17638
39 : SET043+1	0.0002	0.0004	6	4	0.0003	0.0006	5	7
40 : SET044+1	0.0005	0.0005	37	37	0.0004	0.0006	16	28
41 : SET045+1	0.0003	0.0005	16	12	0.0001	0.0004	10	16
42 : SET046+1	0.0026	0.0005	277	578	0.0013	0.0005	48	189
43 : SET047+1		Timeout			3.5518	0.0714	1338	109024
44 : SYN068+1	0.0005	0.0005	40	69	0.0003	0.0008	15	40
45 : SYN069+1	0.0119	0.0053	847	3486	0.0006	0.0006	33	92
46 : SYN070+1	2.2614	0.0454	162131	922934	0.0016	0.0008	65	295

Table 4.1: Comparison between provers on some Pelletier Problems.

entirely unrelated to the theorem provers. The second column of the table gives the sample standard deviation of the measured run times.

The 'expand' column states the number of tableau expansion steps performed during proof search. For the lean*TAP*clone, this can of course very different from the size of the found proof: all expansions are counted, even if they are later removed in a backtracking step. For PrInS, on the other hand, these numbers actually reflect the size of the found proof. The 'unify' column contains the number of calls to the unification procedure. For the lean*TAP*clone, this procedure is called every time a branch is closed (also if it is later reopened via backtracking), for PrInS, the joining/satisfiability testing steps in the Mergers are also counted. Note that in the propositional case, 'unification' just means to check whether two literals are the same. These numbers are of course *not* average values, as they are the same for all 50 runs.

Comparison of the run times is complicated by the statistical variations of the measurements. We use a *t-test* to determine whether the differences in average running time may be considered significant. More precisely, as the standard deviations of the measurements differ between the two provers for many of the problems, the significance test is of the kind that is known as *Behrens-Fisher problem.* We use the Welch modification (see [Wel47]) to the t-test to take this into account. Note that there is a potential problem with the application of the t-test: as times are measured in milliseconds, and some of the problems take only very few milliseconds, we actually get a discrete distribution for our samples. We assume that this is no problem in our case, as the mean of 50 samples very nearly follows a normal distribution, whether the samples are from a discrete set or not. Using an α level of 1%, the difference in running time is considered significant for the entries shown with a shaded background in Table 4.1. The shading was applied for the faster of the two provers.

Problems 1 to 17 are propositional. The difference in running time is significant in only one case, although the measurements indicate that the lean*TAP* clone tends to be slightly faster. This was to be expected, as the incremental closure technique incurs a certain overhead for for the allocation of Mergers, etc. without having any advantages, because no backtracking is required in any case. Note that both provers find the same proofs for the propositional case, hence the identical numbers of tableau expansions.

For the non-propositional problems 18 to 46, the superiority of the incremental closure prover becomes apparent. There is a significant improvement in many cases, and the speedup ratio is sometimes dramatic. For three problems, a proof was found within seconds by the incremental closure prover, while non was found after five minutes by the backtracking prover. The backtracking prover is significantly faster on only three of the simpler non-propositional problems; the lean*TAP* clone needs to do very little backtracking, so the administrative overhead of the incremental closure prover exceeds the gain from the complete avoidance of backtracking.

To understand the situation, it is also helpful to look at Fig. 4.10, which contains a scatter plot of the average runtimes of lean*TAP* vs. PrInS for the given problems (excluding problems 34, 38, and 43), differentiating between propositional and non-propositional problems. The points below the diagonal represent problems for which the the average

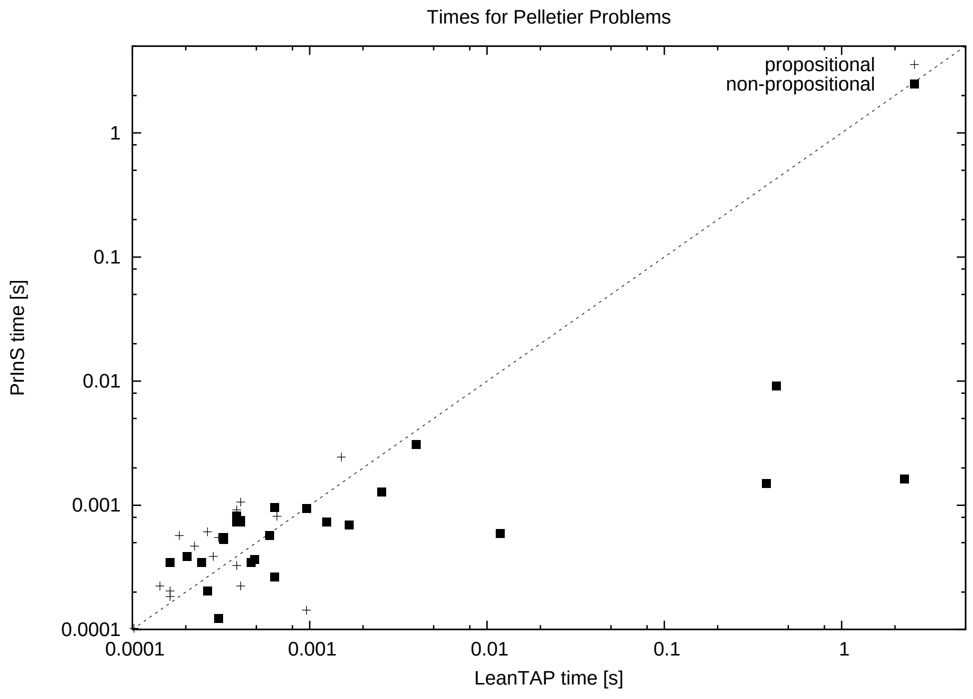


Figure 4.10: Scatter plot of average times for the two provers.

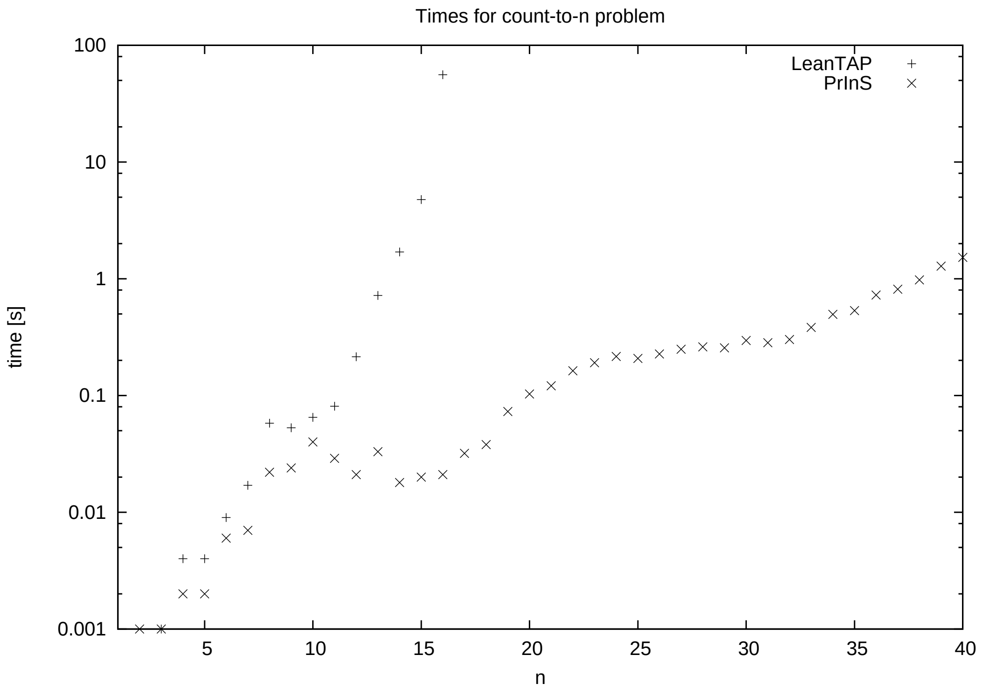


Figure 4.11: Times for the count-to-n problem for the two provers.

required time of PrInS is below that of the lean$T^A\!P$ clone. The distribution of the points clearly indicates that the incremental closure technique introduces a slight overhead for propositional and very simple non-propositional problems, but an interesting speedup for the more complicated ones.

This behaviour becomes clearly visible if one runs the provers on a *family* of problems which require heavy backtracking. For instance, take the formula set

$$p(a), \neg p(f^n(a)), \forall x.(p(x) \to p(f(x))) \quad ,$$

where we define $f^0(a) = a$ and $f^{n+1}(a) = f(f^n(a))$. We call this the count-to-n problem. This set is unsatisfiable for every n. With our set of α,β and γ rules, it requires n γ-expansions of $\forall x.(p(x) \to p(f(x)))$, each of which yields a β formula $p(X) \to p(f(X))$ for some free variable X which then lead to a split in the tableau. Each of the branches in the tableau contains mainly literals of the form $\neg p(X)$ and $p(f(X))$ for different free variables. There are in general very many possibilities of unifying two such literals on any branch, but only one combination of closing complementary pairs will close the proof. This means that a backtracking prover will do a lot of backtracking before it finds a proof.

Fig. 4.11 shows measured running times for both provers for a range of values for n. While the lean$T^A\!P$ clone needs almost one minute for $n = 16$, PrInS solves the problem for $n = 40$ in about 1.5 seconds.

4.7 The Functional View

The incremental closure technique described in this chapter was originally called the *instance stream* technique and sketched in in a position paper of 2 pages [Gie00b].

The view taken there was centered around the sequences of closing instantiations passed between the mergers, which were described as lazily computed lists or *streams*.

A refutation procedure was described as a function that takes an open branch as argument and returns a lazy list of instances closing this branch. This means that elements of the stream are calculated only by need.

Here is a rough sketch of a function 'refute' that takes a set M of formulae in negation normal form and returns a stream of instances, i.e. a list of substitutions for free variables occurring in M, under which M can be refuted. If the initial formula set is unsatisfiable, refute returns a non-empty stream.

$$\begin{aligned}
\text{refute}(\ \{\alpha_1 \wedge \alpha_2\} \cup M\ ) &= \text{refute}(\ \{\alpha_1, \alpha_2\} \cup M\ ) \\
\text{refute}(\ \{\beta_1 \vee \beta_2\} \cup M\ ) &= \text{merge}(\ \text{refute}(\ \{\beta_1\} \cup M\ ),\ \text{refute}(\ \{\beta_2\} \cup M\ )\ ) \\
\text{refute}(\ \{\forall x.\gamma_1\} \cup M\ ) &= \text{refute}(\ \{\gamma_1[x/X], \forall x.\gamma_1\} \cup M)\ |_{\text{FreeVars}(\{\forall x.\gamma_1\} \cup M)} \\
\text{refute}(\ \{L, \neg L'\} \cup M\ ) &= \text{first mgu}(L, L'),\text{ if it exists,} \\
&\ \text{followed by refute}(\ \{L, \neg L'\} \cup M\ ) \\
&\ --\ (\text{ only once for each pair of literals })
\end{aligned}$$

The auxiliary function merge lazily combines unifiers from two lazy lists with each other. Its main function is basically the same as that of the Merger objects, but there is an additional aspect: To start proof search, one computes whether refute(M) is empty for a set of formulae M. The lazy evaluation process then triggers the implicit expansion of the prof tree. In particular, the implementation of the merge function controls the choice of the next branch to expand, because it determines from which of the two streams a next element should be queried.

This view of incremental closure is appealing in its way. The author's first (JAVA) implementation was structured according to the functional view, and it has even prompted other researchers to attempt an implementation in a functional language, see [vE00, ON02].

However, the functional view has a number of severe drawbacks which eventually led to the adoption of the imperative, object-oriented view presented earlier in this chapter.

One point stems from the fact that it is vital for efficiency to use a certain amount of subsumption checking before the merging of instance streams. This means that if the same closing instantiation is found several times for the same subtree, it is going to be processed only once. Consequently, it is quite possible that further expansion of a subtree for which a closing instantiation has already been found will not yield any *new* possibilities. The merge function must thus be implemented very carefully, because requesting the next element of one of the instance streams might not terminate. Certain additional mechanisms have to be built in to ensure fairness. Further complexity arises if refinements like pruning, see Sect. 5.6, are to be incorporated. In the end, the merge function became so complicated that it was practically impossible to implement correctly. Making it too lazy made it inefficient, and making it not lazy enough led to termination problems. There were just too many opportunities for mistakes. In fact, the implementations presented in both [vE00] and [ON02] are accidentally incomplete. See [HS03] for a very nice discussion of this problem.

Another problem of the lazy list view lies in the decision on which branch to apply the next rule. For the lazy list view, this is done implicitly by the merge function. Now it turns out that this is practically the only indeterminism in the proof procedure. It is *the* main decision point. But the primary purpose of the merging functions should be to efficiently find out whether a proof is closed. Although the information available to the mergers can be useful for the code that decides how to expand the proof (see Sect. 5.2), these are two separate concerns which should also be implemented separately.

4.8 Related Work

Apart from the other approaches to tableau proof search without backtracking mentioned in Chapter 3, one very related approach and one more remote connection should be mentioned.

Konev and Jebelean [KJ00] have independently developed an approach similar to the one presented in this chapter. The basic idea of expanding a proof and calculating closing instantiations following the proof structure is the same as for incremental closure. Konev and Jebelean use a natural deduction calculus formulated as a single-succedent sequent calculus, while we use a tableau calculus which is equivalent to a multiple-succedent sequent calculus, but this is only a minor difference.

Instead of delaying branch closure as we do, Konev and Jebelean define proof trees as AND-OR-trees. This means that there are two kinds of branching nodes in their proof trees. AND nodes represent β splits in the tableau. Both of the branches have to be closed. OR nodes are used to represent the decision of whether or not to apply a rule that requires an instantiation. In particular, an OR node is generated at branch closure. On one of the branches, the goal is closed, requiring an according instantiation, on the other it is left open and may be expanded further. There is a certain elegance in this approach. For instance, it permits the application of *rules* which require an instantiation, like for instance the simplification rules we shall present in Chapter 6 or the equality handling rules of Chapter 7, by producing two branches, one with the rule application performed, one without. This possibility is mentioned in the conclusion of [KJ00], but it is not further discussed. We will cope with such rules in the chapters to come by using constrained formulae. Though this is less elegant and homogeneous than the approach of [KJ00], the constraint solution is probably more efficient: If n independent rule applications are possible on a branch, which require various (independent) instantiations, the OR-node approach will lead to 2^n branches. Later rule applications have to be applied separately on all of these. Using constraints, one only introduces n new constrained formulae on one branch. In other words, it is probably not a good idea to make use of the additional possibilities of the AND-OR-tree formulation.

The major difference between our work and that of Konev and Jebelean however is our focus on refinements, which are needed to make the proof procedure efficient. For instance, the approach as described in [KJ00] contains no mechanism corresponding to our Restritor objects. The instantiations of all free variables are kept and stored. This is sensible for Konev and Jebelean, because they are interested in generating human readable versions of automatically found proofs. But in a pure automated reasoning context, this is a waste of memory. They also do not mention any subsumption test for closing instantiations. Of the various refinements described in the next three chapters, only the goal selection strategy of 5.2 is mentioned.

To summarize, the approach taken by Konev and Jebelean is basically similar to the incremental closure technique. But their presentation lacks a number of features which are important for efficient automated deduction.

An entirely different area of research which may be regarded as related to the incremental closure technique concerns parallel implementations of Prolog. The declarative nature of Prolog programs inspired the hope that it would be possible to develop efficient implementations of Prolog on parallel machines, and a lot of research has been done in this direction [Clo87, Clo92, Wre90]. Amongst others, dataflow architectures seem to have received considerable attention [Bic84, CK85, Hal86]. Unfortunately, to our knowledge, none of the proposed approaches have been only remotely as successful as the conventional implementations of the Prolog language. Apparently, operational aspects play an even larger role for efficient parallel programs than they do for sequential programs, undermining the declarative nature of parallel Prolog programs.

Be that as it may, it is interesting to compare the dataflow approaches to the incremental closure technique. The approaches described in [CK85, Hal86] are based on the idea of using the data channels of a dataflow machine to transport variable instantiations which close single Prolog goals.[8] This is very similar to what the Sink structure does in an incremental closure prover. The difference between the approaches for dataflow implementations of Prolog and our work is the organization of the data streams. In our case, closing instantiations from two subtableaux, i.e. solutions for two sub-problems are sent to a single entity, the Merger, which coordinates the work and tries to join compatible solutions. In the cited papers on parallel Prolog, the data channel is directly between the two inference machines responsible for two sub-problems. Any solution for the first problem is passed on to the second inference machine, which tries to complete it in order to obtain a combined solution. This architecture makes sense in the context of logic programming, because it means that the programmer can impose an order on the sub-problems to be solved, as in sequential Prolog. This can be very important for efficiency. Using a concept like the Mergers for Prolog execution would certainly be much too inefficient in most cases.

In an automated theorem prover however, it is impossible to say which of the two goals produced by a β-expansion should be considered first, so we may as well try to approach them simultaneously and employ some heuristic to decide how much work should be spent on which problem.

It is not hard to envisage a multiprocessor implementation of an incremental closure prover, where independent provers are spawned to find closing instantiations for parts of

[8]Apparently this idea was pioneered as early as 1974 by Kowalski [Kow74].

the tableau. The δ-propagation would then be spread over several processors. We have not put any effort into the investigation of this possibility however.

4.9 Summary

In this chapter, we have introduced the incremental closure technique, which can be used to eliminate backtracking from a proof procedure for free variable tableaux. It is built around the idea of incrementally computing instantiations that close sub-tableaux until one global instantiation is found that closes the whole tableau.

The technique was first presented on an abstract level, using potentially infinite sets of instantiations. The abstract procedure was then refined in a number of steps, into a detailed, implementation oriented design.

An experimental comparison between an incremental closure prover and a backtracking prover was given, which showed the superiority of the incremental closure approach for problems which require much backtracking.

Finally, some related work, including a functional view of incremental closure, was presented.

5 Refinements

The Incremental Closure approach has the desirable property that it can easily be refined in a number of ways. We stress this point, because incremental closure is surely not the answer to all problems in automated theorem proving. It is therefore important to see how this new technique can be combined with successful existing approaches.

This section presents a number of possible refinements. While some of them are particular to the incremental closure technique, many are adaptations of refinements known from backtracking procedures.

5.1 The Propositional Case

It occasionally happens that a sub-tableau T is closable under *any* instantiation. This is always the case in proofs of propositional formulae, where no free variables are required at all, but it can also happen with first-order formulae if there is a complementary pair that is unifiable without instantiation of variables that are introduced outside T.

Example 5.1 Consider again the tableau in Fig. 4.5 on page 28. The left branch is closable with $[Y/b]$, so after restriction to the set $\{X\}$ of free variables introduced at or above the β-extension at n_2, one finds that the left branch is closable under any instantiation for X. It is useless to expand that part of the tableau any further, because no more closing instantiations can be found.

Definition 5.2 *A sub-tableau rooted at some node n is called* propositionally closable*, if* $\mathrm{cl}(n) = \mathcal{I}(\mathrm{fv}(n))$.

In the implementation using constraints, this corresponds to a constraint (equivalent to) *true* being passed through the sink structure. If this is detected, the corresponding goals and the Sink structure may be deallocated to reduce memory consumption.

Although this optimization has only little effect in most cases, it is indispensable for propositional logic, and it is easily implemented.

5.2 Goal Selection

So far, the Sink structure built during a proof has only been used to check whether the tableau is closable. It turns out that it can also be useful for goal selection, i.e. deciding on which goal the next expansion step should take place.

Consider a tableau node n with two children n' and n''. There is a corresponding Merger object that contains two MergerSinks, each of which contains a buffer B that represents the current value of $\mathrm{cl}(n')$, resp. $\mathrm{cl}(n'')$. One or both of these sets may be empty:

- If $\mathrm{cl}(n') = \mathrm{cl}(n'') = \emptyset$, a closing instantiation has been found for neither of the branches. At least one has to be found on each of the branches, so the procedure might as well restrict expansion steps to, say, the left sub-tree until one is found there. It might turn out that this sub-tree is propositionally closable, so the corresponding data structures can be deleted. Simultaneously expanding the other sub-tree would only increase the required space.
- If $\mathrm{cl}(n') \neq \emptyset$ and $\mathrm{cl}(n'') = \emptyset$ (resp. $\mathrm{cl}(n') = \emptyset$ and $\mathrm{cl}(n'') \neq \emptyset$), at least one closing instantiation has been found for the left (resp. right) branch and not for the other. Before any other expansions are performed on the left (resp. right) branch to find further closing instantiations, the other branch should be expanded until a closing instantiation is found there as well.
- If $\mathrm{cl}(n') \neq \emptyset$ and $\mathrm{cl}(n'') \neq \emptyset$, some heuristic has to be applied that ensures that expansions are regularly performed on both branches.

These considerations can be used to find the next goal that should be expanded by starting from the root sink and at each Merger selecting one of the sub-tableaux. For propositional goals, if we delete propositionally closable sub-tableaux as described in the previous section, the procedure then becomes a 'one-branch-at-a-time procedure'. This means that the prover only works one branch of the tableau at a time, like backtracking provers do.

The case that at least one closing instantiation has been found for both subtrees is a little tricky. One should keep in mind that due to subsumption checks, a closable subtableau may or may not have more than one closing instantiation. There is no way to find out, except by further expanding the subtableau. One heuristic that does definitely *not* work is to expand each subtableau until it yields a new closing instantiation – there might not be such an instantiation, so this heuristic is unfair.

A fair heuristic would be to chose the left and right subtableaux alternatingly. However, experimentation suggests that this is a little inefficient, because it leads to doing a little work on too many branches simultaneously. A sensible compromise seems to be to work on a branch until a γ-expansion becomes necessary, then give the other branches a chance. This is fair because only a finite number of expansions can be done on a subtableau without intervening γ-expansions.

There are many possibilities for goal selection heuristics, and much can be gained by a good choice. But one must take care not to make the proof procedure incomplete.

5.3 Formula Selection

One of the most obvious refinements for the proof procedure as implemented in the prototype PrInS introduced in Sect. 4.6 is to be more careful in the selection of the formula expanded on each branch. It is obviously sensible to give certain formula types higher priority than others. In particular, literals should be considered first, to see if a branch is already closable. Then α-expansions should be performed, because they lead neither to splitting, nor to introduction of free variables. It is harder to decide whether β- or γ-expansions should be preferred, but β-expansions are generally performed first, because they do not need to be reapplied like γ-expansions. Of course one can also develop heuristics to choose e.g. between several eligible β-formulae.

While the formula selection problem is very similar to that of a backtracking prover, there are some new aspects for an incremental closure prover. First, the number of rule applications needed to close a proof can be much smaller, because there is no backtracking. That means that more time can be expended per rule application, e.g. to decide which formula to expand. And second, heuristics for formula selection and goal selection may be combined, to pick a preferred expansion among all possible expansions of all goals.

5.4 Subsumption in Buffers

An obvious place for optimizations in an implementation of the incremental closure technique are the buffers contained in MergerSinks. These were defined in Sect. 4.4.1, to store sets $\mathrm{cl}(n)$ of known closing instantiations of subtableaux. In Sect. 4.4.2, we suggested implementing these buffers as sets of conjunctive unification constraints. In practice, this implementation, using the subsumption technique described on page 36 is quite sufficient in many cases. In fact, the overall running time of the prover is rarely dominated by the δ propagation process. This is due to the fact that most of the computed δ constraints don't 'travel' very far in the Sink structure. They either get subsumed at some point, or they are incompatible with the known closing instantiations of some other subtree. On the other hand, if propagation *does* bring a δ close to the root, this often means a significant progress in the proof search, because a large subtableau can already be closed.

But the problem of detecting closability of a tableau is NP-hard, so there will always be cases in which δ propagation is expensive. For such cases, it is worthwhile to work on the efficiency of δ propagation. Even though the problem remains NP-hard, we might be able to avoid the exponential behaviour in some cases, and in other cases an improvement by a large constant factor could still make a big difference in practicality.

If we analyze the final implementation of the put method for MergerSinks given in Sect. 4.5, we see that there are two potentially time consuming operations: Checking whether the new δ is subsumed by one of the constraints in the buffer, and if this is not the case, checking whether the conjunction with each of the constraints in the other buffer is satisfiable. Clearly, the easiest way to keep the amount of work low is to keep the buffers small. Any superfluous constraint in a buffer slows down subsequent δ propagations. Checking whether a new δ is subsumed by one of the known ones as is done in Sect. 4.4.2 and 4.5 is one way of keeping buffers small: if a new closing constraint that is subsumed by a previously known one were not discarded, it would get stored in at least one of the buffers, making it larger unnecessarily. We refer to this subsumption check as *forward subsumption*. Forward subsumption means discarding new information if it is subsumed by older information.

One can also check whether the converse holds, namely whether a new closing instantiation that is about to be put into the buffer subsumes one of the previously known instantiations. This is referred to as *backward subsumption*. Any constraints in a buffer that are subsumed by a new constraint can be removed from the buffer.

Example 5.3 Let the buffer of some MergerSink contain the constraint $C_1 = (X \equiv a \,\&\, Y \equiv a)$, indicating that the prover has found out that the subtableau in question can be closed if both X and Y are instantiated to a. After some further expansion, the prover finds that the subtableau can be closed whenever X and Y receive the same instantiation. The constraint $C_2 = (X \equiv Y)$ gets propagated. C_2 is not (forward) subsumed by C_1, so it gets added to the buffer. But C_1 is (backward) subsumed by C_2, so C_1 can be removed from the buffer.

There is a trade-off to consider when subsumption is used. While subsumption checking can help in keeping buffers small, it also takes time. Experimentation with PrInS revealed that forward subsumption has a considerable impact even for the small problems discussed in Sect. 4.6. In fact, without forward subsumption, the incremental closure prover is rarely faster than the backtracking prover, because although few expansion steps were needed, the running time was dominated by δ propagation.

The case is different for backward subsumption. In the problems it was tested on, it was almost never the case that a constraint in the buffer was subsumed by a newer one. But testing for backward subsumption slowed the prover down noticeably. Of course, depending on the application domain, one might encounter problems where backward subsumption pays off. In any case, it is an option to consider when buffers grow too large.

While subsumption checking can ward off exponential blow-up in some cases at a modest cost by reducing the size of buffers, it is also possible to accelerate operations on large buffers. This will be discussed in the next section.

5.5 Indexing

There are at least two places in our suggested implementation where we perform what amounts to a linear search over a set of entities which might become large. One of these searches occurs when a new literal L is added to a branch. The prover scans the literals already present on that branch and checks whether any of these are unifiable with L. In a large proof, this might be time consuming, although most of the literals present are probably not unifiable with L. A linear search is also performed by the MergerSinks' put method, both for checking subsumption by a constraint in the buffer and for finding compatible constraints in the buffer of the buffer of the 'other' MergerSink. If backward subsumption is used, this would also done by a linear scan of the constraints in the buffer. Again, if the buffers are large, these scans can be time consuming, and for most of the constraints in the buffer, the conjunction will not be satisfiable, resp. the subsumption check will fail.

The term *indexing techniques* is used in automated theorem proving literature for data structures and algorithms which store a large number of items (e.g. terms, clauses, substitutions, etc.) in such a way that a small subset of these items relevant for a particular query (e.g. terms unifiable, clauses subsumed, substitutions compatible with a given term/clause/substitution) can be retrieved quickly. Most of the known indexing techniques have been developed for resolution theorem provers (see e.g. [Gra96, NHRV01, RSV01]), where it is important to find a small number of candidates for clause subsumption, resolution, or superposition from a set of possibly millions of clauses. Indexing techniques have classically not played a major role for tableau provers for two reasons: One is that the operations in a tableau prover typically concern only the formulae on a single branch. Instead of producing very many clauses, like a resolution prover, a tableau prover will produce very many branches, each of which is usually of moderate size, so there is not much benefit in using indexing techniques. The other reason is that the data structures used for indexing are sometimes quite complex, so the cost of just allocating and initializing them is not negligible. This does not matter for a resolution system, where an index is created once in the beginning, but for a tableau, one has this overhead for every new branch.

The situation is a little different for an incremental closure prover. For one thing, the MergerSink buffers *can* get large. It is not unusual for a buffer to accumulate in the order of fifty constraints. This does not compare to a set of millions of clauses, but it can make indexing worthwhile. Depending on the problem to be solved, a branch might also accumulate a considerable number of literals to be scanned for unifiability. This is especially the case if the techniques from Sect. 6.7 are used. And finally, the overhead for creating the indexing structures is not as severe as for a backtracking prover because branches are not constantly deleted and recreated. If an index is created, it will remain useful for the rest of the proof search.

Unfortunately this does not mean that we can usefully adopt the most successful indexing techniques from resolution theorem provers. The amount of effort that should be put into an indexing structure strongly depends on the typical size of the sets of items constructed by the prover.

Take the set of literals on a branch, for instance. In an application domain where one rarely gets more than five literals on a branch, one should not worry about indexing at all. Of course, it never hurts to store positive and negative literals separately and scan only one of the lists. This simple measure will already halve the average time needed for finding unifiable literals. If the application domain leads to larger numbers of literals, one can go a step further by keeping a separate list of literals for each top-level predicate symbol. Literals with distinct top-level predicate symbols can't be unifiable. Of course, this works only if the problem formulations typically use several different predicate symbols. If there are *many* different predicate symbols, the lists of literals per predicate can be organized in a hash table, otherwise a linear list is probably more effective. If indexing on top-level predicate symbols is not sufficient – either because there are only few predicate symbols, or because the lists still become too large – one might consider a Prolog-like indexing scheme which uses the predicate and the top-level function symbol of the first subterm. The point is that the indexing technique to be chosen according to the intended application domain.

For MergerSink buffers, most of the known indexing techniques for resolution are useless, because we need to index constraints and not clauses or terms. There is however a method called *substitution tree indexing* [Gra96, Gra94] which actually stores a set of substitutions in a tree structure. Though this was not thoroughly investigated, it is expected that this technique could be adapted to suit our needs. In experiments with the current implementation of PrInS, buffer sizes rarely grew to a size which would have warranted the design of an indexing structure. But this may well be necessary, depending, as usual, on the application domain.

5.6 Pruning

Pruning (see e.g. [BH98, Häh01, BFN96a]), which is also known as Condensation, Back-jumping or intelligent backtracking is an important technique known from backtracking procedures and SAT-provers for propositional logic. It is an excellent example of how a known refinement can be 'ported' to the incremental closure framework.

Pruning can reduce the search space dramatically: The prover keeps track of the *ancestry* of formulae, i.e. the set of formulae in the tableau which were used to derive it.[1] If a branch is closed by unifying a particular complementary pair $\phi, \neg\psi$, the prover examines the β-expansions that occurred earlier on the branch. If for a particular expansion at a node n, neither the ancestry of ϕ, nor that of $\neg\psi$ contains the sub-formula $\beta_{1/2}$ introduced by the expansion, then the closure would have been possible without that expansion. This is because the sequence of proof steps that led to a closed subtableau under $n_{1/2}$ would have been possible below n, leading to a closed subtableau without performing the β-expansion step.

Consequently, the β-expansion can be removed *a posteriori*, saving the work of closing the other branch introduced by it. Of course, in a backtracking prover, the decision for that particular complementary pair might be revised in a backtracking step, and then the expansion has to be reintroduced. By contrast, in a propositional setting, no backtracking is needed, and the β-expansion can be eliminated permanently.

In a way, pruning is a way to make up for errors in the formula selection heuristics. If a β-expansion turns out to be unnecessary, it can be rendered harmless by pruning.

The question is now how the pruning technique can be adapted to the incremental closure approach. As in the backtracking case, we have to provide for the case that the complementary pair $\phi, \neg\psi$ in question is not the one used to finally close the whole proof. A backtracking prover reintroduces the β-expansion when it backtracks. We want to do without backtracking, so we cannot simply remove the expansion. But what *can* we do?

As in the backtracking prover, we record for each formula an ancestry of β-expansions on which it depends. We can use references to the Merger objects for this, as there is

[1]Actually, it suffices to record only the β-subformulae introduced by β-expansions.

exactly one of these for each β-expansion. In terms of the abstract view given in Sect. 4.3, we now compute with sets of pairs (σ, h) of instantiations with ancestries, instead of just sets of instantiations. We have to redefine the operations unif, $\cap$, $\cup$ and $\setminus$ to work with sets of such pairs. In particular, the $\cap$ operation in the Merger has to *combine* the histories of instantiations, taking the unions of ancestor sets.

The 'pruning' takes place, when a Merger m receives an instantiation that does not have m in its ancestry: it can pass such an instantiation to the output sink independently of the contents of the buffer of the other branch:

```
MergerSink::put(S) **is**
  P := $\{(\sigma, h) \in \mathsf{S} \mid \text{this Merger} \notin h\}$
  out.put(P)
  S := S $\setminus$ P
  J := S $\cap$ other.buffer
  buffer := buffer $\cup$ S
  out.put(J)
**end**
```

The first three lines in the body perform the pruning, while the other three process the remaining instantiations as before.

Example 5.4 Pruning can be used in the tableau in Fig. 4.5 on page 28. Let m be the Merger corresponding to the β-expansion at n_2. Then the literals qX in n_3 and n_5 and $\neg pX$ in n_4 have a history of $\{m\}$, while the other literals have an empty history. The closing instantiation $[Y/b]$ discovered in n_5 depends on the literals qY and $\neg qb$, which have an empty history, so the history of $[Y/b]$ is also empty, and the instantiation can be passed through by m directly, without combining it with $\mathrm{cl}(n_4)^{\mathrm{old}}$.

It is interesting to ask whether passing through of closing instantiations is as effective as actually discarding a branch and reinserting it for backtracking. Consider a problem where a given backtracking prover finds the *right* complementary pair immediately for some subtableau, so there is no need for backtracking. In that case, the given closing instantiation is also the first one that the incremental closure prover will find for that subtableau. The goal selection strategy of Sect. 5.2 will then ensure that at least one closing instantiation will be found for the rest of the tableau before the pruned branch is reconsidered. If the backtracking prover required no backtracking, those other closing instantiations should close the whole proof, so the incremental closure prover will not reconsider the pruned branch either.

We now give an experimental comparison of the prover as described in Sect. 4.6 to a version that uses pruning and that discards propositionally closed subtrees as proposed in Sect. 5.1. Table 5.1 shows some results for the pigeon hole family of problems in the propositional MSC007-1 formulation of the TPTP library. For this kind of propositional problem, pruning obviously brings a dramatic improvement.

For the problems of Table 4.1 on page 41, there is hardly any improvement for the propositional problems. They are just too simple for pruning to have any discernible

	without pruning		with pruning	
size	time [s]	expansions	time [s]	expansions
2	0.001	3	0.001	3
3	0.006	61	0.003	36
4	35	33566	0.046	438
5	>1000	?	1.0	5671
6	>1000	?	140	75260

Table 5.1: Effect of Pruning for Pigeon Hole problems (MSC007−1)

	without pruning			with pruning		
problem	time [s]	expand	unify	time [s]	expand	unify
SYN054+1	0.0160	128	634	0.0120	88	331
SYN056+1	0.0940	275	2137	0.0590	271	1764
SYN059+1	0.0100	83	422	0.0110	64	309
SYN063+1	0.0050	97	219	0.0040	91	180
SYN036+2	2.9020	3938	64201	0.4150	2581	14662
SYN067+1	0.2440	1330	17638	0.0580	838	5601
SET047+1	3.5330	1338	109024	0.0690	664	3995

Table 5.2: Effect of Pruning for some Pelletier problems

effect. Table 5.2 shows some results for non-propositional problems.[2] The benefit becomes of pruning is quite noticeable for the more complex problems, although of course not as extreme as for the pigeon hole problems.

5.7 Universal Variables

In the tableau calculus as it has been presented so far, each occurrence of a free variable introduced by a γ rule had to receive the same instantiation in order to close a proof. In fact, finding a *consistent* instantiation for the free variables is the central problem we want to solve with the incremental closure approach. Variables with the property that all occurrences have to be instantiated in the same way are called *rigid variables.* In a resolution calculus, for instance, the variables occurring in clauses are not rigid, as they may be instantiated differently (by unification) in each resolution step. Indeed, one often talks about an implicit variable renaming step before each resolution. It is not allowed to rename the rigid variable occurrences in a tableau.

As has been observed e.g. in [BH98], there are cases in which this rigidity may be relaxed in a tableau calculus. That means that it is sound to instantiate the same variable independently on different branches or even in different literals on the same

[2]The times are different from Table 4.1 because the experiments were done on another computer.

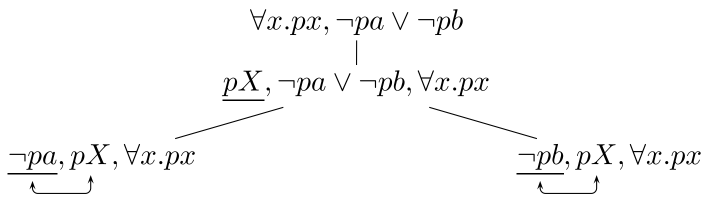


Figure 5.1: Tableau closable with universal variables

branch.

Example 5.5 The tableau in Fig. 5.1 is not closable according to the definitions given so far: X has to be instantiated with a to close the left branch and with b to close the right branch. But the formula $\forall x.px$ is still present on each branch, so a γ-expansion could be applied on one of the branches to introduce a literal pX'. The proof then becomes closable.

One can formalize this possibility to introduce a variant of a formula, that is a copy with renamed variables.

Definition 5.6 *Let $\phi \in G$ be a formula occurring in a goal G of a tableau, and X some free variable. Let $\bar{G}$ be the union of G and all goals above G in the tableau. ϕ is called* universal with respect to X *if $\sigma([X/t](\phi))$ is a logical consequence of $\sigma(\bar{G})$ for all $\sigma \in \mathcal{I}$ and all ground terms t. Let $\mathrm{uv}(\phi)$ be the set of variables with respect to which ϕ is universal.* □

Obviously, if a formula $\phi \in G$ is universal with respect to a variable X, it is sound to include a variant $[X/X']\phi$ in G, for which X' is then universal, or simply to replace ϕ by such a variant. Or we can decide to keep this renaming of universal variables implicit, and simply allow universal variables to be instantiated independently per formula. Of course, multiple occurrences of the same variable in *one* formula must still receive the same instantiations. We can modify our definition of a closed tableau (see Def. 4.2) accordingly:

Definition 5.7 *A goal G is* closed under an instantiation σ*, iff there are a complementary pair $\{\phi, \neg\psi\} \subseteq G$, and ground substitutions μ and ν for the universal variables of ϕ and $\neg\psi$ respectively, such that $\sigma(\mu(\phi)) = \sigma(\nu(\psi))$. A tableau T is* closed under an instantiation σ*, iff each leaf goal of T is closed under σ. A tableau is* closable *iff it is closed under some instantiation.*

The point is that the substitutions μ and ν can be chosen independently per branch.

Example 5.8 Consider again the tableau of Fig. 5.1. In both goals, pX is universal w.r.t. X, because of the formula $\forall x.px$. The left branch is closed under any instantiation σ, because we can instantiate the universal variables with $\mu = [X/a]$, and similarly for the right branch with $\mu = [X/b]$. ν can be left empty on both branches. Taking universal variables into account, the whole tableau is thus closed under any instantiation σ.

As universality of a formula with respect to a given variable is undecidable in general, one usually uses the simple criteria of the following lemma to detect universality in common cases.

Lemma 5.9 *The following sufficient conditions for universality of variables hold:*

1. *If an α-expansion is applied on a formula $\alpha = \alpha_1 \land \alpha_2$, then $\mathrm{uv}(\alpha_{1/2}) \supseteq \mathrm{uv}(\alpha)$ in the resulting goal.*
2. *If a β-expansion is applied on a formula $\beta = \beta_1 \lor \beta_2$, then $\mathrm{uv}(\beta_1) \supseteq \mathrm{uv}(\beta) \setminus \mathrm{fv}(\beta_2)$ in the resulting left goal, and similarly for β_2 in the right goal.*
3. *If a γ-expansion is applied on a formula $\gamma = \forall x.\gamma_1$ introducing a new free variable X, then $\mathrm{uv}([x/X]\gamma_1) \supseteq \{X\} \cup \mathrm{uv}(\gamma)$ in the resulting goal.*

Proof. We only prove the α case, the others are similar. Let G and G' be the goals before and after the expansion. Further, let $X \in \mathrm{uv}(\alpha_1 \land \alpha_2)$. Take an arbitrary $\sigma \in \mathcal{I}$ and some ground term t. Then $\sigma([X/t]\alpha_1)$ is certainly a logical conseuence of $\sigma([X/t](\alpha_1 \land \alpha_2))$, which in turn, since X is universal for the conjunction, is a logical consequence of $\bar{G}$. Hence, $\sigma([X/t]\alpha_1)$ is also a consequence of $\bar{G}'$. □

In an implementation, one can keep a set of universal variables together with each literal. Alternatively, and more economically, one can keep a set of rigid variables for each goal and treat all other free variables occurring in formulae of a goal as universal. If one uses the given criterion, rigid variables are added to this set only in β-expansions. The new rigid variables in a β-expansion are exactly those variables which occur free in both β_1 and β_2.

As rigid variables are now introduced by β-expansions instead of γ-expansions, one can modify the calculus to require a copy of the expanded formula to be retained in β-expansions and not in γ-expansions, as in the following definition:

Definition 5.10 *A* universal variable tableau *for a finite set of SNNF formulae S is defined inductively as follows:*

1. *The tableau consisting of the root node labeled with the goal S is a tableau for S, called the* initial tableau.
2. *If there is a tableau for S that has a leaf n with goal $\{\alpha_1 \land \alpha_2\} \cup G$, then the tableau obtained by adding a new child n' with goal $\{\alpha_1, \alpha_2\} \cup G$ to n is also a tableau for S. (α-expansion)*
3. *If there is a tableau for S that has a leaf n with goal $\{\beta_1 \lor \beta_2\} \cup G$, then the tableau obtained by adding two new children n', resp. n'' with goals $\{\tau\beta_1, \beta_1 \lor \beta_2\} \cup G$, resp. $\{\tau\beta_2, \beta_1 \lor \beta_2\} \cup G$ to n, where τ is a renaming of the variables with respect to which β is known to be universal, is also a tableau for S. (β-expansion)*

4. *If there is a tableau for S that has a leaf n with goal $\{\forall x.\gamma_1\} \cup G$, then the tableau obtained by adding a new child n' with goal $\{[x/X]\gamma_1\} \cup G$ to n, where X did not previously occur in the tableau, is also a tableau for S. (γ-expansion)*

This gives us a complete calculus if we use a criterion for detection of free variables at least as strong as that of Lemma 5.9. See [Gie98], Sect. 7.4, for a completeness proof.

Apart from reducing the number of needed tableau expansions in some cases, this way of formulating the calculus has the advantage of leading to a more homogenous formula selection problem (see Sect. 5.3): It is always sensible to do α- and γ-expansions first, fairness is now an issue only for β-expansions. In a sense, we have combined the old β- and γ-expansions into our new β rule.

An interesting special case occurs for a β-expansion when the sets $\mathrm{fv}(\beta_1)$ and $\mathrm{fv}(\beta_2)$ of free variables of the subformulae are disjoint. In that case, the β-expansion does not introduce any rigid variables. The calculus can be modified in such a way that the original formula $\beta_1 \vee \beta_2$ is not included in the new goals. This means that the β-expansion does not need to be applied multiple times to ensure completeness.

So far, in this section, we have only talked about the tableau construction aspects of universal variables, and hardly about the question of detecting tableau closure. How can universal variables be handled in an incremental closure prover? We have to accommodate the new notion of closed goal and tableau, and we should also review the places where we talk about the free variables $\mathrm{fv}(n)$ present at a node.

It is clear that the instantiations in our closer sets cl should not refer to universal variables, because it is sufficient to know that an instantiation μ or ν for them *exists* when we unify a new complementary pair. In Def. 4.10, we defined the closer sets $\mathrm{cl}(n)$ as sets of instantiations of the free variables $\mathrm{fv}(n)$ introduced above (but not at) the node n. If we use universal variables, we should revise the definition of $\mathrm{fv}(n)$ to be the set of *rigid* variables introduced by β-expansions above (but not at) n. In Lemma 4.15, the calculation of

$$\delta(n) = \mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(n)} \setminus \mathrm{cl}(l)^{\mathrm{old}}$$

then automatically performs the restriction to the set of relevant rigid variables. Variables which are universal for ϕ *and* ψ should be renamed before unification to make sure that no unifying instantiations are missed. The remaining δ propagations of Lemma 4.15 have to be adapted, taking into consideration that the γ rule does not introduce rigid variables, but the β rule does. This leads to

$$\delta(n) = \delta(n')$$

for a γ-expansion at n and

$$\delta(n) = \delta(n')|_{\mathrm{fv}(n')} \cap \mathrm{cl}(n'')^{\mathrm{old}}$$

for a β-expansion. With this modification, the combination of Restrictor s described in Sect. 4.5 is automatic, and as restriction is now part of the calculation for β-expansions, it would even be natural to leave away the Restrictor objects altogether and perform the computation of the restrictors in the MergerSinks.

Clause set:
$px \lor qx$
$\neg pb \lor ra$
$\neg qb$
$\neg ra$

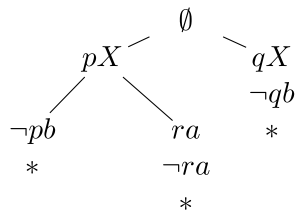


Figure 5.2: A simple clause tableau.

It might seem that the detection and handling of universal variables are not of much consequence for practical problems. After all, the fact that $\forall x.px$ holds for a formula means that the predicate p is not very interesting semantically. Things become more interesting for binary predicates however, like in $\forall x.x \geq 0$. Universal variables are also interesting in combination with the simplification rules introduced in Chapter 6. And finally, for equality handling they permit using rules similar to (unfailing) Knuth-Bendix completion which are much more powerful than rewriting techniques for rigid variables only, see the second item on page 120.

5.8 Regularity

The regularity restriction is a well-known restriction for backtracking clausal tableau provers, see [LSBB92, BH98, Häh01]. For clausal ground tableaux, this restriction forbids rule applications that introduce a literal on a branch that is already present. It is not hard to see that rule applications leading to such duplicates are not needed for a complete proof procedure.

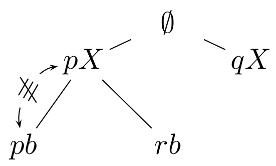
In our setting, there are two problems with the regularity restriction. The first is that we are interested in first order tableaux with rigid variables. Whether two literals are equal or not might depend on the instantiation of rigid variables, and we do not yet know this instantiation when a tableau is expanded.

The second problem is that it is not straightforward to define regularity for non-clausal problems. Forbidding multiple occurrences of the same formula on a branch might render the calculus incomplete, as the simple formula $(p \land p) \land \neg p$ shows. It is unsatisfiable, but this can be shown only by performing an α-expansion on $p \land p$, which leads to two identical literals.

A solution for the first order case has been proposed in [LSBB92]. It consists in gathering the requirements that instantiations should not render literals equal in a *global* constraint. In other words it is not the formulae which are annotated with constraints, but the whole tableau. The trouble is that this approach is not compatible with backtracking free proof search without further effort.

Example 5.11 Consider the clause set and proof in Fig. 5.2. The tableau is closed with $\sigma(X) = b$, and one sees that an instance of the clause $px \lor qx$ with $[x/b]$ is indeed needed to find a closed tableau for the clause set. Now in Fig. 5.3, the redundant

Clause set:
$px \vee qx$
$\neg pb \vee ra$
$\neg qb$
$\neg ra$
$pb \vee rb$



Constraint: $X \not\equiv b$

Figure 5.3: The problem with a Global Constraint.

clause $pb \vee rb$ was added. Assume that the prover expands the left branch with $pb \vee rb$, after initially using $px \vee qx$ as before. Instantiating X with b would now lead to an irregular tableau, so we add $X \not\equiv b$ to the global constraint. But then a new instance of the clause $px \vee qx$ is needed on each of the branches, and the same problem might recur indefinitely.

This behaviour is acceptable for a backtracking prover, which will at some point try expanding with another clause instead of $pb \vee rb$. But for a procedure without backtracking, the only solution would be to use a complicated fairness condition to avoid repeating the mistaken expansion.

As for the question of non-clausal regularity conditions, several proposals may be found in the literature, see e.g. [BH98, Let99, BHRM94]. These usually rely on the concept of *disjunctive paths* through formulae, which we are also going to need in the next chapter, see Def. 6.25. Although the existing proposals are valid, they lack the simplicity of the original clausal version.

There is an elegant way of handling the regularity restriction in our framework using constrained formulae and a simplification rule. As these shall only be introduced in the next chapter, we defer the description of our approach to Sect. 6.8.

5.9 k-ary Branching

Up to this point in our presentation of the incremental closure technique, we have assumed conjunction and disjunction to be binary operations. This makes the presentation and the implementation simpler, but it is not a crucial requirement. In this section, we will sketch how to cope with variadic conjunction and disjunction.

Handling variadic conjunction is in fact trivial: One only needs an expansion rule that replaces $\alpha_1 \wedge \cdots \wedge \alpha_k$ by $\alpha_1, \ldots, \alpha_k$ in a goal. The δ propagation process does not need to be changed for this modification.

Variadic β-expansion is more interesting. The expansion rule itself introduces k new goals for a k-ary disjunction, as one would expect. If one uses universal variables, any free variable that was universal for the disjunction and which occurs in more than one of the disjuncts β_i must be made rigid. One then needs a variadic Merger object.

Let n be the node where the k-ary β-expansion is performed, and let $n_1, \ldots, n_k$ be the

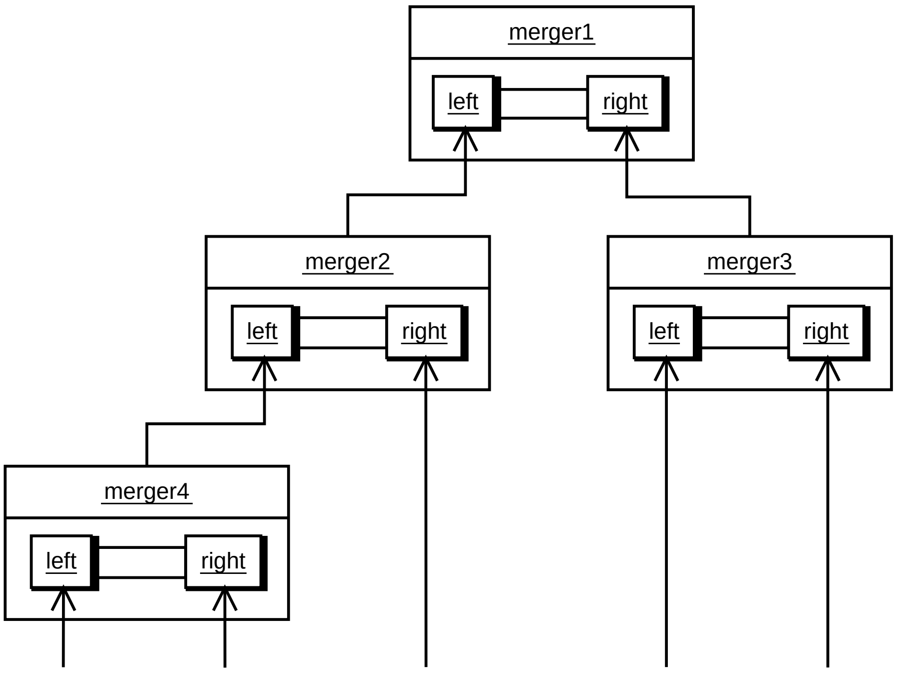


Figure 5.4: A 5-ary Merger tree

new children. Adapting the β case of Lemma 4.15, and assuming a focused leaf that lies below n_1, we get

$$\delta(n) = \delta(n_1) \cap \mathrm{cl}(n_2)^{\mathrm{old}} \cap \cdots \cap \mathrm{cl}(n_k)^{\mathrm{old}} \quad .$$

In other words, we have to compute the new closing instantiations which also close *all* of the other branches. As it is, this formula is not suitable for computation. If we only keep buffers for $\mathrm{cl}(n_k)$, the intersection of $k-1$ buffers has to be computed every time a new δ is processed, which is inefficient. On the other hand, introducing buffers for $\bigcap_{j \neq i} \mathrm{cl}(n_j)$ leads to k additional buffers which have to be updated every time a new δ is processed. One easily sees that updating these buffers requires the computation of the intersection of $k-2$ buffers is needed, so we recursively run into the same problem again.

The solution is to combine several mergers into a tree. We can build a k-ary Merger from $k-1$ binary Mergers, which are arranged in a tree structure of minimal depth. See Fig. 5.4 for a 5-ary Merger tree, for example. This solution requires one buffer for each of the k inputs and $k-2$ extra 'internal' buffers. Only $O(\log k)$ intersection and buffer update operations are needed for each δ.

The same structure of Mergers could be achieved by inserting appropriate parentheses. For instance, the structure in Fig. 5.4 is generated by $((\beta_1 \vee \beta_2) \vee \beta_3) \vee (\beta_4 \vee \beta_5)$. There are however some minor differences:

- In the collection of ancestries for pruning (see Sect. 5.6, the Merger tree can be considered a unit, represented for instance by the topmost Merger. If one of the outer MergerSinks receives an instantiation that can be passed through, it can be passed on directly to the topmost sink, without going through the inner Mergers. In particular, the inner MergerSinks do not need to do the ancestry checking step required for pruning.

- If the MergerSinks restrict instantiation domains, e.g. as suggested in Sect. 5.7, this calculation is not needed for the inner MergerSinks.
- In the case one of the new branches is propositionally closable (see Def. 5.2), one could, at least in theory reduce the depth of the Merger tree by restructuring it accordingly. However, this is hardly of practical relevance unless one regularly has problems with very large disjunctions which often lead to propositionally closable branches.

It remains to note that it does not necessarily speed up proof search to handle long disjunctions in this way. If large disjunctions are successively bisected, a good formula selection strategy might be able to postpone splitting on some of the disjunctions and thus be more efficient. However, such considerations are very dependent on the heuristics used and the problems to be solved. The purpose of this section was to point out how k-ary branching *can* be implemented, if this is wished.

5.10 Summary

In this chapter, we have discussed a number of refinements for an incremental closure prover. Some, like subsumption and indexing in instantiation buffers (Sect. 5.4 and 5.5) were specific to the incremental closure technique. Others, like pruning or universal variables (Sect. 5.6 and 5.7) showed how known refinements from backtracking provers may be integrated into our framework.

All of these refinements required relatively small modifications of the basic framework presented in Chapter 4. The next two chapters will cover two further families of refinements, namely simplification rules and equality handling. These will require a new ingredient: adding syntactic constraints to formulae in goals.

6 Simplification Rules

In this chapter, we shall extend our tableau calculus by adding the syntactic feature of constrained formulae. These will be used to introduce a family of very powerful tableau extension rules. It is conceivable, in principle, to adapt these rules to a backtracking proof procedure without constrained formulae. But using constraints reduces the need for backtracking in proof search. And we shall see that the incremental closure framework and constrained formulae are a perfect match. The same principle shall be used in Chapter 7 to obtain a calculus with built-in equality handling.

We will start out with a discussion of simplification rules for tableaux.

6.1 Simplification Rules for Tableaux

Non-clausal form analytic tableaux have a number of advantages over the proof procedures for clausal form implemented in most successful automated theorem provers. For instance, when the logic is enhanced by modal operators, clausal form cannot be used without previously translating the problems into first-order. Another case is the integration of automated and interactive theorem proving [ABH$^+$98, Gie98], where normal forms would be counter-intuitive. Unfortunately, standard non-clausal form tableaux tend to be rather inefficient, as many of the refinements available to clausal procedures are hard to adapt. Typical cases in point are unit resolution, especially for propositional provers like the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DLL62], the β^c rules of the KE calculus [DM94], the application of 'result substitutions' in Stålmarcks Procedure [SS98], and hyper tableaux [BFN96a]. The common feature of these techniques is that they involve inferences between several formulae derived from the formula to be proved, either by using one formula to simplify another one, or—for hyper tableau—making tableau expansions depend on the presence of certain literals on a branch. In the basic tableau calculus we have considered so far, the only action that takes more than one formula into account is branch closure.

In [Mas98], Massacci presents a simplification rule for propositional and modal tableau calculi. This rule is of the form

$$\mathit{simp} \quad \frac{\psi, \phi}{\psi[\phi], \phi}$$

which should be read as a new way of expanding a tableau in Def. 4.2, p. 17, namely:

> If there is a tableau for S that has a leaf n with goal $\{\psi, \phi\} \cup G$, then the tableau obtained by adding a new child n' with goal $\{\psi[\phi], \phi\} \cup G$ to n is also a tableau for S.

The interesting part of this rule is the notation $\psi[\phi]$, which denotes the formula that results form first replacing any occurrence of ϕ, resp. $\neg\phi$, in ψ by *true*, resp. *false* and applying a set of boolean simplifications to the result. Formally, for the propositional case, one defines

$$\psi[\phi] = \begin{cases} \mathit{true} & \text{if } \psi = \phi \\ \mathit{false} & \text{otherwise, if } \neg\psi = \phi, \\ \mathit{eval}^{\neg}(\psi_1[\phi]) & \text{otherwise, if } \psi = \neg\psi_1, \\ \mathit{eval}^{\wedge}(\psi_1[\phi], \psi_2[\phi]) & \text{otherwise, if } \psi = \psi_1 \wedge \psi_2, \\ \mathit{eval}^{\vee}(\psi_1[\phi], \psi_2[\phi]) & \text{otherwise, if } \psi = \psi_1 \vee \psi_2, \\ \psi & \text{otherwise.} \end{cases}$$

The *eval* functions perform boolean simplification as follows:

$$\mathit{eval}^{\neg}(\psi) = \begin{cases} \mathit{false} & \text{if } \psi = \mathit{true}, \\ \mathit{true} & \text{if } \psi = \mathit{false}, \\ \neg\psi & \text{otherwise} \end{cases}$$

$$\mathit{eval}^{\wedge}(\psi_1, \psi_2) = \begin{cases} \psi_2 & \text{if } \psi_1 = \mathit{true}, \\ \psi_1 & \text{if } \psi_2 = \mathit{true}, \\ \mathit{false} & \text{if } \psi_1 = \mathit{false} \text{ or } \psi_2 = \mathit{false}, \\ \psi_1 \wedge \psi_2 & \text{otherwise} \end{cases}$$

$$\mathit{eval}^{\vee}(\psi_1, \psi_2) = \begin{cases} \psi_2 & \text{if } \psi_1 = \mathit{false}, \\ \psi_1 & \text{if } \psi_2 = \mathit{false}, \\ \mathit{true} & \text{if } \psi_1 = \mathit{true} \text{ or } \psi_2 = \mathit{true}, \\ \psi_1 \vee \psi_2 & \text{otherwise} \end{cases}$$

This simplification operator can easily be extended to allow other connectives like implication and equivalence.

Massacci shows that proof procedures using this rule can subsume a number of other theorem proving techniques for propositional logic, e.g. the unit rule of DPLL [DLL62], the β^c rules of KE [DM94], regularity and hyper tableaux [BFN96a]. This is done mainly by specifying the strategy of when and where to apply the *simp* rule.

While DPLL and hyper tableaux are originally formulated for problems in clause normal form (CNF), the simplification rule is applicable for arbitrary non-normal form formulae, making it a good framework to generalize CNF techniques to the non-normal form case. Massacci gives variants of the simplification rule for various modal logics. In [Mas97], he also gives a variant of the rule for first-order free-variable tableaux. Unfortunately, this rule does *not* in general subsume first-order versions of unit-resolution, hyper tableaux etc., because it places strong restrictions on the instantiation of free variables.

This chapter presents variants of the simplification rule for first order logic which overcome this limitation, and which are particularly suited for use in an incremental closure prover. They were first introduced in [Gie00a], but the proofs of various theorems were only sketched there. See also [Gie03] for an alternative proof of theorem 6.8.

6.2 Simplification with Global Instantiation

Consider a tableau goal containing the formulae $pX \vee qX$ and $\neg pa$, where X is a (rigid) free variable. If X were instantiated with a, the disjunction could be simplified:

$$\begin{array}{ll} & (pa \vee qa)[\neg pa] \\ \rightsquigarrow & \mathit{eval}^{\vee}(pa[\neg pa], qa[\neg pa]) \\ \rightsquigarrow & \mathit{eval}^{\vee}(\mathit{false}, qa) \\ \rightsquigarrow & qa \end{array}$$

Our task is to find a version of this ground simplification that works with free variables. The step from a ground version to a free variable version of a rule or a proof is usually referred to as *lifting*.

In a backtracking prover, where rigid variables are globally instantiated, the obvious way of lifting the simplification rule consists in applying a substitution to the whole proof that unifies certain subformulae, so that a simplification becomes possible.

Such a rule would be formulated using the most general unifier (mgu) of the simplifying formula and some subformula of the simplified formula. A little care must be taken to prevent the instantiation of bound variables by such a unifier.

Definition 6.1 *An occurrence of a subformula ϕ in ψ is called* simplifiable*, if no variable occurring free in ϕ is bound by ψ.*

Example 6.2 px is simplifiable in $\forall y.(qy \wedge px)$, but not in $\forall x.(qy \wedge px)$. It is also simplifiable in $(\forall x.qx) \wedge px$, because the quantifier does not bind the x occurring free in px.

We have to take account of universal quantifiers in the definition of $\psi[\phi]$. We define:

$$(\forall x.\psi_1)[\phi] := \mathit{eval}^{\forall x}(\psi_1[\phi]) \quad ,$$

where

$$\mathit{eval}^{\forall x}(\psi) = \begin{cases} \mathit{true} & \text{if } \psi = \mathit{true}, \\ \mathit{false} & \text{if } \psi = \mathit{false}, \\ \forall x.\psi & \text{otherwise.} \end{cases}$$

Using this notion, a simplification rule with global instantiation can be given:

$$\frac{\psi, \phi}{\mu(\psi)[\mu(\phi)], \mu(\phi)}$$

where μ is an mgu of ϕ and some simplifiable subformula of ψ,
and μ is applied on all open goals.

While this approach is sound, it relies on the application of a global instantiation for the free variables. The problem with such a rule is that it introduces a new backtracking point, because the applied unifier might not lead to a proof. Not only does this make the rule unsuitable for an incremental closure prover. It is also problematic for a backtracking prover as efficiency will suffer if more backtracking points than necessary are introduced.

6.3 Constrained Formulae

A universal technique for avoiding the global application of substitutions is to decorate formulae with unification constraints. In the terminology of [GH03], we use *constrained formula tableaux.* A unification constraint C is a conjunction of syntactic equalities between terms or formulae, written as

$$s_1 \equiv t_1 \,\&\, \ldots \,\&\, s_k \equiv t_k \quad .$$

We use the symbol $\equiv$ for syntactic equality in the constraint language to avoid confusion with the meta-level $=$.

Definition 6.3 *Let $C = s_1 \equiv t_1 \,\&\, \ldots \,\&\, s_k \equiv t_k$ be a unification constraint. We define*

$$\mathrm{Sat}(C) = \{\sigma \in \mathcal{I} \mid \textit{for all } i,\ \sigma(s_i) \textit{ equals } \sigma(t_i) \textit{ syntactically}\}$$

to be the set of instantiations satisfying a constraint. A constraint C is called satisfiable, *if* $\mathrm{Sat}(C)$ *is not empty, which means that there is a simultaneous unifier for the pairs $\{s_i, t_i\}$. A constraint C* subsumes *a constraint D, iff* $\mathrm{Sat}(D) \subseteq \mathrm{Sat}(C)$. *Two constraints C and D are* equivalent, *iff* $\mathrm{Sat}(C) = \mathrm{Sat}(D)$.

A constrained formula *is a pair of a formula ϕ and a constraint C, written as $\phi \ll C$. The empty constraint, which is satisfied by all ground substitutions, is usually omitted.*

A goal *is henceforth a finite set of constrained formulae. For a goal G and an instantiation $\sigma \in \mathcal{I}$, we define*

$$\sigma(G) = \{\sigma(\phi) \mid (\phi \ll C) \in G \textit{ and } \sigma \in \mathrm{Sat}(C)\} \quad .$$

The intuition for a constrained forumla is to consider the formula ϕ as available for branch closure only if the free variables are instantiated in a way that satisfies the constraint.

Example 6.4 Consider a goal containing two constrained formulae

$$pX \ll X \equiv Z, \neg pa \ll Z \equiv b \quad .$$

The goal is not closed with this complementary pair under any instantiation σ, as unification forces X to be instantiated with a, the first constraint requires $\sigma(Z) = \sigma(X) = a$, while the second constraint imposes $\sigma(Z) = b$. Note that it is quite possible for variables to occur in a constraint but not in the accompanying formula, and vice versa.

We can replace rules which globally apply an mgu of two formulae ϕ and ψ by versions which add unification constraints of the form $\phi \equiv \psi$ to the resulting formulae. This is a local operation that does not require backtracking. The check that free variables are instantiated correctly for the given rule application is postponed to the branch/tableau closure test.

Example 6.5 Let a goal contain two formulae

$$pX \vee qX, \neg pa \quad .$$

> The simplification step of the previous section requires instantiation of X with a and yields a simplified formula qa. We can thus add the constrained formula $qa \ll X \equiv a$ to the goal.

Obviously, if formulae $\phi_i \ll C_i$ which already carry constraints are involved in a rule application, the conjunction $C_0 \,\&\, C_1 \ldots$ has to be passed on to the resulting formulae. This is referred to as *constraint propagation.* Constraints are propagated through rule applications, until a branch is closed. Closure between two literals $L \ll C$ and $\neg L' \ll C'$ is only allowed if the constraint $C \,\&\, C' \,\&\, L \equiv L'$ is satisfiable. Formally, the clauses of Def. 4.2 now become[1]

> If there is a tableau for S that has a leaf n with goal $\{\alpha_1 \wedge \alpha_2 \ll C\} \cup G$, then the tableau obtained by adding a new child n' with goal $\{\alpha_1 \ll C, \alpha_2 \ll C\} \cup G$ to n is also a tableau for S. (α-expansion)

and similarly for the other expansions, while branch closure is defined as follows:

> A goal G is *closed under an instantiation* σ, iff there is a complementary pair $\{\phi \ll C, \neg\psi \ll D\} \subseteq G$ with $\sigma \in \mathrm{Sat}(C) \cap \mathrm{Sat}(D)$, and $\sigma(\phi) = \sigma(\psi)$, or if there is a formula $(\mathit{false} \ll C) \in G$ with $\sigma \in \mathrm{Sat}(C)$.

The last sentence is needed because the simplification rules we shall introduce simplify formulae to *false* occasionally.

Using unification constraints, the simplification rule takes the form

$$\mathit{simp}^{c0} \quad \frac{\psi \ll C, \phi \ll D}{\psi \ll C, \phi \ll D, \mu(\psi)[\mu(\phi)] \ll (C \,\&\, D \,\&\, \phi \equiv \xi)}$$

where ξ is a simplifiable subformula of ψ,
μ is an mgu of ξ and ϕ,
and $C \,\&\, D \,\&\, \phi \equiv \xi$ is satisfiable

The simp^{c0} rule keeps an unsimplified copy of $\psi \ll C$ in the goal. This will change in later versions. An immediate consequence of keeping both original formulae is that completeness follows trivially from the completeness of the calculus without simplification. We only need to ascertain soundness.

[1] We are not taking universal variables into account at this point.

Theorem 6.6 *The tableau calculus with constrained formulae using the $simp^{c0}$ rule is sound, i.e. if a proof exists for a finite set of SNNF formulae, then that set is unsatisfiable.*

Proof. Let σ be the instantiation used to close the tableau. Consider the ground proof-tree obtained by replacing each goal of the proof by $\sigma(G)$. In particular, this implies omitting any formulae with constraints that are not satisfied by σ. Tableau expansions for formulae with unsatisfied constraints are left out. For a β expansion this means that only one of the branches needs to be kept, it doesn't matter which.

The definition of a closed constrained formula tableau guarantees that there is a complementary pair in each leaf goal of the resulting ground proof. Furthermore, as constraints can only be strengthened by rule applications, all proof steps needed to derive the complementary pair are still present in the reduced proof. Simplification steps are are transformed into instances of the ground simplification rule

$$\frac{\psi, \phi}{\psi, \phi, \psi[\phi]}$$

It remains to show that this ground version of the rule is sound. For this, it is sufficient to show that in any model where ψ and ϕ hold, $\psi[\phi]$ also holds, which immediately follows from the definition of $\psi[\phi]$. □

It is a little misleading to call $simp^{c0}$ a simplification rule, because the original formula $\psi \ll C$ has to be retained for completeness. Indeed, one cannot simply delete the original formula, because there is no guarantee that the closing instantiation of the proof will be such that the simplification is possible.

There is however an important special case: if the 'new' part of the constraint $D \& \phi \equiv \xi$ subsumes the 'old' part C, the original formula $\psi \ll C$ may be discarded, because this means that the simplification step is valid in all ground instances of ψ allowed by the constraint C. Let $simp^{c1}$ be the rule obtained with this modification:

$$simp^{c1} \quad \frac{\psi \ll C, \phi \ll D}{(\psi \ll C)^?, \phi \ll D, \mu(\psi)[\mu(\phi)] \ll (C \,\&\, D \,\&\, \phi \equiv \xi)}$$

where ξ is a simplifiable subformula of ψ,
μ is an mgu of ξ and ϕ,
$C \,\&\, D \,\&\, \phi \equiv \xi$ is satisfiable,
and $\psi \ll C$ is omitted from the result if $D \,\&\, \phi \equiv \xi$ subsumes C.

This allows real, destructive simplification steps in some cases, especially when no unification is needed.

Example 6.7 Consider a goal G containing constrained formulae

$$pX \lor qX \ll X \equiv a \,\&\, Y \equiv a,\ \neg pa \ll X \equiv Y \quad .$$

The constraints in question are $C = (X \equiv a \,\&\, Y \equiv a)$ and $(D \,\&\, \phi \equiv \xi) = (X \equiv Y \,\&\, pX \equiv pa)$. These are equivalent, so the latter one certainly subsumes the former. We can simplify the goal to

$$qa \ll X \equiv a \,\&\, Y \equiv a,\ \neg pa \ll X \equiv Y \quad .$$

Note that the resulting combined constraint is equivalent C whenever the subsumption holds.

Theorem 6.8 *The $simp^{c1}$ rule is sound. It is also complete, in the sense that a goal that can be closed under some σ after applying a sequence R of expansions steps, can still be closed under σ by a modified sequence R' after an application of the $simp^{c1}$ rule. Moreover, there is such an R' that is at most as long as the original R.*[2]

Proof. Soundness may be shown as for $simp^{c0}$, see Theorem 6.6.

Completeness would be difficult to show by a Hintikka-style argument, because of the destructive nature of $simp^{c1}$. Apart from that, such a proof would not yield the statement about the proof sizes. We shall construct R' from R by a proof transformation, in which rule applications on (descendents of) a discarded original formula $\psi \ll C$ can either be applied to (descendents of) the simplified or simplifying formula, or be discarded altogether. We have described an alternative way of performing this proof transformation in [Gie03].

In case the $simp^{c1}$ application does not discard the original formula, we can simply take $R' = R$. If the original formula ψ is discarded, we start by categorizing all subformulae of $\sigma(\psi)$. Still assuming formulae in SNNF, subformulae might be

1. untouched by the simplification step, or
2. changed by the simplification step in such a way that the top operator remains untouched, i.e. only proper subterms were modified, or
3. part of a subformula replaced by *true* or *false* in the base cases of the definition of $\psi[\phi]$, implying that they are also subformulae of $\sigma(\phi)$, or
4. part of a subformula removed or replaced by *true* or *false* by the *eval* function.

We can now analyze the tableau expansions needed by R to close the proof. Expansions steps in R are 'analytic', meaning that they derive only subformulae of previously present formulae. Expansion steps that do not operate on ψ or some formula derived from ψ can be applied unaltered. The same holds for subformulae derived from ψ, as long as they belong to the first two categories. At some point however, there will be α, resp. β-expansions in R, for a formula $A \wedge B$, resp. $A \vee B$, where only one subformula A is left, because of an evaluation of $A \wedge \mathit{true}$, resp. $A \vee \mathit{false}$ in $\psi[\phi]$. In the case of a β-expansion, we can just leave out the branch for B and use the one for A in R'. The proof will still be closable and smaller than the original one as we have at least one branch less. In the case of $A \wedge B$, where B was simplified to *true*, we need to analyze B. We recursively define a formula ξ to be *ϕ-implied*, if

- $\xi = \phi$, or
- $\xi = \xi_1 \wedge \xi_2$, where ξ_1 and ξ_2 are both ϕ-implied, or

[2]Note that the $simp^{c1}$ application is not counted in R'. So the overall proof size may increase by 1.

- $\xi = \xi_1 \vee \xi_2$, where at least one of ξ_1 and ξ_2 is ϕ-implied, or
- $\xi = \forall x.\xi_1$, where ξ_1 is ϕ-implied and x is not free in ϕ.

If one now goes through the definitions of $\psi[\phi]$ and the *eval* functions, remembering that we assume formulae in SNNF, one sees that B must be ϕ-implied in order to be simplified to *true*. One now proceeds as follows for the α-expansion of $A \wedge B$ and all following expansions of subformulae of B in R, maintaining the invariant that all formulae derived by such proof steps on the current branch are ϕ-implied: α- and γ-expansions are simply omitted. β-expansions are also omitted but one continues on the branch of R that receives the ϕ-implied disjunct. This is done until the expansions of R actually derive ϕ. One can then continue expansions on ϕ normally, as that formula has been left in the goal by the simplification step.

Each branch of the reduced proof corresponds to some branch of the original proof, with some expansion steps and formulae omitted. We know that all omitted formulae are ϕ-implied but not equal to ϕ, so they cannot be literals. The final goal of the transformed proof thus contains all the literals of the corresponding branch of the original proof. Therefore, the resulting proof is still closable.

We left out two special cases, namely that the simplification step simplifies the *whole* formula to *true* or *false*. In the *false* case, the branch is of course immediately closable. In the *true* case, the whole original formula ψ is ϕ-implied, and we can proceed as before for the $A \wedge$ *true* case. □

This proof can be extended to arbitrary first order formulae by using polarity of subformulae in the notion of ϕ-implied formulae, amongst other modifications. We are not going to discuss this in detail as the basic technique remains the same.

The *simp*^{c} rules enjoy an interesting relative termination property, which it shares with the α and β rules, namely that only a finite number of simplification steps can be performed without intervening γ-expansions, under certain side conditions.

Definition 6.9 *Two constrained formulae $\phi \ll C$, $\psi \ll D$ are called* variants*, if C and D are equivalent, and for all $\sigma \in \mathrm{Sat}(C)$, $\sigma(\phi) = \sigma(\psi)$.*

Example 6.10 $pX \ll X \equiv a$ and $pa \ll X \equiv a$ are variants. $pa \ll X \equiv a$ and $pa \ll X \equiv Y$ are not variants according to this definition, because the constraints are not equivalent.

We can now formulate our relative termination property:

Theorem 6.11 *Starting from a given tableau goal, only a finite number of α, β, and simp^{c0} rule applications without intervening applications of the γ rule are possible, if simp^{c0} is never applied twice to the same pair of constrained formulae, and any formula which is a variant of a formula already present on a branch is discarded. The same is true for the simp^{c1} rule.*

Proof. A formula ϕ can only be simplified by setting one of its subterms to *true* or *false*, and the resulting simplified formulae are all smaller than ϕ. So the number of distinct formulae that can be generated is finite. On the other hand, all constraints that could be generated are conjunctive combinations of existing constraints and syntactic equations between subformulae of formulae on a branch, so there can be only finitely many non-equivalent constraints. Accordingly, the number of non-variant constrained formulae must be finite. For $simp^{c1}$, formulae are occasionally discarded from a goal. This implies that even less rule applications are possible, so the same argument holds. □

As a practical consequence of this termination property, there is no need to interleave γ and $simp^c$ applications in a proof procedure to guarantee fairness. One can apply all viable simplifications before considering an application of the γ rule. Note that it is not clear how useful this property really is in a serious implementation. A similar property is discussed for a set of equality handling rules in Sect. 7.4.2, and a number of problems with it are discussed in Sect. 7.4.3.

6.4 Constrained Formulae and Incremental Closure

It is quite easy to adapt the incremental closure technique to work with constrained formula tableaux. The only place where a change is needed is in the handling of complementary pairs that initiates δ propagation. Where Lemma 4.15 on page 26 stated

$$\delta(n) = \mathrm{unif}(\phi, \psi)|_{\mathrm{fv}(n)} \setminus \mathrm{cl}(l)^{\mathrm{old}}$$

for the focused leaf $n = l$, we now have to take the constraints into account, getting

$$\delta(n) = (\mathrm{unif}(\phi, \psi) \cap \mathrm{Sat}(C) \cap \mathrm{Sat}(D))|_{\mathrm{fv}(n)} \setminus \mathrm{cl}(l)^{\mathrm{old}}$$

for a new complementary pair $\phi \ll C, \psi \ll D$. As δ propagation is also implemented using constraints, this simply translates to propagating the constraint

$$\phi \equiv \psi \,\&\, C \,\&\, D$$

instead of just $\phi \equiv \psi$.

Although tableaux with constrained formulae and incremental closure are a very natural match, it should be mentioned that constrained formula tableaux may be used in a backtracking procedure as well. There are at least two different ways to do this.

One possibility is to take the constraints into account when a branch is closed by computing a unifier that not only unifies the complementary pair but also satisfies the attached constraints. If there is no such unifier, the branch closure is not allowed. This unifier can then be applied on the whole tableau as usual.

Another possibility is to use backtracking without global instantiation by computing the unification constraint $\phi \equiv \psi \,\&\, C \,\&\, D$ when a branch is to be closed, and adding it to an accumulated global constraint. Any rule application or branch closure that produces a constraint which is incompatible with the accumulated constraint is forbidden. Of course, simultaneous backtracking is required for the accumulated constraint and the tableau if this scheme is used.

We shall briefly reconsider these alternatives at the end of Sect. 7.3.1 in the context of equality handling rules.

6.5 Dis-Unification Constraints

Although the *simp*^{c1} rule permits the original formula $\psi \ll C$ to be deleted from the goal in some cases, it will usually have to be kept. This can lead to redundancies as exemplified by the following tableau branch for the set of formulae $\{pa, qa, \neg pX \vee \neg qX \vee rX\}$. We write only the new formula in each goal to make the tableau more readable. All the old formulae remain in each goal.

$$\begin{aligned} 1 &: \{pa, qa, \neg pX \vee \neg qX \vee rX\} \\ 2 &: \{\neg qa \vee ra \ll X \equiv a, \ldots\} \\ 3 &: \{\neg pa \vee ra \ll X \equiv a, \ldots\} \end{aligned}$$

Goal 2 is the result of simplifying the disjunction with pa, goal 3 comes from a simplification of the same formula with qa. After generation of 2, one would intuitively expect the step leading to goal 3 to be redundant: The constraint introduced in 3 demands that X be instantiated with a. If this were done globally, we would get a ground proof starting as follows:

$$\begin{aligned} 1 &: \{pa, qa, \neg pa \vee \neg qa \vee ra\} \\ 2 &: \{pa, qa, \neg qa \vee ra\} \end{aligned}$$

The original disjunction is kept by the first-order simplification rule only in case X is *not* instantiated with a. In the ground case, qa can now be used to simplify the formulae $\neg qa \vee ra$ introduced in goal 2, leading to a final goal

$$3 : \{pa, qa, ra\} \quad .$$

One can easily see that in the presence of a large formula and many simplifying literals, a large number of redundant formulae may be generated by the *simp*^{c0} and *simp*^{c1} rules. We need to modify the simplification rule to forbid multiple simplifications of one formula with the same instantion.

6.5.1 Simplification with Dis-unification constraints

One way of solving this problem is to use constraints to record instantiations under which a formula *could have been* discarded in a ground calculus. To do this, we have to require the constraint language to be closed under negation (denoted '!') as well as conjunction. The resulting constraint satisfiability problems are known as *dis-unification* problems, see e.g. [Com91], so we will talk of dis-unification or DU constraints.

A little care has to be taken with the semantics of DU constraints: Some DU constraints that are not satisfiable in a given signature might become satisfiable when the signature is extended. E.g., $!\,X \equiv a$, is not satisfiable in a signature consisting only of the constant symbol a, but it becomes satisfiable if the signature is extended by new constant or function symbols. In our context, satisfiability should be considered with respect to a possibly extended signature. For one thing, it turns out that satisfiability and subsumption can be checked more efficiently with this definition. The same effect for term ordering constraints has previously been noted in [NR95a]. And if we decide to extend the calculus with a δ rule for existential quantification, new skolem symbols might be introduced at a later point, extending the signature, so we would need the extended signature semantics anyway.

Using DU constraints, the simplification rule can be reformulated as follows:

$$\textit{simp}^{c2} \quad \frac{\psi \ll C,\ \phi \ll D}{\psi \ll C \,\&\, !(D \,\&\, \phi \equiv \xi),\ \phi \ll D,\ \mu(\psi)[\mu(\phi)] \ll (C \,\&\, D \,\&\, \phi \equiv \xi)}$$

where ξ is a simplifiable subformula of ψ,
μ is an mgu of ξ and ϕ,
and $C \,\&\, D \,\&\, \phi \equiv \xi$ is satisfiable

This rule differs from $\textit{simp}^{c0}$ in that the constraint of the original formula ψ is changed by adding $!(D \,\&\, \phi \equiv \xi)$. What this means is that the formula is no longer available for simplification steps requiring an instantiation under which *this* simplification would have been possible.

We can allow the original formula to be discarded if the new constraint $C\&!(D\&\phi \equiv \xi)$ is unsatisfiable, because it could never participate in a branch closure in that case. It turns out that this new constraint is unsatisfiable if and only if $D \,\&\, \phi \equiv \xi$ subsumes C. In other words the $\textit{simp}^{c2}$ rule will discard the original formula in exactly the same cases as $\textit{simp}^{c1}$.

Example 6.12 For the formula set above, we now get the following derivation:

$$\begin{array}{rl} 1: & \{pa,\ qa,\ \neg pX \lor \neg qX \lor rX\} \\ 2: & \{\ \underline{\neg qa \lor ra \ll X \equiv a}, \\ & \phantom{\{}\ pa,\ qa,\ \neg pX \lor \neg qX \lor rX \ll \underline{!\,X \equiv a}\} \end{array}$$

The constraint $!\,X \equiv a$ now prevents the simplification of the original disjunction with qa. Instead, the new formula can be simpified with qa, and in this case it can even be discarded, as no unification is needed. We thus get the following goal:

$$3: \{\underline{ra \ll X \equiv a},\ pa,\ qa,\ \neg pX \lor \neg qX \lor rX \ll !\,X \equiv a\}$$

If X is instantiated with a, this corresponds closely to the ground proof shown above. One arrives at the same goal by simplifying first with qa, and then with pa. In a sense, the $\textit{simp}^{c2}$ rule removes redundancy due to commutation of simplification steps.

The $\textit{simp}^{c2}$ rule has similar properties as the previous versions of the simplification rule.

Theorem 6.13 *The simp^{c2} rule is sound. It is also complete in the sense of Theorem 6.8.*

Proof. Soundness follows from Theorem 6.6, as stricly less rule applications are possible than with *simp*^{c0}. Completeness is shown using the same technique as for Theorem 6.8. We take the addition of the DU constraint into account by considering three cases. If the closing constraint σ does not satisfy C, any proof steps on $\psi \ll C$ can be left out anyway, as they do not contribute to the closure of the subtableau. If $\sigma \in \mathrm{Sat}(C)$, but $\sigma \notin \mathrm{Sat}(D \,\&\, \phi \equiv \xi)$, we can perform all extensions as in R, because σ satisfies the changed constraint $C \,\&\, !(D \,\&\, \phi \equiv \xi)$. Finally, if $\sigma \in \mathrm{Sat}(C)$ and $\sigma \in \mathrm{Sat}(D \,\&\, \phi \equiv \xi)$, we perform the proof transformation as in the proof of Theorem 6.8, considering the original formula to be deleted, because its constraint and the constraints of any formulae derived from it is not satisfied by σ. □

Theorem 6.14 *Starting from a given tableau goal, only a finite number of α, β, and simp^{c2} rule applications without intervening applications of the γ rule are possible, under the same conditions as for Theorem 6.11.*

Proof. The *simp*^{c2} allows even less rule applications than the *simp*^{c1} rule, so the property follows from Theorem 6.11. □

We introduced formulae with unification constraints in Sect. 6.3 as a universal means of translating rules which require global instantiation and backtracking into local rules which can do without backtracking. In this section, we introduced dis-unification constraints as a means of encoding another kind of destructiveness, namely the deletion of formulae from goals for the purpose of excluding rule applications. Note that in classical predicate logic, there is no other reason to delete a formula from a tableau goal due to the monotonicity properties of the logic and the calculus. DU constraints can thus also be considered a universal technique for adapting refinements from backtracking provers to the incremental closure framework.

6.5.2 Conjunctive DU Constraints

The drawback of this variant of the simplification rule is the high cost of checking satisfiability for arbitrary dis-unification constraints. As a compromise, one can keep unification (U) and dis-unification (DU) parts of constraints separate and weaken the DU part of constraints, when this makes calculations more convenient. The unification part has to be preserved, as it is needed for soundness. But the DU part only serves to reduce redundancy during proof search, so it is legitimate to simplify constraint handling to improve the overall performance.

Definition 6.15 *A* conjunctive DU constraint *is a constraint of the form*

$$C_0 \,\&\, !\, C_1 \,\&\, !\, C_2 \ldots \quad ,$$

where the C_i are conjunctive unification constraints as defined in Sec. 6.3, that is without negation. C_0 is the unification (U) part *and* $!\,C_1 \,\&\, !\,C_2 \ldots$ *the* dis-unification (DU) part *of the constraint.*

Satisfiability (for possibly extended signatures) is fairly easy to check for constraints of that form, as we will show in Sect. 6.5.3. The problem is that $!(D \,\&\, \phi \equiv \xi)$ no longer has this form if the constraint D of the simplifying formula ϕ has a non-empty DU part. The solution is to drop the DU part of D when $\phi \ll D$ is used to simplify another formula. We shall illustrate this strategy in an example.

Example 6.16 Let a goal contain three formulae

$$F = pa, \quad G = pX \land qb, \quad H = (pX \land qY) \land rc \quad .$$

Now G can be simplified with F with a unifier $\mu = [X/a]$, giving

$$\mu G[\mu F] \ll X \equiv a \quad = \quad qb \ll X \equiv a \quad .$$

Also, H can be simplified by G with a unifier $\nu = [Y/b]$, giving

$$\nu H[\nu G] \ll Y \equiv b \quad = \quad rc \ll Y \equiv b \quad .$$

Using full DU constraint handling, we get the following derivation if first G, then H are simplified:

$$\{F,\ G,\ H\}$$

$$\{\underline{\mu G[\mu F] \ll X \equiv a},\ F,\ G \ll \underline{!\,X \equiv a},\ H\}$$

$$\{\underline{\nu H[\nu G] \ll Y \equiv b \,\&\, !\,X \equiv a},\ \mu G[\mu F] \ll X \equiv a,\ F,$$
$$\underline{G \ll !\,X \equiv a,\ H \ll} \underline{!(Y \equiv b \,\&\, !\,X \equiv a}\}$$

The final constraint of H is no longer in the desired form, and it cannot be transformed into an equivalent constraint of that form either. The suggested solution would omit the DU constraint $!\,X \equiv a$ of G when it is used to simplify another formula. This gives the following final goal, in which all constraints are of the desired form:

$$\{\underline{\nu H[\nu G] \ll Y \equiv b},\ \mu G[\mu F] \ll X \equiv a,\ F,\ G \ll !\,X \equiv a,\ H \ll \underline{!\,Y \equiv b}\}$$

The $simp^{c2}$ rule is still sound with this modification, as all rule applications would also have been possible with $simp^{c0}$, and all generated constraints are at still at least as strong as the ones $simp^{c0}$ would have generated. On the other hand, this modification allows more rule applications than the original $simp^{c2}$ with full DU constraint propagation, so the calculus remains complete.

In a way, the $simp^{c0}$ rule gives an upper bound on allowed rule applications to preserve soundness, while $simp^{c2}$ is a lower bound needed to preserve completeness. Any strategy for weakening constraints or discarding formulae which lies between these two is still sound and complete.

There is another place where the DU parts of constraints can (and should) be discarded, namely before δ propagation. Remember that the DU constraints are only meant to restrict rule applications in a completeness preserving way. If the tableau can be closed by an instantiation which violates the DU constraints of some of the complementary pairs, the closure is still sound. The procedure is still complete if we choose to propagate DU constraints, but that would make the tableau harder to close, which is not desirable.

The same observation holds for any other kind of constraints which are only introduced to restrict rule applications, but which are not necessary for soundness, like the syntactic ordering constraints used for equality handling in the following chapter, for instance.

6.5.3 Checking Satisfiability of Conjunctive DU Constraints

We shall now see that checking the satisfiability of conjunctive DU constraints is comparatively simple, if one adopts the extended signature semantics. It can be reduced to checking subsumption of conjunctive unification constraints. We start by some well-known facts about most general unifiers.

Proposition 6.17 *If a conjunctive unification constraint*

$$C \quad = \quad l_1 \equiv r_1 \,\&\, \cdots \,\&\, l_k \equiv r_k$$

is satisfiable, then there exists an idempotent most general unifier μ_C, i.e. a unification such that

- $\mu_C = \mu_C \circ \mu_C$,
- $\mu_C(l_i) = \mu_C(r_i)$ *for* $i = 1, \ldots, k$, *and*
- *For all instantiations $\sigma \in \mathrm{Sat}(C)$, there is an instantiation $\tau \in \mathcal{I}$ such that $\sigma = \tau \circ \mu_C$.*

The following lemma extends the connection between constraints and most general unifiers.

Lemma 6.18 *Let C and D be satisfiable conjunctive unification constraints.*

1. $\sigma \in \mathrm{Sat}(C)$ *iff* $\sigma = \sigma \circ \mu_C$ *for any* $\sigma \in \mathcal{I}$.
2. *C subsumes D iff* $\mu_D = \mu_D \circ \mu_C$.

Proof. Claim 1.: If $\sigma \in \mathrm{Sat}(C)$, then there is a $\tau \in \mathcal{I}$ with $\sigma = \tau \circ \mu_C$, hence

$$\sigma = \tau \circ \mu_C = \tau \circ \mu_C \circ \mu_C = \sigma \circ \mu_C \quad .$$

Conversely, if $\sigma = \sigma \circ \mu_C$, then for any equation $l \equiv r$ of C,

$$\sigma(l) = \sigma(\mu_C(l)) = \sigma(\mu_C(r)) = \sigma(r) \quad ,$$

so σ is indeed a solution of C.

Claim 2.: Note that two terms s, t are equal, iff for all $\sigma \in \mathcal{I}$, $\sigma(s) = \sigma(t)$. Hence, two substitutions μ, μ' are equal iff for all $\sigma \in \mathcal{I}$, $\sigma \circ \mu = \sigma \circ \mu'$.

Let C subsume D. For any $\sigma \in \mathcal{I}$, we have $\sigma \circ \mu_D = \sigma \circ \mu_D \circ \mu_D$ by idempotency. Hence, $\sigma \circ \mu_D \in \mathrm{Sat}(D)$, according to the first part of this lemma. Because C subsumes D, we also have $\sigma \circ \mu_D \in \mathrm{Sat}(C)$, from which $\sigma \circ \mu_D = \sigma \circ \mu_D \circ \mu_C$. As this holds for arbitrary σ, it follows that $\mu_D = \mu_D \circ \mu_C$.

Conversely, assume that $\mu_D = \mu_D \circ \mu_C$, and let $\sigma \in \mathrm{Sat}(D)$. Then,

$$\sigma = \sigma \circ \mu_D = \sigma \circ \mu_D \circ \mu_C = \sigma \circ \mu_C \quad ,$$

and hence $\sigma \in \mathrm{Sat}(C)$. This holds for arbitrary σ, and thus C subsumes D. □

We can now prove a simple characterization of the satisfiability of conjunctive DU constraints.

Theorem 6.19 *A conjunctive DU constraint*

$$C \quad = \quad C_0 \,\&\, !\, C_1 \,\&\, \cdots \,\&\, !\, C_k,$$

where C_0 to C_k are conjunctive unification constraints, is satisfiable by an instantiation to a possibly extended signature, iff

- *C_0 is satisfiable, and*
- *$C_0 \,\&\, !\, C_i$ is satisfiable, that is C_i does not subsume C_0, for $i = 1, \ldots, k$.*

Proof. First, let C be satisfiable. We choose some $\sigma \in \mathrm{Sat}(C)$. Due to the semantics of constraints, $\sigma \in \mathrm{Sat}(C_0)$, and $\sigma \notin \mathrm{Sat}(C_i)$ for $i = 1, \ldots, k$. Hence none of the C_i subsumes C_0.

For the converse, let C_0 be satisfiable and none of the C_i subsume C_0 for $i > 0$. We may assume without loss of generality that all C_i for $i = 1, \ldots, k$ are satisfiable. If some were unsatisfiable, we could leave them out without changing $\mathrm{Sat}(C)$. We now pick corresponding mgus $\mu_i = \mu_{C_i}$ for C_0 to C_k. Let V be the set of all variables occurring in $\mu_i(x)$ for any $x \in \mathrm{dom}\,\mu_i$, $i = 0, \ldots, k$. We introduce a new constant symbol c_X for each $X \in V$, plus an additional constant symbol d. We define an instantiation $\delta \in \mathcal{I}$ by

$$\delta(X) := \begin{cases} c_X & \text{if } X \in V, \\ d & \text{otherwise.} \end{cases}$$

Because of the new constant symbols used, δ is a bijection between terms over the old signature with variables in V and ground terms of the new signature. Hence two substitutions μ, μ' over the old signature with variables in V are equal iff $\delta \circ \mu = \delta \circ \mu'$. We now define $\sigma := \delta \circ \mu_0$. We will prove that $\sigma \in \mathrm{Sat}(C)$, by showing that $\sigma \in \mathrm{Sat}(C_0)$ and $\sigma \notin \mathrm{Sat}(C_i)$ for $i > 0$.

As μ_0 is an mgu for C_0, it follows that $\sigma = \delta \circ \mu_0 \in \mathrm{Sat}(C_0)$. Assume that $\sigma \in \mathrm{Sat}(C_i)$ for some $i > 0$. Due to Lemma 6.18, point 1, $\sigma = \sigma \circ \mu_i$, and thus $\delta \circ \mu_0 = \delta \circ \mu_0 \circ \mu_i$. Hence, due to the properties of δ, $\mu_0 = \mu_0 \circ \mu_i$, so C_i subsumes C_0, which is a contradiction. □

6.6 Simplification with Universal Variables

In practice, the simplification rules as outlined above tend to require a lot of instances of γ-formulae. This can lead to a further redundancy problem.

Example 6.20 Given the formulae

$$\{pa,\ pb,\ pc,\ \forall x.\neg px \vee qx\} \quad ,$$

one can apply a γ expansion, yielding a formula $\neg pX \vee qX$. This can be simplified by the various p-literals, to

$$qa \ll X \equiv a,\ qb \ll X \equiv b,\ qc \ll X \equiv c$$

But these literals have mutually contradictory constraints, so any further rule application or closure can involve at most one of these literals. One needs three instances of the γ formula to produce *compatible* literals

$$qa \ll X_1 \equiv a,\ qb \ll X_2 \equiv b,\ qc \ll X_3 \equiv c$$

But with three instances, not only these three useful literals are deducible, but a total of nine q-literals coming from the simplification of each instance $\neg pX_i \vee qX_i$ with each of the three p-literals:

$$\begin{array}{l} qa \ll X_1 \equiv a,\ qb \ll X_1 \equiv b,\ qc \ll X_1 \equiv c, \\ qa \ll X_2 \equiv a,\ qb \ll X_2 \equiv b,\ qc \ll X_2 \equiv c, \\ qa \ll X_3 \equiv a,\ qb \ll X_3 \equiv b,\ qc \ll X_3 \equiv c \end{array}$$

As all of these will subsequently be used to simplify any subformula qY on the branch, this can quickly lead to a huge (though finite) number of rule applications.

One way to reduce the number of distinct instances of γ formulae is to use universal variables as introduced in Sect. 5.7. A universal variable may be instantiated differently for consecutive applications of the simplification rule. We shall write $[\bar{X}]\phi \ll C$ for a constrained formula which is universal with respect to some variables $\bar{X}$. Before giving a formal definition of the simplification rule with universal variables, let us continue our example to give an intuition of how it works.

Example 6.21 Starting from the same set of formulae as in Example 6.20, the γ-expansion now yields $[X](\neg pX \vee qX)$. The variable X is universal, so we may instantiate it individually for the three applications of the simplification rule, giving

$$[X]qa \ll X \equiv a,\ [X]qb \ll X \equiv b,\ [X]qc \ll X \equiv c$$

The important thing here is that X is still universal for these simplified formulae. They are thus no longer incompatible, because X may be instantiated differently for each of them. Due to the fact that the constraint completely determines the instantiation of X in each case, the universal variable and constraint could be eliminated altogether in these literals, giving

$$qa,\ qb,\ qc \quad .$$

That is a technical optimization however, which is not strictly necessary.

What was said about universal varibales in Sect. 5.7 also applies to constrained formula tableaux. But we need to modify the definition of universal variables to take the constraint into account.

Definition 6.22 *Let $(\phi \ll C) \in G$ be a constrained formula occurring in a goal G of a constrained formula tableau, and X some free variable. Let $\bar{G}$ be the union of G and all goals above G in the tableau. $\phi \ll C$ is called* universal with respect to X *if $\sigma([X/t](\phi))$ is a logical consequence of $\sigma(\bar{G})$ for all $\sigma \in \mathcal{I}$ and all ground terms t such that $\sigma \circ [X/t] \in \mathrm{Sat}(C)$.*

To simplify the formulation of the simplification rule, we can require the universal variables of two formulae to be made disjoint by renaming before we apply a simplification step. We need a criterion for detecting universal variables in the result of a simplification step, similar to Lemma 5.9. This criterion is simply that any free variable appearing in the simplified formula that was universal in the original formulae, remains universal.

Lemma 6.23 *Consider an application of the simp^{c2} rule, where the universal variables of the two original formulae are disjoint. Then the resulting simplified formula is universal with respect to any variable for which one of the original formulae was universal. The universal variables of the original formulae are unchanged.*

Proof. We show only the case for a universal variable X of the original formula $\psi \ll C$, the case for a universal variable of $\phi \ll D$ being very similar. Let G and G' be the goals before and after the expansion. Take an arbitrary $\sigma \in \mathcal{I}$ and some ground term t with $\sigma \circ [X/t] \in \mathrm{Sat}(C \,\&\, D \,\&\, \phi \equiv \xi)$. We abbreviate $\sigma' := \sigma \circ [X/t]$. Since $\sigma' \in \mathrm{Sat}(C)$, $\sigma'(\psi)$ is a logical consequence of $\bar{G}$. But we also have $\sigma'(\phi) = \sigma'(\xi)$, so $\sigma'(\psi)$ may be (ground) simplified to $\sigma'(\psi)[\sigma'(\phi)]$, which is thus also a consequence of $\bar{G}$. As σ' is a unifier of ϕ and ξ, and μ is an mgu, it follows from Lemma 6.18 that this formula is syntactically equal to $\sigma'(\mu(\psi)[\mu(\phi)])$, which is thus also a consequence of $\bar{G}'$. □

If we annotate the *simp*^{c2} rule, with the sets $[\bar{X}]$ of universal variables, we get the following:

$$\mathit{simp}^{c2u} \quad \frac{[\bar{X}]\psi \ll C,\ [\bar{Y}]\phi \ll D}{\begin{array}{c}[\bar{X}]\psi \ll C \,\&\, !(D \,\&\, \phi \equiv \xi), \quad [\bar{Y}]\phi \ll D, \\ [\bar{X} \cup \bar{Y}]\mu(\psi)[\mu(\phi)] \ll (C \,\&\, D \,\&\, \phi \equiv \xi)\end{array}}$$

where ξ is a simplifiable subformula of ψ,
μ is an mgu of ξ and ϕ,
and $C \,\&\, D \,\&\, \phi \equiv \xi$ is satisfiable

If we use universal variables, the 'simplifiable subformula' condition could be relaxed to permit simplification of subformulae with bound variables under certain conditions. In a non-SNNF setting, simplification below existential quantifiers is also possible, for instance simplification of $\exists y.(py \vee qy)$ with $[X].\neg pX$ to $\exists y.qy$. A formal definition of this becomes rather technical however, so we shall keep the restriction to simplify our discussion.

The *simp*^{c2u} rule is sound and complete for a free variable tableau calculus with universal variables. Completeness can be shown by a combination of the technique of [Gie98], Sect. 7.4, for showing completeness of tableaux with universal variables, and the proof transformation technique of Theorems 6.8 and 6.13. By contrast, the termination property does not hold anymore, if universal variables are used. To apply the *simp*^{c2u} rule, it is necessary in general to rename universal variables in the original formulae to make them disjoint. But this renaming destroys termination.

Example 6.24 Consider the formulae pa and $[X]\neg pX \vee pfX$. With simplification and renaming of universal variables, one can consecutively deduce the following constrained literals without any intervening applications of other expansion rules.

$$\begin{gathered}[X_1].pfa \ll X_1 \equiv a \\ [X_1, X_2].pffa \ll X_1 \equiv a \,\&\, X_2 \equiv fa \\ [X_1, X_2, X_3].pfffa \ll X_1 \equiv a \,\&\, X_2 \equiv fa \,\&\, X_3 \equiv ffa \\ \text{etc.}\end{gathered}$$

This means that in a prover using the *simp*^{c2u} rule, simplification and β- or γ-expansions[3] need to be interleaved to retain fairness. How exactly this should be done falls into the realm of heuristics, but we can point out some considerations, based on practical experience with our implementation PrInS.

As the *simp*^{c2u} rule *without* renaming obviously enjoys the termination property, renaming and γ/β-expansion can be interleaved, but that would amount to effectively ignoring universality most of the time. Simplification steps tend to be much more useful for most problems than further rigid variable introducing expansions.

It is interesting to note that there are many problems, Schubert's 'Steamroller' [Sti86] being a particularly prominent example, in which simplification with universal variables actually *does* terminate. This is true, in particular, when some *simplification strategy*, like the hyper strategy discussed in the next section is used, which does not apply arbitrary simplification steps. To handle such cases efficiently, it is advisable to equip a proof procedure with some sort of cycle detection that only interleaves simplifier applications with γ rules, if they threaten to lead to infinite simplification sequences. One possibility is to set a limit to the size of inferred formulae, which can be incrementally increased as the tableau is expanded. This would always allow rule applications which really simplify a formula in the sense of making it smaller.

In PrInS, a heuristic based on histories of formulae is employed: For each goal, a list of deferred rule applications is kept. Pending repetitions of γ/β-expansions are put

[3]Recall that in Sect. 5.7, we discussed the two possibilities of either repeated γ-expansions on the same formula, or repeated β-expansions with renaming of variables.

into this list, as well as simplifications steps where a formula ψ is to be simplified with some ϕ that is itself a descendant of some formula obtained by simplifying ψ. Deferred rule applications are rescheduled when no other expansions are possible anymore. This heuristic is simple to implement—the history bookkeeping is similar to that for pruning, see Sect. 5.6—and it has proven to be sufficiently effective in many cases.

6.7 Emulating Hyper Tableaux

Although we have identified cases in which we can discard the original formula in a simplification step, we should not forget that this is not possible in general. With the simp^{c2} and simp^{c2u} rules, we can at least strengthen the constraint of the original formula, but this does not change the fact that our so-called simplification rule actually makes goals larger in most cases. The reason of using the name simplification is the analogy to the ground and propositional simplification rules which our first-order version subsumes.

In order for the simplification rules to be useful in a prover, one needs a *simplification strategy*, that is a strategy that prescribes when to apply which kinds of simplification steps.

We claimed at the beginning of this chapter that our simplification rules are capable of simulating first-order versions of various refinements, including hyper tableaux, and regularity. We have yet to show that this has been achieved. In this section, we shall describe a simplification strategy that implements a non-clausal analogue of hyper tableaux, and in Sect. 6.8, we shall consider the regularity restriction which was already mentioned in Sect. 5.8.

Hyper tableaux are defined for problems stated in clause normal form (CNF), see [BFN96a, BFN96b, Küh97, Bau98]. For clause tableaux, it is customary not to include the clauses in the tableau itself. Instead, one only uses the literals which result from expanding the tableau with a clause. Hyper tableaux permit an expansion with a clause only if all new branches which receive negative[4] literals of the clause are immediately closed, as shown in Fig. 6.1. All inner nodes are thus positive literals.

Alternatively, one can take the view of interpreting the clauses as tableau expansion rules themselves. In this view, a clause is 'fired' if there is a positive literal on a branch for every negative literal of the clause. The tableau is then extended by one new branch for each of the positive literals of the clause. It is very intuitive to write clauses as implications to support this view, see Fig. 6.2.

In the first-order case, one has to apply a substitution to unify the negative literals of the clause with corresponding positive literals on the branch. The way variables are handled differs between the various presentations of hyper tableaux. While [Bau98] uses universal variables in branch literals where possible, that version of hyper tableaux does not use rigid variables. Instead, it uses 'purifying substitutions' which generate ground instances of clauses if necessary. This happens whenever a variable is shared

[4]We consider *positive* hyper tableaux here. It is possible to exchange the roles of positive and negative literals, which leads to negative hyper tableaux.

Clause set:

A
B
$\neg D$
$\neg E$
$\neg A \vee \neg B \vee C$
$\neg C \vee D \vee E$

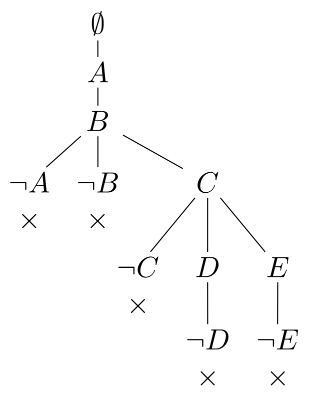


Figure 6.1: A propositional hyper tableau.

Clause set:

$true \rightarrow A$
$true \rightarrow B$
$D \rightarrow false$
$E \rightarrow false$
$A \wedge B \rightarrow C$
$C \rightarrow D \vee E$

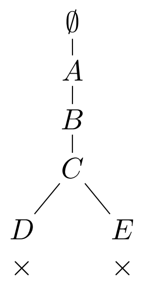


Figure 6.2: A ‘rule view’ hyper tableau.

Clause Set:
$p(a,b)$
$p(x,z) \to p(x,y) \vee p(y,z)$
$p(x,f(x)) \to q(x)$

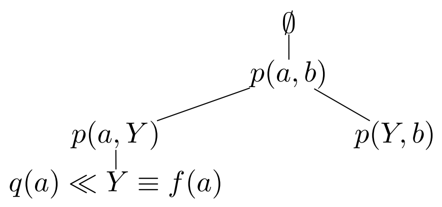


Figure 6.3: A first-order hyper tableau using rigid variables and constrained formulae

between two positive literals of a clause without occurring in any of the negative literals. A version described in [Küh97] uses rigid variables in such situations, using copies of clauses to avoid destructive instantiation, in a way somewhat similar to the disconnection calculus [Bil96]. In [vE01], a variant with rigid variables and constraints is proposed, but constraints are attached to branches instead of formulae as is done in our calculi.

We can define a version of first-order hyper tableaux using constrained formulae. As usual, we use rigid variables when necessary, and constraints to capture necessary instantiations. An example of this approach is given in Fig. 6.3. After putting the literal $p(a,b)$ on the branch using the first clause, we expand the tableau using the second clause, where $p(x,z)$ is instantiated with $p(a,b)$. As the two branches share the new variable Y, this has to be rigid. Subsequent expansion of the left branch with the third clause is possible only if Y is instantiated to fa. This restriction is captured in the constraint of the generated literal.

These rigid variable, constrained formula hyper tableaux can be emulated using our simplification rule with universal variables and a suitable simplification strategy. From now on, we shall consider normal analytic tableaux again. The set of clauses is given as a set of universally quantified disjunctions of literals. For the clause set of Fig. 6.3, we would start with a goal

$$\{p(a,b),\ \forall x,y,z(\neg p(x,z) \vee p(x,y) \vee p(y,z)),\ \forall x.(\neg p(x,f(x)) \vee q(x))\} \quad .$$

γ-expansion then leads to disjunctions of literals where all free variables are universal:

$$\{p(a,b),\ [X,Y,Z]\neg p(X,Z) \vee p(X,Y) \vee p(Y,Z),\ [X]\neg p(X,f(X)) \vee q(X)\} \quad .$$

A hyper tableau expansion step is now simulated by simplifying away negative literals in the disjunctions using the goal's positive literals, and applying β-expansion only when no negative literals are left. In our example we would first derive

$$[X,Y,Z]p(a,Y) \vee p(Y,b) \ll X \equiv a \,\&\, Z \equiv b \quad ,$$

where the universal variables X, Z may be eliminated as their instantiation is completely determined by the constraint, giving

$$[Y]p(a,Y) \vee p(Y,b) \quad .$$

A β-expansion then yields two branches with $p(a, Y')$ and $p(Y', b)$ respectively, where Y' is a new rigid variable. This β-expansion will be repeatedly applied by the proof procedure introducing a new rigid variable each time. Similarly, simplifying the disjunction $[X]\neg p(X, f(X)) \vee q(X)$ with $p(a, Y')$ will lead to a literal $q(a) \ll Y' \equiv f(a)$, which requires no β-expansion.

This principle can be captured in a strategy, which is an ensemble of rules describing when rule applications are allowed. In this case, the strategy would be:

> Use simplification only to simplify any negative literals inside disjunctions with positive literals occurring on the branch. Use β-expansion only for disjunctions which contain no negative literals.

As this strategy requires simplifying away *all* negative literals in a disjunction before a β-expansion is allowed, we can further restrict the strategy to simplify away the negative literals in a fixed order:

> Use simplification only to simplify any *leftmost* negative literals inside disjunctions with positive literals occurring on the branch. Use β-expansion only for disjunctions which contain no negative literals.

With this strategy, the emulation of a hyper tableau expansion will require only as many intermediate simplification steps as there are negative literals in the clause/disjunction in question.

There is obviously not much merit in using this emulation of hyper tableaux in an actual implementation, if problems are given as clause sets.[5] It would be simpler and more efficient to implement a rigid variable constrained formula hyper tableau calculus directly, instead of implementing non-clausal tableaux and simplification, and then restricting it to clauses. The interesting point about the emulation is that it suggests a way of generalizing hyper tableaux to non-clausal problems.

The simple unsatisfiable propositional formula $\phi = (p \wedge \neg p) \vee (q \wedge \neg q)$ demonstrates that one cannot simply forbid β-expansions of formulae if they contain a negative literal. That strategy would prohibit any expansion for ϕ, so we would lose completeness. Instead, one has to look at *disjunctive paths* (d-paths) through formulae [And81, Bib87, MR93]. These can be considered as clauses generated if a formula were to be transformed into clause normal form using the standard procedure based on distributivity. For instance, a formula $p \vee (q \wedge \neg r)$ corresponds to two clauses $p \vee q$ and $p \vee \neg r$, so the d-paths of that formula are the sequences[6] $\langle p, q \rangle$ and $\langle p, \neg r \rangle$. Although it would be possible in principle to let d-paths traverse quantifiers, we are going to simplify this presentation by treating quantified formulae as atomic entities, in keeping with our decision not to discuss simplifications below quantifiers. Note that for formulae in negation normal form, one can always shift all universal quantifiers to the top level, possibly renaming

[5] There is an advantage in cases where the original formula can be discarded, see the remark on page 88, after the proof of Theorem 6.26.

[6] We shall denote sequences by surrounding them with angle brackets $\langle \cdot \rangle$. Concatenation of sequences p, q is written as juxtaposition pq.

bound variables, and then replace them by a set of quantifier-free formulae with universal variables through a series of γ-expansions.

Definition 6.25 *The set of* disjunctive paths, *or* d-paths *of a formula ϕ, denoted* $\mathrm{dp}(\phi)$ *is defined by induction over the structure of ϕ as follows.*

- *If ϕ is a literal or a universally quantified formula, then* $\mathrm{dp}(\phi) := \{\langle\phi\rangle\}$.
- *If $\phi = \alpha_1 \wedge \alpha_2$ is a conjunction, then* $\mathrm{dp}(\phi) := \mathrm{dp}(\alpha_1) \cup \mathrm{dp}(\alpha_2)$.
- *If $\phi = \beta_1 \vee \beta_2$ is a disjunction, then* $\mathrm{dp}(\phi) := \{pq \mid p \in \mathrm{dp}(\beta_1), q \in \mathrm{dp}(\beta_2)\}$.

For the formula $\phi = (p \wedge \neg p) \vee (q \wedge \neg q)$, this definition gives:

$$\begin{aligned} \mathrm{dp}(p \wedge \neg p) &= \{\langle p\rangle, \langle\neg p\rangle\} \\ \mathrm{dp}(q \wedge \neg q) &= \{\langle q\rangle, \langle\neg q\rangle\} \\ \mathrm{dp}(\phi) &= \{\langle p, q\rangle, \langle p, \neg q\rangle, \langle\neg p, q\rangle, \langle\neg p, \neg q\rangle\} \end{aligned}$$

As d-paths correspond closely to clauses, it is not surprising that the correct generalization of our simplification strategy may be formulated like this:

> Use simplification only to simplify any *leftmost* negative literal of some d-path of a formula on the branch. Use β-expansion only for disjunctions which have at least one d-path that does not contain a negative literal.

For our formula ϕ, β-expansion will thus be applied because of the d-path $\langle p, q\rangle$. Let us call this strategy the *NNF hyper tableau strategy.*

Theorem 6.26 *The constrained variable tableau calculus with universal variables and the simp^{c2u} rule is complete if restricted according to the NNF hyper tableau strategy.*

Proof. We will not give a full completeness proof here, because it resembles previous proofs given in this chapter and elsewhere. The overall approach consists in

1. proving completeness for a corresponding (non-destructive) Smullyan-style calculus.[7]
2. lifting the completeness proof to a free variable version with universal variables.
3. showing completeness with DU constraints as in Theorem 6.13.

The first point is the one most specific to our hyper tableau strategy, so we shall have a closer look at it. We assume a Smullyan-style tableau calculus with a non-destructive simplification rule

$$\frac{\psi, \phi}{\psi[\phi], \psi, \phi}$$

[7]By a Smullyan-style tableau calculus [Smu68], we mean one without free variables: the γ rules instantiate quantifiers directly by ground terms. Therefore, all formulae occurring in the tableau are closed, and all literals are ground. See also Sect. 3.1.

Since we consider a Smullyan-style calculus, we do not need unification in this rule. The aspect of omitting ϕ can be deferred to the third step of the proof. Completeness is shown with a Hintikka-style construction. We call a set of formulae H a Hintikka set if

1. $\alpha_1 \in H$ and $\alpha_2 \in H$ for any $\alpha_1 \wedge \alpha_2 \in H$.
2. $\beta_1 \in H$ or $\beta_2 \in H$ for any $\beta = \beta_1 \vee \beta_2 \in H$, such that there is a $p \in \mathrm{dp}(\beta)$ which contains no negative literals.
3. $[x/t]\gamma_1 \in H$ for any $\forall x.\gamma_1 \in H$ and all ground terms t.
4. $\beta[L] \in H$ for any $L \in H$ such that $\neg L$ is the leftmost negative literal of some d-path of a disjunctive formula $\beta \in H$.
5. there are no complementary literals $L, \neg L \in H$, and *false* $\notin H$.

As usual, one now shows that any Hintikka set has a model. To do this one defines a model M which has the set of ground terms as domain, and which interprets any positive literal $L \in H$ as true and *every* other literal false.[8] We can now show by induction on the number of junctors and quantifiers of formulae (or any other quantity which is decreased by rule applications) that every formula in H is indeed valid in M. This is standard for α- and γ-formulae, as well as for literals. For β formulae, we have two cases.
Case 1.: If $\beta = \beta_1 \vee \beta_2$ with $p \in \mathrm{dp}(\beta)$ containing no negative literals, either $\beta_1 \in H$ or $\beta_2 \in H$, due to point 2. Thus, by the induction hypothesis, one of the disjuncts is valid in M, and so β also is.
Case 2.: If every $p \in \mathrm{dp}(\beta)$ contains at least one negative literal, we have to consider two sub-cases:
Case 2a.: There is some $L \in H$ such that $\neg L$ is the leftmost negative literal of one of the d-paths of β. Then β is simplifiable with L, and the resulting formula $\beta[L]$ is in H due to point 4. Furthermore, the induction hypothesis ensures that both L and $\beta[L]$ are valid in M, from which validity of β follows.
Case 2b.: There is no literal $L \in H$ for any of the leftmost negative literals $\neg L$ of any d-path of β. Our definition of the interpretation ensures that each of the leftmost negative literals is valid in M. Every d-path of β thus contains at least one valid literal, from which one can derive validity of β by a simple induction on the structure of β.

It now remains to show that every open branch of an infinite tableau constructed by fair application of all rules is a Hintikka set, which is done as for standard tableaux. □

Sometimes, a simplification step permits to discard the original formula ψ. In such cases, a prover using the NNF hyper tableaux strategy has an advantage over usual clausal hyper tableaux, even if the problem is given in clausal form: it can simplify the clause set while proof search is under way. Essentially, unit resolution between a universal branch literal and a clause is performed. For instance, given a literal $[X]p(X)$ and a universal disjunction $[Y]\neg pY \vee rY$, the latter can be destructively simplified to

[8]This is different from the standard proof, where the evaluation of literals not in H is irrelevant, but common for clausal hyper-tableaux-like calculi, see e.g. [Häh01].

$[Y]rY$ for that branch. This can not be done in normal hyper tableaux, as these do not keep separate clause sets per branch. Note that these separate clause sets do not imply higher memory consumption, because the representation of clauses can easily be shared between branches in an implementation.

The NNF hyper tableau strategy was implemented in the prover PrInS. Tables 6.1 and 6.2 list some results comparing the prover with that strategy (right columns) to the version without simplification, but including pruning. The effectiveness of hyper-tableaux has of course been asserted earlier, e.g. in [BFN96a] and [Küh97]. Our results clear confirm this. We will point out some particular results in the remainder of this section.

With the given strategy, PrInS is able to solve the *Steamroller* problem in the full first order formalization PUZ031+1 in about 50 ms.[9] This used to be considered a hard problem for a long time, although today, no state-of-the-art theorem prover has difficulties with it. In particular, hyper tableaux are a good way of quickly finding a proof. The interesting aspect of PrInS solving PUZ031+1 is that it does *not* use CNF transformation. To our knowledge, PrInS is the first *non clausal* theorem prover to have solved the Steamroller problem.

The problem SYN067+1, also known as Pelletier problem 38, is an example for the advantage of not needing a clause normal form. The full first order formalization SYN067+1 has a rating of 0.33 in version 2.6.0 of the TPTP library, meaning that one third of the provers considered state-of-the-art are *not* able to solve it. The reason for this is that the clause normal form for this problem, if computed in the standard way, consists of more than 80 clauses of length up to 9. The full first order version in SYN036+2 is an equivalence of two formulae with nested quantifiers, and is not very large. The NNF PrInS works on is of course somewhat larger, because $p \leftrightarrow q$ has to be translated to $(p \vee \neg q) \wedge (\neg p \vee q)$. But the NNF still helps in keeping large parts of the formula nested below the top level operators which are handled first. With the NNF hyper tableaux strategy, PrInS solves SYN067+1 in about 35 ms on average. The prover performs 64 α, β and γ expansions and 71 simplification steps. Both the NNF hyper tableau strategy *and* incremental closure help in this case: the simple version of PrInS described in Sect. 4.6 needs 1330 rule applications and about 0.23 seconds for SYN067+1, while the lean$T\!^A\!P$ clone (without simplification) cannot solve the problem within 5 minutes.

The problems SYN548+1 and SYN550+1, which are rated 0.67 and 0.33, respectively, are translations of the modal formulae

$$\Diamond\Box(\Box(p \vee \Box q) \Leftrightarrow \Box p \vee \Box q)$$

and

$$\Diamond\Box p \Leftrightarrow \Diamond\Box\Diamond\Box p$$

in the logic S4 into first order logic. The translation used transforms a box formula into $\forall y.(R(x,y) \rightarrow \ldots)$ and a diamond to $\exists y.R(x,y) \wedge \ldots)$, where the predicate R represents

[9]The machine and environment are as described in Sect. 4.6.

	without simplification			with simplification		
problem	time [s]	expand	unify	time [s]	expand	unify
GRA001−1	0.041	206	1283	0.002	48	15
GRP001−4	Out of Memory			Time Out		
GRP010−4	Out of Memory			Time Out		
GRP011−4	Out of Memory			Time Out		
LCL039−1	Out of Memory			Time Out		
LCL076−1	Out of Memory			Time Out		
LCL077−1	Out of Memory			Time Out		
LCL078−1	Out of Memory			Time Out		
LCL181+1	0.003	16	23	0.002	8	3
LCL230+1	0.000	9	14	0.001	5	3
MSC007−1.008	Time Out			108.304	308160	109591
MSC007−2.005	Time Out			1.867	11677	26455
PUZ001+1	Out of Memory			0.185	1687	358
PUZ031+1	Time Out			0.053	277	544
SET043+1	0.001	5	7	0.001	5	2
SET044+1	0.002	16	28	0.003	16	10
SET045+1	0.000	11	16	0.001	10	4
SET046+1	0.001	54	144	0.001	10	4
SET047+1	0.048	664	3995	0.015	52	51
SYN001+1	0.000	4	7	0.001	4	1
SYN036+2	0.353	2581	14662	0.152	810	417
SYN040+1	0.048	16	23	0.334	8	7
SYN041+1	0.000	3	2	0.001	3	2
SYN044+1	0.001	19	70	0.003	11	5
SYN045+1	0.004	40	70	0.006	14	4
SYN046+1	0.002	16	23	0.001	8	7
SYN047+1	0.008	108	180	0.008	30	21
SYN048+1	0.012	4	2	0.030	2	1
SYN049+1	0.002	13	34	0.002	4	1
SYN050+1	0.003	20	16	0.002	9	1
SYN051+1	0.008	24	66	0.001	7	3
SYN052+1	0.002	14	25	0.004	11	3
SYN053+1	0.002	23	32	0.005	12	5
SYN054+1	0.012	88	331	0.003	12	5
SYN055+1	0.001	9	19	0.001	7	1
SYN056+1	0.067	371	1764	0.009	41	19
SYN057+1	0.005	43	208	0.002	14	5

Table 6.1: Effect of Simplification for some TPTP problems

	without simplification			with simplification		
problem	time [s]	expand	unify	time [s]	expand	unify
SYN058+1	0.003	20	46	0.002	11	3
SYN059+1	0.012	64	309	0.007	41	17
SYN060+1	0.000	11	17	0.018	6	3
SYN061+1	0.002	9	26	0.001	8	3
SYN062+1	0.001	31	66	0.001	10	1
SYN063+1	0.003	91	180	0.012	39	42
SYN064+1	0.000	6	2	0.003	2	1
SYN065+1	0.004	10	15	0.002	9	1
SYN066+1	0.001	22	26	0.002	12	5
SYN067+1	0.062	838	5601	0.035	135	98
SYN068+1	0.000	13	30	0.001	10	2
SYN069+1	0.001	31	86	0.003	23	4
SYN070+1	0.001	57	169	0.007	42	17
SYN071+1	0.332	471	8803	0.011	57	39
SYN072+1	Out of Memory			0.195	1294	576
SYN073+1	0.065	6	3	0.001	5	3
SYN074+1	6.143	1570	50197	0.021	160	29
SYN075+1	5.217	1613	46193	0.026	214	41
SYN076+1	Out of Memory			Time Out		
SYN077+1	Out of Memory			0.055	243	652
SYN078+1	Out of Memory			0.004	88	21
SYN079+1	0.000	5	13	0.001	4	3
SYN080+1	0.008	66	419	0.001	12	4
SYN081+1	0.005	18	40	0.001	8	5
SYN082+1	0.009	13	26	0.001	13	6
SYN083+1	3.264	136	29514	0.005	29	12
SYN084+1	0.005	233	924	0.002	46	54
SYN387+1	0.000	1	1	0.000	1	1
SYN388+1	0.001	1	1	0.001	1	1
SYN389+1	0.000	3	5	0.001	3	3
SYN390+1	0.001	4	7	0.003	4	1
SYN391+1	0.001	11	27	0.020	9	3
SYN392+1	0.003	38	83	0.005	18	15
SYN393+1	0.020	192	598	0.038	106	47
SYN416+1	0.000	3	2	0.000	3	2
SYN548+1	Out of Memory			0.112	559	329
SYN550+1	Out of Memory			0.031	135	35

Table 6.2: Effect of Simplification for some TPTP problems (cont.)

the reachability relation. This is a very good situation for the NNF hyper tableau strategy, because in the disjunction coming from the universal formula, the R literal will be the left-most negative literal on any disjunctive path. Thus, the formula will not be extended further until the R literal can be simplified away. This effect disappears if the input formula is transformed into clause form first.

Prominent among the problems *not* solved by the prover with simplification in Table 6.1 are the group theory statements GRPxyz, which would require equality handling. This is covered in Chapter 7. The LCL problems that cannot be solved express derivability of certain formulae in Hilbert calculi. The problem for hyper tableaux is that the clauses of the problems are interpreted as hyper tableau clauses that blindly enumerate theorems in using the axioms and modus ponens rule of the Hilbert Calculus, hoping to come across the right conclusion by chance.

6.8 Implementation of Regularity

We already mentioned the regularity restriction in Sect. 5.8. For ground clause tableaux, this restriction simply forbids expansions which lead to multiple occurrences of the same literal on a branch. We pointed out that it is not easy to define a useful regularity restriction for non-clausal first order tableaux. Specifically, the problem with non-clausal tableaux is that some expansions are needed for completeness although they produce multiple occurrences of the same formula on a branch. The problem with the first order case is that two literals might be equal under some instantiation of the rigid variables, and not under another, and that the instantiation is not yet known when the tableau expansion takes place.

Massacci [Mas97, Mas98], points out that eager application of his simplification rule for propositional logic entails a non-clausal version of regularity. In this section, we shall show that our simplification rules permit to lift this result to the first-order case.

We shall first demonstrate how simplification can be used to *emulate* a regularity check for propositional clauses. To do this, we have to put the clause set into the initial goal of the tableau again, instead of keeping it separate as is done for clausal tableaux.

Example 6.27 We start with the goal

$$\{\neg A \vee B,\ A \vee B,\ A \vee \neg B,\ \neg A \vee \neg B\} \quad .$$

β-expansion on $\neg A \vee B$ gives two new goals

$$\begin{aligned} G_1 &= \{\neg A,\ A \vee B,\ A \vee \neg B,\ \neg A \vee \neg B\} \quad , \\ G_2 &= \{B,\ A \vee B,\ A \vee \neg B,\ \neg A \vee \neg B\} \quad . \end{aligned}$$

The regularity restriction now prohibits expanding G_1 with $\neg A \vee \neg B$, and G_2 with $A \vee B$, as this would lead to goals with two occurrences of the literals $\neg A$ and B respectively. But these disjunctions can be simplified using the literals:

$$\begin{aligned} (\neg A \vee \neg B)[\neg A] &= \mathit{eval}^{\vee}(\mathit{true}, \neg B) &&= \mathit{true} \quad , \\ (A \vee B)[B] &= \mathit{eval}^{\vee}(A, \mathit{true}) &&= \mathit{true} \quad . \end{aligned}$$

This means that the two disjunctions can be discarded from the goals, which effectively prevents their use in an expansion. Note that the simplification rule could (and probably should) also be used to simplify the remaining two disjunctions from each goal. But that is not required by the regularity restriction.

It is now clear how we will get regularity for the non-clausal case: we will simply require formulae to be simplified with respect to the literals present on the branch before an expansion, at least in those cases where the simplification step is 'positive' in the sense that some subformula gets simplified to *true*. In a sense, this is the opposite of the NNF hyper tableau strategy of the previous section, where we simplified negative occurrences of literals using positive literals. We illustrate this technique with another example.

Example 6.28 Instead of the goal G_1 in the previous example, consider the equivalent NNF goal

$$\{\neg A,\ (A \vee B) \wedge ((A \wedge \neg A) \vee \neg B)\} \quad .$$

We can immediately simplify the occurrence of $\neg A$ in the complex formula with the literal, giving

$$((A \vee B) \wedge ((A \wedge \neg A) \vee \neg B))[\neg A] = (A \vee B) \wedge (A \vee \neg B) \quad .$$

No subsequent expansion of the simplified formula can now lead to a duplication of the literal $\neg A$.

This, in a nutshell, is the way regularity is subsumed by simplification in the propositional case, as described in [Mas97, Mas98]. After reading the previous section, it should come as no surprise that regularity can be enforced for first order tableaux using our first order simplification rules with DU constraints. To do this, we adopt the following simplification strategy:

> Any possible 'positive' simplification step, that is any step in which a subformula gets simplified to *true* gets applied on a β-formula before it is expanded. Also, any simplification step possible on a literal is applied before a literal is used to simplify another formula.

The main difference from the propositional case is that we do not require eager application of simplification steps on conjunctions. The reason for this is that such a strategy can lead to cycles, similar to those for the hyper tableau strategy.

Example 6.29 Take a goal with the formulae

$$\{pa,\ [X]pX \wedge pfX\} \quad .$$

The left side of the conjunction can be simplified to true under $X \equiv a$, which gives a new literal $[X']pfX' \ll X' \equiv a$, which is equivalent to pfa. Remember that universal variables are—at least conceptually—renamed. This literal can in turn be used to simplify the left side of the conjunction, yielding $pffa$, etc.

Luckily, we can do without such simplification steps. We simply apply α-expansion leading to a goal with the three literals

$$\{pa,\ [X]pX,\ [Y]pfY\} \quad ,$$

which permit various simplifications. The second literal may be used, for instance, to simplify away the first and third ones.

For disjunctions of literals, positive simplification steps have the effect of adding DU constraints to the disjunction. The 'simplified' formula $\psi[\phi]$ is simply evaluated to *true*, so it can be discarded immediately. When the β formula is finally expanded, the literals on the new branches inherit the DU constraints, which ensure that they are not used in a branch closure that requires an instantiation that would render them identical to a literal already on the branch. In a sense, we use constrained formulae to achieve the effect for which a global constraint is used in [LSBB92], see Sect. 5.8.

Example 6.30 Let the following goal be given:

$$\{pa,\ \neg qb,\ pX \vee \neg qX\} \quad .$$

Before β expansion is permitted, the β formula has to be simplified with the two literals. While the simplified formula $\psi[\phi]$ is *true* (with some constraint) in both cases, the simplification steps add the DU constraint $!(X \equiv a)\ \&\ !(X \equiv b)$ to the disjunction. The β-expansion then gives two branches with goals

$$\begin{array}{l} \{pa,\ \neg qb,\ pX \ll\ !(X \equiv a)\ \&\ !(X \equiv b)\} \quad , \\ \{pa,\ \neg qb,\ \neg qX \ll\ !(X \equiv a)\ \&\ !(X \equiv b)\} \quad . \end{array}$$

The constraints do not prevent X from being instantiated with a or b, but they prevent the new literals to be used for branch closure in that case, and they prevent further proof steps involving these literals which would require such an instantiation.

The question arises whether it is effective enough to add DU constraints to formulae. After all, in the previous example, the β expansion was executed anyway, there will be no backtracking step to remove it later, and the variable X might still be instantiated to a or b. The idea is that this method should be used together with the pruning technique as described in Sect. 5.6. Then, if the left subtree can be closed under an instantiation with $\sigma(X) = a$ for instance, this closure can not depend on the constrained literal $pX \ll \cdots$ introduced by this β-expansion, so the Merger will simply pass the closing instantiation up the tree without requiring closure of the right subtree.

6.9 Development of Refinements

In the previous sections, the introduction of simplification rules with DU constraints and universal variables was motivated by pointing out redundancies arising if these refinements are not used. It might be interesting to note that the author was prompted to introduce these refinements in the prover PrInS through *observations* of the prover.

An immediate consequence of using a non-backtracking proof procedure is that the prover can be halted in the middle of an unsuccessful proof search, and that the current state of the proof tree will reflect the work that has been done so far. In a backtracking prover, most of this information has been thrown away in backtracking steps.

Given an incomplete proof tree, one can then take one of the open branches and try to analyse the formulae in the leaf goal. The redundancies mentioned above were detected that way.

Surely, this approach is not viable if redundant rule applications only appear when a goal already contains hundreds of formulae. But for some problems, a combinatorial explosion will take place early enough, so that one only needs to look at a few dozen formulae to discover it.

6.10 Related Work

A mechanism similar to the simplification rules presented in this chapter has independently been developed by Peltier [Pel97, Pel99]. The idea of using formulae on a branch to simplify other formulae is the same as for the simplification rules of Massacci [Mas97, Mas98], and the ones presented here. The problem of dealing with the instantiation of rigid variables is solved differently however. While we use ordinary first order formulae and attach a syntactic constraint to them, Peltier intertwines constraints and formulae. For instance, in a formula

$$\forall x.\forall y.(x = y \lor p(x,y)) \quad ,$$

$x = y$ plays the role of a constraint, which makes the formula $p(x,y)$ available only if x and y are instantiated by different ground terms. The symbol $=$ denotes syntactic equality, and not an equality predicate like $\doteq$ in the following chapter. On the other hand, quantification over the variables x, y used in the syntactic equality is possible. This means that the semantics of such mixtures of formulae and constraints can only be defined with respect to a Herbrand model, that is a model in which the elements of the carrier set are ground terms. The possibility of attaching different constraints to different parts of a larger formula might be an advantage of Peltier's approach, but we have not investigated this. Keeping formulae and constraints apart, as we do, and differentiating between variables bound in quantifiers, and free variables which may occur in constraints certainly makes the calculus easier to understand, and easier to reason about.

6.11 Summary

We have used the technique of attaching syntactic unification constraints to the formulae in a tableau to deal with rules which are possible only under certain instantiations of rigid variables in a proof confluent framework. This technique was used to define a number of simplification rules. We showed how these rules can be used to define non-clausal first order versions of hyper tableaux and regularity. We also discussed the use of syntactic dis-unification constraints to simulate destructive rules in a proof confluent way.

In the next chapter, we are going to use constrained formula tableaux to add equality handling to our framework.

7 Equality Handling

The previous chapters dealt with first order predicate logic without equality. It is not strictly necessary to include a special treatment for the equality predicate in the definition of the semantics of a logic, nor is it theoretically necessary to employ any special treatment of equality in a calculus. Given a set of formulae in first order logic with equality, one can add an axiomatization of equality to the problem, so that the combined set of formulae is unsatisfiable with respect to first order logic *without* equality, if and only if the original set was unsatisfiable in the logic *with* equality.

This approach is completely unpractical however. It is well known that only very small problems can be solved with the axiomatic approach. For larger problems, the equality axioms introduce so much redundancy that none of the optimizations described in the previous chapters could cope with it. This holds for resolution provers as well as tableau provers, backtracking or not.

There are basically two ways to improve on the axiomatic approach. One way is to *transform* problems into a form which makes the equality axioms superfluous. The first such transformation was proposed by Brand [Bra75], more recent developments are described in [MS97, BGV97]. The other way to treat equality is to actually build it into the calculus and the proof procedure. This is the approach we are going to investigate here.

The simplest way of building in equality is with *paramodulation* rules, that is rules based on the replacement of equals, as one would do in a mathematical proof. Given a formula $\phi(s)$ containing an occurrence of a term s, and an equation $s \doteq t$, we can derive $\phi(t)$, which has the occurrence of s replaced by t. An integration of paramodulation into the resolution calculus was proposed by Robinson and Wos in [RW69]. For sequent based calculi, work goes back even earlier, see e.g. [Kan63]. A simple paramodulation rule is also used to build equality into free variable tableaux in [Fit96].

Unfortunately, unrestricted paramodulation can also lead to a lot of redundant derivations, as equations have to be applied in both directions. The key to reducing this redundancy is to use *term orderings* and to require that equations only be applied in a way that makes terms and formulae smaller with respect to such an ordering. The most prominent application of term orderings for equality reasoning is of course the Knuth-Bendix Completion procedure [KB70]. A good overview of the state-of-the art in ordered paramodulation from a resolution perspective can be found in [NR01]. An overview of equality handling methods for tableaux and other sequent-based calculi is given in [DV01].

The topic of this chapter is a method for equality handling in free variable tableaux which is based on ordered paramodulation. Our rules are similar in spirit to the simplification rules presented in the previous chapter, and like those they are particularly well suited for the incremental closure approach, but they could also be applied in a backtracking prover. In other words, the results presented in this chapter are useful independently of the incremental closure technique.

7.1 Ordering-Based Equality Handling in Tableaux

Efficient equality handling for first order tableaux or related calculi, like matings or the connection method, has been problematic for a long time. It is generally believed that only techniques based on ordered rewriting can sufficiently reduce the search space of equality reasoning to make it tractable. It was also believed that the best approach to the integration of free variable tableaux and equality handling would be to search for *simultaneous rigid E-unifiers* [GRS87] of disequations on the tableau and use these to close branches instead of usual unifiers. So the overall idea was to solve the rigid E-unification problems using ordered rewriting techniques.

Example 7.1 We construct a tableau from the following formula:[1]

$$\begin{aligned}\forall x,y,u,z.(\quad & (a \doteq b \land \neg g(x,u,v) \doteq g(y,fc,fd)) \\ \lor \; & (c \doteq d \land \neg g(u,x,y) \doteq g(v,fa,fb)) \;)\end{aligned}$$

After applying four γ-, one β- and two α-expansions, we get the following two goals (leaving out the γ formula):

$$\begin{gathered}\{a \doteq b,\ \neg g(X,U,V) \doteq g(Y,fc,fd)\} \\ \{c \doteq d,\ \neg g(U,X,Y) \doteq g(V,fa,fb)\}\end{gathered}$$

Closing *one* of these goals constitutes what is known as a *rigid E-unification* problem. The task is to find instantiations for the rigid variables X, Y, U, V, such that the two sides of the negated equation are equal with respect to the equational theory $E = \{a \doteq b\}$, resp. $E = \{c \doteq d\}$, defined by the positive equational literals in the goal. In general these equations might also contain free variables, and of course, all occurrences of these variables must be instantiated to the same terms.

One gets a *simultaneous rigid E-unification* problem, if one tries to solve several rigid E-unification problems with a *single* instantiation. For instance, the simultaneous rigid E-unification problem produced by the two goals above has the following solution:

$$[X/fa,\ Y/fb,\ U/fc,\ V/fd] \quad .$$

The inequation in the first goal becomes

$$\neg g(fa,fc,fd) \doteq g(fb,fc,fd)$$

under that instantiation, so equality of the two sides does indeed follow from the equation $a \doteq b$, and similarly for the second goal.

[1]This example is taken from [DV97]. We use $\doteq$ to denote the equality predicate.

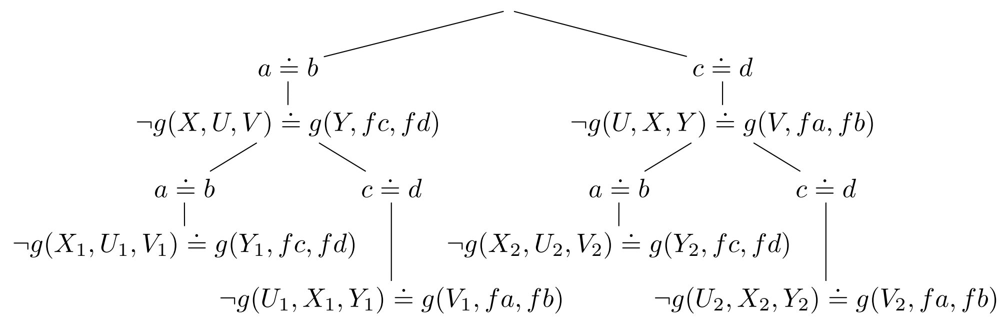

$a \doteq b$
$c \doteq d$
$\neg g(X,U,V) \doteq g(Y,fc,fd)$
$\neg g(U,X,Y) \doteq g(V,fa,fb)$
$a \doteq b$
$c \doteq d$
$a \doteq b$
$c \doteq d$
$\neg g(X_1,U_1,V_1) \doteq g(Y_1,fc,fd)$
$\neg g(X_2,U_2,V_2) \doteq g(Y_2,fc,fd)$
$\neg g(U_1,X_1,Y_1) \doteq g(V_1,fa,fb)$
$\neg g(U_2,X_2,Y_2) \doteq g(V_2,fa,fb)$

Figure 7.1: A tableau with equality.

Unfortunately, in 1996, simultaneous rigid E-unification was shown to be undecidable [DV96], unexpectedly invalidating a number of attempts at a completeness proof that were based on the opposite assumption. The outlined plan could thus only be implemented using incomplete procedures for E-unification. In particular, procedures were used, which produce complete sets of solutions to rigid E-unification problems for each branch, and try to join these solutions to obtain a solution for the simultaneous problem, see e.g. [Bec94]. Experimenting with such a setting, it turned out that the *combination* of a first order theorem prover and an incomplete solver for rigid E-unification problems seemed to be complete despite the incompleteness of the unification machinery, though nobody knew exactly why. Even if the needed simultaneous unifier was not found at a given point in the proof construction, some simultaneous unifier would always be found after the tableau was expanded a little further [Bec93, Bec94, Pet94].

We shall try to give an intuition of this effect without actually attempting to present a procedure that solves rigid E-unification problems. We do assume the reader to be familiar with rewriting techniques in general however.

Example 7.2 For the two goals of the previous example, an E-unification procedure based on ordered rewriting would fail to find a simultaneous unifier. For the first goal, it would find a unifier

$$[X/Y,\ U/fc,\ V/fd] \quad ,$$

but that will not lead to a closure of the second goal. There is no incentive for instantiation of X and Y to fa and fb, if the other goal is not taken into account, and it is desirable to handle goals as independently as possible.

However, if the tableau is further expanded by repeating the same steps as before on each of the two goals, the situation changes. If we omit the γ formulae, we now get a tableau with the four goals

$$\begin{array}{llll}
\{a \doteq b, & & \neg g(X,U,V) \doteq g(Y,fc,fd), & \neg g(X_1,U_1,V_1) \doteq g(Y_1,fc,fd)\} \\
\{a \doteq b, & c \doteq d, & \neg g(X,U,V) \doteq g(Y,fc,fd), & \neg g(U_1,X_1,Y_1) \doteq g(V_1,fa,fb)\} \\
\{c \doteq d, & a \doteq b, & \neg g(U,X,Y) \doteq g(V,fa,fb), & \neg g(X_2,U_2,V_2) \doteq g(Y_2,fc,fd)\} \\
\{c \doteq d, & & \neg g(U,X,Y) \doteq g(V,fa,fb), & \neg g(U_2,X_2,Y_2) \doteq g(V_2,fa,fb)\}
\end{array}$$

The structure of the tableau, showing only the literals at the nodes where they are introduced, is given in Fig. 7.1. Each goal now contains two negated equations. To close the tableau, the prover has to choose one of these from each branch, and then find a simultaneous solution for the four rigid E-unification problems. It turns out that the right choice is to take the second negated equation for the leftmost and rightmost branch, and the first for the two middle branches. Let us consider the corresponding rigid E-unification problems from left to right. First, we have

$$\{a \doteq b,\ \neg g(X_1, U_1, V_1) \doteq g(Y_1, fc, fd)\}$$

Here, the unification

$$[X_1/Y_1,\ U_1/fc,\ V_1/fd]$$

is found. In the next goal

$$\{a \doteq b,\ c \doteq d,\ \neg g(X, U, V) \doteq g(Y, fc, fd)\} \quad ,$$

the negated equation may be rewritten using the equations. Assuming a term ordering where c is larger than d, this will give

$$\neg g(X, U, V) \doteq g(Y, fd, fd) \quad .$$

The corresponding unifier is joined to the other one, giving

$$[X_1/Y_1,\ U_1/fc,\ V_1/fd,\ X/Y,\ U/fd,\ V/fd] \quad .$$

The next goal

$$\{c \doteq d,\ a \doteq b,\ \neg g(U, X, Y) \doteq g(V, fa, fb)\}$$

also allows rewriting. If we assume a to be greater than b in the ordering, we will get

$$\neg g(U, X, Y) \doteq g(V, fb, fb) \quad .$$

The unifier of these terms can also be joined to the previously collected one, giving

$$[X_1/Y_1,\ U_1/fc,\ V_1/fd,\ X/fb,\ Y/fb,\ U/fd,\ V/fd] \quad .$$

Finally, for the last goal, no rewriting is possible, and a unifier like the one for the first goal is produced. Putting all together, we have found the following simultaneous E-unifier

$$[X_1/Y_1,\ U_1/fc,\ V_1/fd,\ X/fb,\ Y/fb,\ U/fd,\ V/fd,\ U_2/V_2,\ X_2/fa,\ Y_2/fb]$$

To summarize, although there is a simultaneous E-unifier for the tableau after only one γ-expansion, a procedure using rewriting and syntactic unification independently on the goals will not find it. This procedure is thus incomplete for simultaneous rigid E-unification. After two more γ-expansions however, this same procedure finds a unifier that closes the tableau.

The procedure we applied in this example is actually part of a typical procedure to compute per-branch solutions of rigid E-unification problems. The question whether this approach leads to a complete tableau calculus for first order logic with equality remained open for some years.

In 1997, Degtyarev and Voronkov finally showed completeness for such a combination of tableaux and rigid E-unification [DV97, DV98], and thus for a tableau calculus with integrated superposition-based equality handling.

One might have expected that all problems would be solved after this discovery. A number of publications would follow, providing variations on the theme, like what is known as 'basic ordered paramodulation' in the resolution community, or a version with universal variables (see Sect. 5.7), or hyper tableaux (see Sect. 6.7) with equality. Curiously enough, this has not happened! We surmise that the reason for this is the complexity of Degtyarev and Voronkov's completeness proof: It is over ten pages long, and very technical, although the proof of one of the used theorems is not even included in those papers (Theorem A.15 in [DV97] is taken from [DV94]).

In this chapter, we present calculi similar to (a clausal version of) the one presented in [DV98], though we prefer to integrate the superposition process into the tableau calculus instead of defining a separate calculus for rigid E-unification. We then show the completeness of this calculus using an adaptation of the technique called *model generation*, well known for resolution calculi. This technique was first introduced by Bachmair and Ganzinger [BG90, BG94, BG01] to show completeness of resolution calculi with strict superposition. Nieuwenhuis and Rubio have adapted the model generation technique for resolution calculi with constraint propagation [Rub94, NR95b, NR97, NR01], which is nearer to our application. For that reason, we shall follow Nieuwenhuis and Rubio in our notation and presentation.

Apart from being significantly shorter than the proof of Degtyarev and Voronkov, our completeness proof has the advantage of requiring only few additional ingredients not known from resolution. This should make it easy to produce tableau versions of variants known for resolution, like basic ordered paramodulation (see Sect. 7.6) or hyper resolution resp. hyper tableaux. The results in this chapter were first presented in [Gie02].

7.2 Preliminaries

We need to introduce a few more concepts and notations, which are specific to equality handling. We shall also deviate from some of the techniques used in the previous chapters, in order to make our presentation simpler.

As before, we shall assume a fixed signature consisting of function symbols with fixed arity, constant symbols being considered as functions of arity zero. But we now admit only a single binary predicate symbol '$\doteq$' denoting equality. The equality symbol is handled in a symmetric way, i.e. two formulae $s \doteq t$ and $t \doteq s$ are considered identical. A literal is either an equation $s \doteq t$ or a negated equation $\neg s \doteq t$. We do not include other predicates for two reasons: first, it is well known that other predicates can be simulated with equality, by introducing constants *true* and *false* and representing a literal $p(t_1,\ldots,t_n)$ by an equation $p(t_1,\ldots,t_n) \doteq \mathit{true}$. Second, it is conceptually easy to modify our proofs to work with non-equality predicates, but doing so makes them harder to read.

Further, we shall limit our exposition to problems given in clausal form. Questions of equality handling are typically orthogonal to the treatment of the rest of first order logic, and working on clauses makes our proofs much more readable. This implies that we are going to work with clausal tableaux, in contrast to the NNF tableaux which we have been using since their introduction in Definition 4.2. The incremental closure technique remains valid for them, as branch closure works in the same way for both kinds of tableaux. We define a clause to be a finite set of (equality) literals.

We shall have to talk about models and interpretations in our completeness proofs. Contrary to the usual approach of using a carrier set and an interpretation function to evaluate terms, we shall follow [NR01] in defining an interpretation to be a congruence relation on ground terms. In other words, we take a quotient of the set of all ground terms as carrier set, and interpret terms to their congruence class. We do not need to define an interpretation for predicates, as we only have one predicate $\doteq$, which is interpreted as equality of equivalence classes. Herbrand's Theorem guarantees that for our purposes, this definition is equivalent to defining interpretations with arbitrary carrier sets. Validity of a formula ϕ in an interpretation I will be written $I \models \phi$. Interpretations will often be described by sets of rewrite rules in the way captured by the following definition.

Definition 7.3 *A* (ground) rewrite rule *is an ordered pair $l \Rightarrow r$ of ground terms. If R is a set of rewrite rules, the* interpretation induced by R *is the minimal congruence R^* on ground terms, such that lR^*r for all $l \Rightarrow r \in R$.*

The following notions are needed to formalize the application of equations on some subterm of a literal.

Definition 7.4 *A* position *is a sequence of numbers designating subterms. $s|_p$ is the subterm of s at position p, that is*

- *$s|_\lambda = s$, where λ is the empty sequence.*
- *$f(s_1,\ldots,s_n)|_{k.q} = s_k|_q$ for $1 \le k \le n$.*

$s[r]_p$ denotes the result of replacing the subterm at position p in s by r, that is

- *$s[r]_\lambda = r$, and*
- *$f(s_1,\ldots,s_n)[r]_{k.q} = f(s_1,\ldots,s_{k-1},s_k[r]_q,s_{k+1},\ldots,s_n)$ for $1 \le k \le n$.*

We also need to fix a suitable rewrite ordering. In contrast to the usual definitions, we do not need to order terms with variables.

Definition 7.5 *A* total ground reduction ordering *is a total ordering $\succ$ on the set of ground terms, which is*

- *well-founded, i.e. there is no infinite chain $t_0 \succ t_1 \succ t_2 \succ \cdots$, and*
- *monotonic, i.e. $u[s]_p \succ u[t]_p$ for all ground terms u, s, t with $s \succ t$ and positions p.*

A total ground reduction ordering $\succ$ is extended to a total well-founded ordering $\succ_l$ on ground literals as follows: A ground literal is assigned a multiset by $m(s \doteq t) := \{s, t\}$ and $m(\neg s \doteq t) := \{s, s, t, t\}$. Then $L \succ_l L'$ iff $m(L) \succ\!\!\succ m(L')$, where $\succ\!\!\succ$ is the multiset extension of $\succ$.

We shall need the following well-known fact in our proofs:

Proposition 7.6 *Any total ground reduction ordering $\succ$ enjoys the* subterm property*, that is $u \succ t$ for all proper subterms t of a ground term u.*

It will also be useful to keep the following properties of the literal ordering $\succ_l$ in mind, which follow immediately from this definition:

Proposition 7.7 *If $s \succ t$ and $s' \succ t'$, then*

$$\begin{array}{rll} s \doteq t \succ_l s' \doteq t' & \textit{iff} & s \succ s' \textit{ or } (s = s' \textit{ and } t \succ t') \\ s \doteq t \succ_l \neg s' \doteq t' & \textit{iff} & s \succ s' \\ \neg s \doteq t \succ_l s' \doteq t' & \textit{iff} & s \succeq s' \\ \neg s \doteq t \succ_l \neg s' \doteq t' & \textit{iff} & s \succ s' \textit{ or } (s = s' \textit{ and } t \succ t') \end{array}$$

We shall assume a fixed total ground reduction ordering $\succ$, and corresponding literal ordering $\succ_l$, throughout the remainder of this chapter. In our examples, we will occasionally use the lexicographic path ordering (LPO) on ground terms.

Definition 7.8 *Let $>$ be a strict total ordering on the function and constant symbols of a signature, called* precedence*. We inductively define the* lexicographic path ordering *(LPO) $\succ$ on the set of ground terms as follows. Let $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$. Then $s \succ t$, iff*

- *$s_i \succeq t$ for some $i \in \{1, \ldots, m\}$, or*
- *$f > g$ and $s \succ t_j$ for all $j \in \{1, \ldots, n\}$, or*
- *$f = g$, and there is some $j \in \{1, \ldots, n\}$ such that $s_i = t_i$ for $i < j$, $s_j \succ t_j$, and $s \succ t_i$ for $i > j$.*

It is well-known that the LPO is a total ground reduction ordering for any (strict, total) precedence.

Our equality handling rules will use constrained formulae—or rather constrained literals, as we construct clause tableaux—in a similar way as the simplification rules of Chapter 6 did. However, we need a different constraint language here. Specifically, we shall express ordering requirements as constraints.

For the purposes of this chapter, a constraint is a quantifier-free first order formula which can use two predicate symbols with fixed interpretation, namely '$\equiv$' representing (syntactic) equality and and '$\succ$' for the reduction ordering. As before, we denote conjunction as '&' in constraints. Disjunction and negation will not be needed. A substitution satisfies a constraint, if the constraint is true under the fixed interpretation when its free variables are assigned values according to the substitution. A constraint is satisfiable, if there is a substitution that satisfies it.

Example 7.9 The constraint

$$X \equiv f(Y) \,\&\, Y \succ X$$

is not satisfiable under any total ground reduction ordering, due to the subterm property. The constraint

$$X \succ a \,\&\, b \succ X$$

may or may not be satisfiable, depending on whether there is a ground term between a and b in the chosen term ordering.

A family of practical algorithms for checking the satisfiability of constraints interpreted over recursive path orderings (RPOs) is discussed in [NR99]. For Constraints interpreted over Knuth-Bendix orderings, see [KV01].

7.3 A Simple Calculus

In this section, we shall introduce the simplest version of our calculus and show its completeness. Variations of the calculus will be introduced in the following sections.

7.3.1 The Calculus

We describe a clausal free variable tableau calculus to refute sets of clauses. Let a set $\mathcal{C}$ of clauses be given. The calculus consists of three rules:

$$\textit{ext}\quad \frac{}{\theta L_1 \quad | \quad \cdots \quad | \quad \theta L_k}$$

where $C = \{L_1, \ldots, L_k\} \in \mathcal{C}$,
and θ renames each variable in C into a new (free) variable.

$$\textit{sup-p}\quad \frac{\begin{array}{c} s \doteq t \ll A \\ l \doteq r \ll B \end{array}}{s[r]_p \doteq t \ll s|_p \equiv l \,\&\, s \succ t \,\&\, l \succ r \,\&\, A \,\&\, B}$$

where p is a position in s and $s|_p$ is not a variable.

$$\textit{sup-n}\quad \frac{\begin{array}{c} \neg s \doteq t \ll A \\ l \doteq r \ll B \end{array}}{\neg s[r]_p \doteq t \ll s|_p \equiv l \,\&\, s \succ t \,\&\, l \succ r \,\&\, A \,\&\, B}$$

where p is a position in s and $s|_p$ is not a variable.

The superposition rules *sup-p* and *sup-n* are only applied if the constraint of the new literal is satisfiable. The two literals involved as premises in the *sup-p*-rule are required to be distinct,[2] although one might be a renaming of the other.

Note that constraints are attached to the formulae on a branch, and that these constraints are propagated by the rules, like in the simplification rules of the previous chapter. When we don't write a constraint (as in the *ext*-rule) we mean the empty constraint that is satisfied by any substitution.

All rules are non-destructive, i.e. the original formulae stay in the expanded goals. They were not included in the rules given above to save space. The rules are used to expand a tableau, similarly to Def. 4.2.

Branches are now closed by unifying the two sides of a negated equation: An instantiation $\sigma \in \mathcal{I}$ closes a goal G of a tableau, if there is a constrained negated equation $(\neg s \doteq t \ll A) \in G$ such that $\sigma s = \sigma t$ (that is syntactic identity) and σ satisfies A. The whole tableau is closed, if there is a single substitution σ that closes all leaf goals of the tableau simultaneously.

The *sup*-rules implement what is known as *rigid basic superposition*. The term 'rigid' refers to the rigidity of the free variables of our tableau calculus. One talks of superposition when only ordered application of equations is allowed, and *only on the maximal side* of an equation, which in our case is enforced by the constraint $s \succ t$. Finally the *basicness restriction* forbids application of equations on subterms created by unifiers introduced by previous superposition steps. In our case, this is achieved by deriving a literal $s[r]_p \doteq t \ll s|_p \equiv l \ldots$ instead of determining a most general unifier μ of $s|_p$ and l and generating a literal $\mu(s[r]_p \doteq t)$, as would be done in a calculus without constraints.

In an incremental closure implementation, closure of a tableau is determined as usual, except that instead of propagating constraints corresponding to the unification of pairs of complementary literals, we now propagate constraints expressing the unification of two sides of a negated equation. Mergers are introduced for applications of the *ext*-rule. *sup*-applications do not need to be reflected in the Sink structure. Obviously, as clauses are not necessarily of length 2, one needs to use the technique of Sect. 5.9 for n-ary branching. One should also consider keeping the unification constraints and the ordering constraints separate, as unification constraints are relevant for soundness, while ordering constraints are not. Before δ-propagation, one can then discard the ordering part, as was suggested for the dis-unification part of constraints in Sect. 6.5.2. The advantage of this approach is twofold: One gets less restrictive constraints, which might allow closing the proof earlier. And the constraint is simplified by leaving out orderings, reducing the burden for the closure test.

Example 7.10 We give a simple example to show how the calculus works. We will

[2]This restriction cannot be imposed in calculi dealing with universal equations, like the original Knuth-Bendix completion, unfailing Knuth-Bendix completion, or resolution saturation procedures. We can require the literals to be distinct, because we have *rigid* variables. With rigid variables, a term can't be unified with one of its proper subterms, so superposition would only be possible at the top position, leading to a trivial equation.

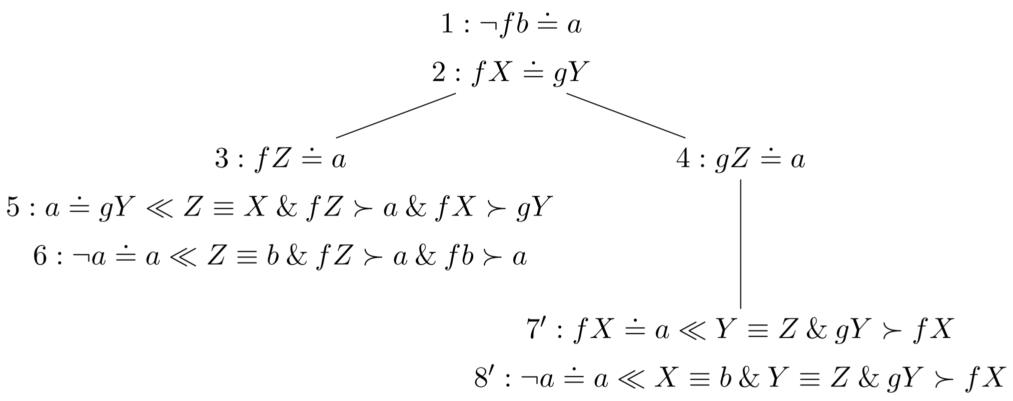

$1 : \neg fb \doteq a$
$2 : fX \doteq gY$
$3 : fZ \doteq a$
$4 : gZ \doteq a$
$5 : a \doteq gY \ll Z \equiv X \,\&\, fZ \succ a \,\&\, fX \succ gY$
$6 : \neg a \doteq a \ll Z \equiv b \,\&\, fZ \succ a \,\&\, fb \succ a$
$7' : fX \doteq a \ll Y \equiv Z \,\&\, gY \succ fX$
$8' : \neg a \doteq a \ll X \equiv b \,\&\, Y \equiv Z \,\&\, gY \succ fX$

Figure 7.2: A tableau using the rigid basic superposition rules.

show that the following set of clauses is unsatisfiable.

$$\begin{gathered} \neg fb \doteq a \\ fx \doteq gy \\ fz \doteq a \lor gz \doteq a \end{gathered}$$

The finished tableau is given in Fig. 7.2, but we shall explain how it is constructed, step by step. As term ordering, we choose a lexicographic path ordering (see Def. 7.8) with precedence $g > f > b > a$. We can start by expanding with the first two clauses, which gives us a goal with the two literals

$$\begin{gathered} 1 : \neg fb \doteq a \\ 2 : fX \doteq gY \end{gathered}$$

where X and Y are free variables. Between these, a *sup-n* application is possible by overlapping the left sides. The resulting literal is

$$\neg gY \doteq a \ll X \equiv b \,\&\, fb \succ a \,\&\, fX \succ gY \quad .$$

The constraint of this literal is unsatisfiable, because it requires X to be instantiated with b, but every term gY is larger than fb under the LPO with the chosen precedence. Accordingly, this rule application is not allowed. Instead, we need to further expand the tableau, using the third clause. We now get two branches, with

$$3 : fZ \doteq a$$

on the left branch, and

$$4 : gZ \doteq a$$

on the right one. On the left branch, a superposition step of 3 on 2 leads to the literal

$$5 : a \doteq gY \ll Z \equiv X \,\&\, fZ \succ a \,\&\, fX \succ gY \quad .$$

This time, the constraint is satisfiable, for instance by instantiating X and Z to ga, and Y to a. But there is no further inference between literal 5 and the other literals on this branch. In particular, superposition between the right sides of 5 and 2 is not possible, because of the ordering constraint $fX \succ gY$ of 5. Superposition of 2 on 3 would lead to the same literal 5. But superposition of 3 on 1 produces

$$6 : \neg a \doteq a \ll Z \equiv b \,\&\, fZ \succ a \,\&\, fb \succ a \quad ,$$

the constraint of which is satisfied whenever Z is instantiated to b. As this is a negated equation between two identical terms, this goal is closed for $\sigma(Z) = b$. No further superposition steps can be performed without expanding the left goal with further clause copies, so we now turn to the right goal, which currently contains the following three literals:

$$1 : \neg fb \doteq a$$
$$2 : fX \doteq gY$$
$$4 : gZ \doteq a$$

These allow only one superposition step, namely between 4 and 2, producing the literal

$$7 : fX \doteq a \ll Y \equiv Z \,\&\, gY \succ fX \,\&\, gZ \succ a \quad ,$$

the constraint of which is easily seen to be satisfiable. As gZ is always larger than a under the given ordering, we can slightly simplify the constraint:

$$7' : fX \doteq a \ll Y \equiv Z \,\&\, gY \succ fX \quad .$$

Literal $7'$ permits one further superposition step on 1, which gives us

$$8 : \neg a \doteq a \ll X \equiv b \,\&\, fb \succ a \,\&\, fX \succ a \,\&\, Y \equiv Z \,\&\, gY \succ fX \quad ,$$

where the constraint may again be simplified by removing the trivially satisfied conjuncts, to

$$8' : \neg a \doteq a \ll X \equiv b \,\&\, Y \equiv Z \,\&\, gY \succ fX \quad .$$

No other superposition steps are possible on this goal. As any term gY is larger than fb in our term ordering, the goal is now closed under any σ with $\sigma(X) = b$ and $\sigma(Y) = \sigma(Z)$. This allows us to close the tableau with

$$[X/b, Y/b, Z/b] \quad .$$

Thanks to the ordering constraints, only few superposition steps were possible, which led to a very small search space.

The *sup-n* and *sup-p* rules can also be used in a backtracking prover. We discussed that possibility for the simplification rules in Sect. 6.4.

- One gets a backtracking calculus similar to that of Degtyarev and Voronkov, if one does not keep the constraints together with the literals, but instead gathers them all in one global constraint G that is required to be satisfiable. This introduces a backtracking choice point for each rule application that adds to the global constraint. In addition branch closure requires backtracking, as usual in free variable tableaux: whenever a negated equation $\neg s \doteq t$ appears on a branch, a backtracking point is introduced and the constraint $s \equiv t$ is added to G. The procedure tries to close the other branches, always keeping G satisfiable, and keeping below a certain instantiation depth limit. If this fails, extension of the branch is continued. If no proof is found up to a given depth limit, the whole procedure is restarted with an increased limit (iterative deepening). In contrast to the classic formulation of tableaux, the unifiers generated in superposition applications and branch closures should *not* be applied to the tableau, as this would yield possibilities for new, spurious rule applications on the other branches, weakening the 'basicness' property. Of course, rule applications on other branches that generate constraints incompatible with the global constraint G need not be considered in this scheme.

  The essential difference to the calculus of Degtyarev and Voronkov is that they discard the first premise in a *sup*-rule application. Of course, actually discarding a literal is possible only in a backtracking calculus. In Sect. 7.4, we shall investigate a way of simulating this destruction in a proof confluent way.

- One can avoid the backtracking points introduced by the *sup*-rules by keeping the constraints of literals. These are only added to the global closure constraint G when a branch is closed, and accordingly backtracking is only needed over branch closures.

We emphasize the applicability of our results in a backtracking context for two reasons: First, to make it clear that our results may be applied in a broader context than just with an incremental closure prover. And second, to convince the reader that our results do indeed subsume those of Degtyarev and Voronkov [DV97], in that our calculus (at least in the version presented in Sect. 7.4) has all the properties they show of theirs.

7.3.2 Completeness

The completeness proof follows the usual lines: Assuming that there is no closed tableau for a set of clauses, one constructs an infinite tableau by applying rules exhaustively—in particular, the *ext*-rule has to be applied infinitely often for each clause on each branch. Then one chooses a ground instantiation σ for the free variables, such that after applying the substitution to the tableau, every branch contains at least one literal from every ground instance of each of the clauses. From the assumption, it follows that at least one branch $\mathcal{B}$ of the tableau is not closed by σ. From the literals on $\sigma\mathcal{B}$, an interpretation is constructed, which is then shown to be a model for the clause set.

Our proof differs from this standard approach only in the construction of the interpretation and in the proof that the clause set is indeed satisfied by it.

First, we need the following notion:

Definition 7.11 *Given a set $\mathcal{B}$ of constrained literals, a ground instantiation σ for all free variables occurring in $\mathcal{B}$, and a set R of ground rewrite rules, the set of* variable-irreducible ground instances of $\mathcal{B}$ under σ with respect to R, *written $irred_R(\sigma, \mathcal{B})$, is the set of all ground literals $(\neg)\sigma l \doteq \sigma r$, where $((\neg)l \doteq r \ll A) \in \mathcal{B}$, A is satisfied by σ, and σx is irreducible by R for all variables x occurring in l or r.*

Note that irreducibility is not required for the whole terms σl and σr, but only for the instantiations of variables occurring in them. Also, the instantiation of variables occurring only in the constraint A is allowed to be reducible. We are going to work only on variable-irreducible ground instances of the constrained literals on a branch. The reason for this will become clear later.

We can now define the 'model generation' process, which constructs a ground rewrite system by induction with $\succ_l$ over variable-irreducible ground instances of literals on a branch. The tricky part here is that the rewrite relation that variable-irreducibility refers to is only just being built during the induction.

Definition 7.12 *Let $\mathcal{B}$ be a set of constrained literals and σ a ground substitution on all variables in $\mathcal{B}$. For any ground literal L, we define* $\mathrm{Gen}(L) = \{l \Rightarrow r\}$ *and say L* generates the rule $l \Rightarrow r$, *iff*

1. *$L \in irred_{R_L}(\sigma, \mathcal{B})$,*
2. *$L = (l \doteq r)$,*
3. *$R_L^* \not\models L$,*
4. *$l \succ r$, and*
5. *l is irreducible w.r.t. R_L,*

where $R_L := \bigcup_{L \succ_l K} \mathrm{Gen}(K)$ is the set of all previously generated rules. Otherwise, we define $\mathrm{Gen}(L) := \emptyset$. *The set of all rules generated by any ground literal is denoted $R_{\mathcal{B},\sigma} := \bigcup_K \mathrm{Gen}(K)$.*

Note that only positive equations generate rules. When no confusion is likely concerning the set $\mathcal{B}$ and the substitution σ, we will just write R instead of $R_{\mathcal{B},\sigma}$.

We have the following two useful lemmas, which are slightly reformulated versions of Lemma 3.2 of [NR01]:

Lemma 7.13 *For any set of constrained literals $\mathcal{B}$ and ground substitution σ, the generated set of rules $R = R_{\mathcal{B},\sigma}$ is convergent, i.e. confluent and terminating. The subset R_L is also convergent for any ground literal L.*

Proof. R terminates because $l \succ r$ for all rules $l \Rightarrow r \in R$ (condition 4). To show confluence, by Newman's Lemma, one thus only needs to show local confluence, which follows from the fact that there can be no critical pairs in R. For assume $l \Rightarrow r \in R$ and $l' \Rightarrow r' \in R$ with $l|_p = l'$. Let $l \Rightarrow r$ be generated by a literal K. $l' \Rightarrow r'$ cannot be in

R_K, for otherwise condition 5 would have prevented the generation of $l \Rightarrow r$. So $l' \Rightarrow r'$ is generated by a literal K' with $K' \succ_l K$. But then either $l' \succ l$, which is impossible because l' is a subterm of l. Or $l' = l$ and $r \succ r'$, but then l' would be reducible by $l \Rightarrow r$, violating condition 5 for $\mathrm{Gen}(K') = \{l' \Rightarrow r'\}$.

For arbitrary ground literals L, $R_L \subseteq R$, so R_L is also terminating, and R_L cannot contain critical pairs either. Hence, R_L is also convergent. □

Lemma 7.14 *For all ground literals L, if $R_L^* \models L$, then $R^* \models L$.*

Proof. Let $R_L^* \models L$.
Case 1: $L = (s \doteq t)$. R contains at least all the rewrite rules of R_L, i.e. $R \supseteq R_L$. Thus, the equation must also hold in R^*.
Case 2: $L = (\neg s \doteq t)$. According to Lemma 7.13, R_L is convergent, so s and t have distinct normal forms $s' \preceq s$ and $t' \preceq t$ w.r.t. R_L. Now consider rules $l \Rightarrow r \in R \setminus R_L$. By definition of R_L, their generating literals $l \doteq r$ must be larger than L in the literal ordering (they can't be equal because L is a negated equation). By the definition of $\succ_l$, this implies that $l \succ s \succeq s'$ and $l \succ t \succeq t'$. So rules in $R \setminus R_L$ can not further rewrite s' or t', hence these are the normal forms of s and t also w.r.t. R. And as they are distinct, $R^* \models \neg s \doteq t$. □

We can now show the central property of the model R^* constructed in Def. 7.12, namely that it satisfies all the irreducible instances (w.r.t R) of literals in $\mathcal{B}$ under certain conditions.

Lemma 7.15 (Model Generation) *Let $\mathcal{B}$ be a set of constrained literals and σ a ground substitution for the free variables in $\mathcal{B}$, such that*

- *$\mathcal{B}$ is closed under the application of the sup-p and sup-n rules, and*
- *there is no literal $\neg s \doteq t \ll A \in \mathcal{B}$ such that $\sigma s = \sigma t$ (syntactically) and σ satisfies A.*

Then $R^* \models L$ for all $L \in \mathit{irred}_R(\sigma, \mathcal{B})$.

Proof. Assume that this were not the case. Then there must be a *minimal* (w.r.t. $\succ_l$) L in $\mathit{irred}_R(\sigma, \mathcal{B})$ with $R^* \not\models L$. We distinguish two cases, according to whether L is an equation or a negated equation:

Case 1: $L = (s \doteq t)$. If $s = t$ syntactically, then clearly $R^* \models L$, so we may assume that $s \succ t$. As $R_L \subseteq R$, we certainly have $L \in \mathit{irred}_{R_L}(\sigma, \mathcal{B})$. Also, due to Lemma 7.14, we already have $R_L^* \not\models L$. But $\mathrm{Gen}(L) = \emptyset$, because otherwise the rule $s \Rightarrow t$ would be in R, implying $R^* \models L$. As conditions 1 through 4 for L generating a rule are fulfilled, condition 5 must be violated. This means that there is a rule $l \Rightarrow r \in R_L$ that reduces s, so $s|_p = l$ for some position p in s. Now let L be the variable-irreducible (w.r.t. R) instance of a constrained literal $L_0 = (s_0 \doteq t_0 \ll A) \in \mathcal{B}$. Similarly, let $l \Rightarrow r$ be generated by a literal $K = (l \doteq r) \prec_l L$ that is the variable-irreducible (w.r.t. R_K)

instance of a constrained literal $K_0 = (l_0 \doteq r_0 \ll B) \in \mathcal{B}$. It turns out that p must be a non-variable position in s_0, because otherwise, since $s = \sigma s_0$, we would have $p = p'p''$ with $s_0|_{p'} = x$ and $\sigma x|_{p''} = l$, thus σx would be reducible by $l \Rightarrow r \in R$, contradicting the variable-irreducibility of L.[3] From all this, it follows that an application of the *sup-p*-rule between the literals $L_0, K_0 \in \mathcal{B}$ is possible:

$$\textit{sup-p}\quad \frac{\begin{array}{c} s_0 \doteq t_0 \ll A \\ l_0 \doteq r_0 \ll B \end{array}}{s_0[r_0]_p \doteq t_0 \ll s_0|_p \equiv l_0 \;\&\; s_0 \succ t_0 \;\&\; l_0 \succ r_0 \;\&\; A \;\&\; B}$$

As $\mathcal{B}$ is required to be closed under rule applications, the resulting literal, call it L'_0, must be in $\mathcal{B}$. Now $L' := (s[r]_p \doteq t) = \sigma L'_0$ is a variable-irreducible (w.r.t. R) instance of L'_0: indeed, σ obviously satisfies the new constraint. Furthermore, σx is irreducible by R for any variable x occurring in s_0 or t_0. For an x occurring in r_0, σx is known to be irreducible by rules in R_K. But for rules $g \Rightarrow d \in R \setminus R_K$, we have $g \succeq l \succ r \succeq \sigma x$, so g cannot be a subterm of σx. This shows that σx is irreducible by R for all variables x in L'_0, so $L' \in \mathit{irred}_R(\sigma, \mathcal{B})$. Moreover, since l and r are in the same R^*-equivalence class, replacing l by r in s does not change the (non-)validity of $s \doteq t$, i.e. $R^* \not\models L'$. And finally, by monotonicity of the rewrite ordering $\succ$, $L \succ_l L'$. This contradicts the assumption that L is the minimal element of $\mathit{irred}_R(\sigma, \mathcal{B})$ which is not valid in R^*.

Case 2: $L = (\neg s \doteq t)$. If $s = t$ syntactically, then the second precondition of this lemma is violated, so we may assume $s \succ t$. Due to Lemma 7.14, $R^*_L \not\models L$, i.e. $R^*_L \models s \doteq t$. According to Lemma 7.13, R_L is convergent. Validity of $s \doteq t$ in R^*_L then means that s and t have the same normal form w.r.t. R_L. This normal form must be $\preceq t$, and thus $\prec s$. Therefore, s must be reducible by some rule $l \Rightarrow r \in R_L$ with $s|_p = l$ for some position p. As in case 1, let L be the variable-irreducible (w.r.t. R) instance of a constrained literal $L_0 = (\neg s_0 \doteq t_0 \ll A) \in \mathcal{B}$ and let $l \Rightarrow r$ be generated by a literal $K = (l \doteq r) \prec_l L$ that is the variable-irreducible (w.r.t. R_K) instance of a constrained literal $K_0 = (l_0 \doteq r_0 \ll B) \in \mathcal{B}$. Again as in case 1, p must be a non-variable position in s_0. It follows that an application of the *sup-n* rule is possible between L_0 and K_0:

$$\textit{sup-n}\quad \frac{\begin{array}{c} \neg s_0 \doteq t_0 \ll A \\ l_0 \doteq r_0 \ll B \end{array}}{\neg s_0[r_0]_p \doteq t_0 \ll s_0|_p \equiv l_0 \;\&\; s_0 \succ t_0 \;\&\; l_0 \succ r_0 \;\&\; A \;\&\; B}$$

We can now show, in complete analogy with case 1 that $L' := (\neg s[r]_p \doteq t) \in \mathit{irred}_R(\sigma, \mathcal{B})$, $R^* \not\models L'$ and $L \succ_l L'$, contradicting the assumption that L is minimal in $\mathit{irred}_R(\sigma, \mathcal{B})$ with $R^* \not\models L$. □

We now have all the necessary tools to show that our calculus is complete in the sense that there exists a finite closed tableau for any unsatisfiable set of clauses. We are going

[3]This is the place where the use of variable-irreducible instances is necessary. Otherwise, the combination of constraint inheritance and the non-variable-position condition would give problems. This idea is also used by Nieuwenhuis and Rubio [NR01], but they need a slightly more complicated notion of variable irreducibility because they work with clauses.

to show a little more, namely that a closed proof will be found if we simply expand the tableau in a fair way without requiring backtracking. Of course, this property is partly due to the fact that we postpone the instantiation of free variables to a global closure test. If we closed branches one at a time, we would have to backtrack over branch closures, but not—contrary to what is the case in the calculus of Degtyarev and Voronkov—over every application of the superposition rules. In order to state the completeness theorem, we need the following definition of a fair proof procedure.

Definition 7.16 *A* proof procedure *is a procedure that takes a set of clauses $\mathcal{C}$ and builds a sequence of tableaux $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots$ for $\mathcal{C}$ where $\mathcal{T}_0$ is the empty tableau, and each $\mathcal{T}_{i+1}$ results from the application of an ext or sup rule on one of the leaf goals of $\mathcal{T}_i$. A proof procedure* finds a proof for $\mathcal{C}$*, if one of the $\mathcal{T}_i$ is closed.*

The limit *of a sequence of tableaux constructed by a proof procedure is the possibly infinite union of all those tableaux. A* leaf goal *of such a limit is the union of all goals on some branch of the limit tableau.*

A proof procedure is fair*, if for any sequence of tableaux it constructs that does not contain a closed tableau, the following holds: If $\mathcal{T}$ is the limit of the sequence of constructed tableaux, then*

- *The ext-rule is applied infinitely often for every clause on every branch of $\mathcal{T}$.*
- *Every possible application of the sup-rules between two literals on a branch of $\mathcal{T}$ is eventually performed on that branch.*

Theorem 7.17 *Let $\mathcal{C}$ be an unsatisfiable set of clauses. Then a fair proof procedure finds a proof for $\mathcal{C}$.*

Proof. Assume that the procedure does not find a proof. Then it constructs a sequence of tableaux $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots$ with a limit $\mathcal{T}$. $\mathcal{T}$ has at least one open leaf goal for any instantiation of the free variables in $\mathcal{T}$. For assume that under a certain σ all leaf goals are closed. Then there is a literal $\neg s \doteq t \ll C$ in every leaf goal with $\sigma s = \sigma t$ and $\sigma \in \mathrm{Sat}(C)$. This literal is introduced at some (possibly inner) node of the limit tableau. Make a new tableau $\mathcal{T}'$ by cutting off every branch below some such node. Then σ still closes $\mathcal{T}'$ and $\mathcal{T}'$ has only branches of finite length and is finitely branching. Thus, by König's Lemma, $\mathcal{T}'$ must be a finite closed tableau for $\mathcal{C}$. One of the tableaux $\mathcal{T}_i$ must contain $\mathcal{T}'$ as initial subtableau, and thus $\mathcal{T}_i$ is closed under σ, contradicting the assumption that the procedure finds no proof.

We now fix the instantiation σ. Namely, σ should instantiate the free variables introduced by the *ext*-rule in such a way that every leaf goal of $\sigma\mathcal{T}$ contains at least one literal of each ground instance of every clause in $\mathcal{C}$. This is can be done because by fairness of the procedure, the *ext*-rule is applied infinitely often for each clause on every branch.

We have seen that there must now be a leaf goal $\mathcal{B}$ of $\mathcal{T}$, such that $\mathcal{B}$ is not closed by σ. We apply the model generation of Def. 7.12 on $\mathcal{B}$ and σ to obtain a set of rewrite rules $R = R_{\mathcal{B},\sigma}$. As $\mathcal{B}$ and σ obviously satisfy the preconditions of Lemma 7.15, every variable-irreducible instance of $\mathcal{B}$ is valid in R^*.

It now remains to show that every clause in $\mathcal{C}$ is valid in R^* to contradict the assumption that $\mathcal{C}$ is unsatisfiable. We do this by showing that all ground instances of clauses in $\mathcal{C}$ are valid. Let τC be a ground instance of $C \in \mathcal{C}$, where τ is a ground substitution for the variables occurring in C. We now define a new substitution τ' such that $\tau' x$ is the normal form w.r.t. R of τx. This makes $\tau' x$ irreducible by R for all variables x of C. Now $\tau' C$ is obtained from τC by replacement of a number of subterms by other subterms equivalent under R^*. Thus $R^* \models \tau C$ iff $R^* \models \tau' C$. By construction of σ and $\mathcal{B}$, there must be a literal $L \in C$ such that $\theta L \in \mathcal{B}$ for some renaming of variables θ, and such that $\tau' L = \sigma\theta L$. As θL carries no constraint, this makes $\tau' L$ a variable-irreducible instance of θL, so $R^* \models \tau' L$ and accordingly $R^* \models \tau' C$. □

7.4 A Calculus with Histories

The calculus proposed by Degtyarev and Voronkov has a relative termination property similar to the one described for the simplification rules in Theorem 6.11 and 6.14: In their calculus, only a finite number of applications of the superposition rules is possible without intervening applications of the γ-rule, which corresponds to our *ext*-rule.

A little surprisingly—after all, we did not use universal variables, which cause non-termination for the simplification rules—this is not the case for the superposition calculus considered so far.

Example 7.18 We start with a goal containing the following two literals:

$$\begin{gathered} 1 : fgx \doteq gx \\ 2 : gx \doteq a \end{gathered}$$

By repeated application of the *sup-p* rule, we can derive

$$\begin{gathered} 3(\textit{sup-p} \text{ of 2 on 1}) : gx \doteq fa \ll gx \succ a \\ 4(\textit{sup-p} \text{ of 3 on 1}) : gx \doteq ffa \ll gx \succ fa \\ 5(\textit{sup-p} \text{ of 4 on 1}) : gx \doteq fffa \ll gx \succ ffa \\ \vdots \end{gathered}$$

where the generated constraints have been suitably simplified. Indeed, all the generated constraints can be seen to be satisfiable (though of course by different instantiations), regardless of the reduction ordering chosen.

Such a derivation cannot occur in the calculus of Degtyarev and Voronkov, because they discard the formula that superposition takes place on. The cost of this is that backtracking over the order of applied superposition steps becomes necessary.

The practical value of this relative termination property is questionable. We will discuss this in Sect. 7.4.2. Still, it is at least of a certain theoretical value, so we will show how our calculus may be modified to make superposition derivations terminating. We shall see that backtracking is not required in our case.

We need to somehow simulate destructive rule applications in our calculus. More precisely, we have to prohibit a rule application if it depends on literals, which would previously have been discarded in a destructive calculus. We have already seen one possibility for doing this in Sect. 6.5.1, namely to add dis-unification constraints to formulae. We are going to follow a different approach here for two reasons. First, even the weak form of destructiveness caused by changing the constraint of an existing literal in a rule application would be rather difficult to integrate into our model generation proof, because in certain cases, the accumulated constraint can become unsatisfiable, so that the original formula is discarded. This would mean that it is no longer present in the 'leaf goal' of the limit tableau, see Sect. 7.5. Second, this is an opportunity to introduce a different technique for simulating destructive rules.

We start by explaining the general principle. We shall use a slight variation in the actual calculus. The idea is as follows: instead of discarding rewritten literals, we label each literal L with a *history* h_L, which is a list of (references to occurrences of) literals that *would have been* discarded during the derivation of L in a destructive calculus. We constrain the superposition rules in a way that excludes rule applications between L and a literal that occurs in the history h_L of L.

A calculus that discards the rewritten literal would then have a positive superposition rule like this:

$$\frac{\begin{array}{c} L = (s \doteq t \ll A \cdot h_L) \\ K = (l \doteq r \ll B \cdot h_K) \end{array}}{s[r]_p \doteq t \ll s|_p \equiv l \,\&\, s \succ t \,\&\, l \succ r \,\&\, A \,\&\, B \cdot \{L\} \cup h_L \cup h_K}$$

where p is a position in s, $s|_p$ is not a variable and $L \notin h_K$ and $K \notin h_L$.

The important part is the history of the new literal. Histories are propagated somewhat like constraints. Derivation of L would have implied discarding the literals in h_L, and likewise for h_K. The new superposition step now discards L, so we get the combined history of $\{L\} \cup h_L \cup h_K$.

This rule is entirely non-destructive. If we used DU-constraints, we would change the constraint of L to register the instantiations under which L could have been discarded. Here we remember the destruction of L in the new literal, while L itself is not changed.

With this formulation of the rules, our calculus would closely correspond to that of Degtyarev and Voronkov, though one can show that it still allows certain derivations excluded by the destructive version.

Example 7.19 Assume a goal with three literals

$$\begin{array}{c} 1 : gfc \doteq a \\ 2 : c \doteq b \\ 3 : gfc \doteq fb \end{array}$$

and an LPO with precedence $g > f > c > b > a$. Now superposition of 2, resp. 3

on 1 produces two new literals:

$$4(\textit{sup-p}\text{ of 2 on 1}) : gfb \doteq a \cdot \{1\}$$
$$5(\textit{sup-p}\text{ of 3 on 1}) : fb \doteq a \cdot \{1\}$$

We can now apply superposition of 5 on 4, to derive

$$6(\textit{sup-p}\text{ of 5 on 4}) : ga \doteq a \cdot \{1,4\}$$

This step would be possible in no derivation of a destructive calculus, as only either 4 *or* 5 could have been derived, due to the destruction of 1.

We are not going to let this worry us; we shall conduct our proofs for an even more restrictive calculus for the following reasons:

- As we show completeness of a more restrictive calculus, our completeness result is strictly stronger. It works in exactly the same way for a weaker restriction.
- The proof is not further complicated by the stronger restriction.
- The termination property is much easier to show in our calculus.
- In a way, the underlying principle of our completeness proof becomes clearer with the more restrictive calculus.

We shall call the literals introduced by the *ext*-rule (as opposed to those introduced by applications of the superposition rules) *ext-literals*. Our calculus will record in the history of each literal which ext-literals were involved in its derivation. Putting other literals into the histories is not going to be necessary. We allow each ext-literal to be used at most once in the derivation of a literal, which is easily formalized by requiring the histories of literals used in the superposition rules to be *disjoint*. In a destructive, backtracking formulation analogous to the one of Degtyarev and Voronkov, this would mean that *both* literals used in a superposition step are discarded.

Here are the three rules of our calculus with histories:

$$\textit{ext}\ \frac{}{L_1 \cdot \{L_1\} \quad | \quad \cdots \quad | \quad L_k \cdot \{L_k\}}$$

where $\{L_1, \ldots, L_k\} = \theta C$, with $C \in \mathcal{C}$
and θ renames each variable in C into a new (free) variable.

$$\textit{sup-p}\ \frac{\begin{array}{c} s \doteq t \ll A \cdot h_1 \\ l \doteq r \ll B \cdot h_2 \end{array}}{s[r]_p \doteq t \ll s|_p \equiv l \,\&\, s \succ t \,\&\, l \succ r \,\&\, A \,\&\, B \cdot h_1 \cup h_2}$$

where p is a position in s, $s|_p$ is not a variable and $h_1 \cap h_2 = \emptyset$.

$$\textit{sup-n}\ \frac{\begin{array}{c} \neg s \doteq t \ll A \cdot h_1 \\ l \doteq r \ll B \cdot h_2 \end{array}}{\neg s[r]_p \doteq t \ll s|_p \equiv l \,\&\, s \succ t \,\&\, l \succ r \,\&\, A \,\&\, B \cdot h_1 \cup h_2}$$

where p is a position in s, $s|_p$ is not a variable and $h_1 \cap h_2 = \emptyset$.

Again, the superposition rules *sup-p* and *sup-n* are only applied if the constraint of the new literal is satisfiable. The two literals involved as premises in the *sup-p*-rule are required to be distinct.

As in the simple calculus of the previous section, a ground instantiation σ closes a goal G of a tableau, if there is a constrained negated equation $\neg s \doteq t \ll A \cdot h \in \mathcal{B}$ such that $\sigma s = \sigma t$ (that is syntactic identity) and σ satisfies A. The whole tableau is closed, if there is a single instantiation σ that closes all leaf goals simultaneously.

One easily sees that the history of every literal has at least one element, and that the literals with a one element history are precisely the ext-literals.

7.4.1 Completeness of the Calculus with Histories

For our completeness proof, we have to modify the proof given in the previous section slightly, to cope with the disjoint history restriction in the *sup*-rules. We require the following notions.

Definition 7.20 *Let $\mathcal{S}$ be a set of constrained literals with history and σ an instantiation for the free variables in $\mathcal{S}$. Two literals $L, K \in \mathcal{S}$ are called* variants*, if they are equal up to renaming of free variables, if histories are not regarded.*[4] *They are called* copies *(under σ) if moreover the free variables are assigned the same ground terms under σ. $\mathcal{S}$ is called* rich *(under σ), if every literal $L \in \mathcal{S}$ has an infinite number of copies with* pairwise disjoint histories *in $\mathcal{S}$.*

For instance, $f(X) \doteq Y \ll X \equiv a \cdot \{L_1, L_2\}$ and $f(U) \doteq V \ll U \equiv a \cdot \{L_1, L_3\}$ are variants. They are also copies under σ if $\sigma X = \sigma U$ and $\sigma Y = \sigma V$.

As will become apparent in the proof of Theorem 7.22, we will do the model construction with only a subset of the literals on an open leaf goal. To avoid confusion, we are going to denote the concerned sets of constrained literals with history $\mathcal{S}$ instead of $\mathcal{B}$ as in the previous section.

The construction of a model from a set $\mathcal{S}$ works exactly as in Def. 7.12. The only new aspect is that the literals in $\mathcal{S}$ have histories: we simply forget those when applying a ground substitution. So $\mathit{irred}_R(\sigma, \mathcal{S})$ shall simply be a set of ground literals without history as before. Thus, for the generated set of ground rewrite rules $R = R_{\mathcal{S},\sigma}$, Lemmas 7.13 and 7.14 hold as before.

The differences are in the Model Generation Lemma (Lemma 7.15 for the simple calculus) and the actual completeness proof. Most of the Model Generation Lemma and its proof are actually identical to the simple version, but we shall repeat the proof here to make it more readable and also to make sure that we do not accidentally skip an important difference. The new parts are marked by a gray bar in the margin.

Lemma 7.21 (Model Generation) *Let $\mathcal{S}$ be a set of constrained literals with history and σ an instantiation for the free variables in $\mathcal{S}$, such that*

- *$\mathcal{S}$ is closed under the application of the sup-p and sup-n rules,*

[4]The variable renaming also applies to the constraints.

- *there is no literal $\neg s \doteq t \ll A \cdot h \in \mathcal{S}$ such that $\sigma s = \sigma t$ (syntactically) and σ satisfies A, and*
- *$\mathcal{S}$ is rich under σ.*

Then $R^* \models L$ for all $L \in \mathit{irred}_R(\sigma, \mathcal{S})$.

Proof. Assume that this were not the case. Then there must be a *minimal* (w.r.t. $\succ_l$) L in $\mathit{irred}_R(\sigma, \mathcal{S})$ with $R^* \not\models L$. We distinguish two cases, according to whether L is an equation or a negated equation:

Case 1: $L = (s \doteq t)$. If $s = t$ syntactically, then clearly $R^* \models L$, so we may assume that $s \succ t$. As $R_L \subseteq R$, we certainly have $L \in \mathit{irred}_{R_L}(\sigma, \mathcal{S})$. Also, due to Lemma 7.14, we already have $R_L^* \not\models L$. But $\mathrm{Gen}(L) = \emptyset$, because otherwise the rule $s \Rightarrow t$ would be in R, implying $R^* \models L$. As conditions 1 through 4 for L generating a rule are fulfilled, condition 5 must be violated. This means that there is a rule $l \Rightarrow r \in R_L$ that reduces s, so $s|_p = l$ for some position p in s. Now let L be the variable-irreducible (w.r.t. R) instance of a constrained literal $L_0 = (s_0 \doteq t_0 \ll A \cdot h_L) \in \mathcal{S}$. Similarly, let $l \Rightarrow r$ be generated by a literal $K = (l \doteq r) \prec_l L$ that is the variable-irreducible (w.r.t. R_K) instance of a constrained literal $K_0 = (l_0 \doteq r_0 \ll B \cdot h_K) \in \mathcal{S}$. As $\mathcal{S}$ is rich, there are infinitely many copies under σ of L_0 with pairwise disjoint histories. Each of the finitely many elements of h_K can be contained in the history of at most one of these copies, and all the remaining ones have a history disjoint to h_K. So we may assume that L_0 and K_0 are chosen in a way that h_K and h_L are disjoint. Further, it turns out that p must be a non-variable position in s_0, because otherwise, since $s = \sigma s_0$, we would have $p = p'p''$ with $s_0|_{p'} = x$ and $\sigma x|_{p''} = l$, thus σx would be reducible by $l \Rightarrow r \in R$, contradicting the variable-irreducibility of L. From all this, it follows that an application of the *sup-p*-rule between the literals $L_0, K_0 \in \mathcal{S}$ is possible:

$$\textit{sup-p}\quad \frac{\begin{array}{c} s_0 \doteq t_0 \ll A \cdot h_L \\ l_0 \doteq r_0 \ll B \cdot h_K \end{array}}{s_0[r_0]_p \doteq t_0 \ll s_0|_p \equiv l_0 \,\&\, s_0 \succ t_0 \,\&\, l_0 \succ r_0 \,\&\, A \,\&\, B \cdot h_L \cup h_K}$$

As $\mathcal{S}$ is required to be closed under rule applications, the resulting literal, call it L_0', must be in $\mathcal{S}$. Now $L' := (s[r]_p \doteq t) = \sigma L_0'$ is a variable-irreducible (w.r.t. R) instance of L_0': indeed, σ obviously satisfies the new constraint. Furthermore, σx is irreducible by R for any variable x occurring in s_0 or t_0. For an x occurring in r_0, σx is known to be irreducible by rules in R_K. But for rules $g \Rightarrow d \in R \setminus R_K$, we have $g \succeq l \succ r \succeq \sigma x$, so g cannot be a subterm of σx. This shows that σx is irreducible by R for all variables x in L_0', so $L' \in \mathit{irred}_R(\sigma, \mathcal{S})$. Moreover, since l and r are in the same R^*-equivalence class, replacing l by r in s does not change the (non-)validity of $s \doteq t$, i.e. $R^* \not\models L'$. And finally, by monotonicity of the rewrite ordering $\succ$, $L \succ_l L'$. This contradicts the assumption that L is the minimal element of $\mathit{irred}_R(\sigma, \mathcal{S})$ which is not valid in R^*.

Case 2: $L = (\neg s \doteq t)$. If $s = t$ syntactically, then the second precondition of this lemma is violated, so we may assume $s \succ t$. Due to Lemma 7.14, $R_L^* \not\models L$, i.e. $R_L^* \models s \doteq t$. According to Lemma 7.13, R_L is convergent. Validity of $s \doteq t$ in R_L^* then means that

s and t have the same normal form w.r.t. R_L. This normal form must be $\preceq t$, and thus $\prec s$. Therefore, s must be reducible by some rule $l \Rightarrow r \in R_L$ with $s|_p = l$ for some position p. As in case 1, let L be the variable-irreducible (w.r.t. R) instance of a constrained literal $L_0 = (\neg s_0 \doteq t_0 \ll A \cdot h_L) \in \mathcal{S}$ and let $l \Rightarrow r$ be generated by a literal $K = (l \doteq r) \prec_l L$ that is the variable-irreducible (w.r.t. R_K) instance of a constrained literal $K_0 = (l_0 \doteq r_0 \ll B \cdot h_K) \in \mathcal{S}$. Again as in case 1, p must be a non-variable position in s_0, and we can choose L_0 and K_0 with disjoint histories. It follows that an application of the *sup-n* rule is possible between L_0 and K_0:

$$\textit{sup-n}\quad \frac{\begin{array}{c}\neg s_0 \doteq t_0 \ll A \cdot h_L \\ l_0 \doteq r_0 \ll B \cdot h_K\end{array}}{\neg s_0[r_0]_p \doteq t_0 \ll s_0|_p \equiv l_0 \,\&\, s_0 \succ t_0 \,\&\, l_0 \succ r_0 \,\&\, A \,\&\, B \cdot h_L \cup h_K}$$

We can now show, in complete analogy with case 1 that $L' := (\neg s[r]_p \doteq t) \in \mathit{irred}_R(\sigma, \mathcal{S})$, $R^* \not\models L'$ and $L \succ_l L'$, contradicting the assumption that L is minimal in $\mathit{irred}_R(\sigma, \mathcal{S})$ with $R^* \not\models L$. □

The main point is that if $\mathcal{S}$ is rich, we can find enough copies of the required literals that some of them have disjoint histories. Now in the actual completeness proof, we have to extract a rich set of literals from an open branch in such a way that the validity of the irreducible instances of that set will imply the validity of each of the clauses in our clause set.

Using the definitions of a fair proof procedure from Def. 7.16, we can now show the following completeness theorem.

Theorem 7.22 *Let $\mathcal{C}$ be an unsatisfiable set of clauses. Then a fair proof procedure for the calculus with histories finds a proof for $\mathcal{C}$.*

Proof. Assume that the procedure does not find a proof. As in the proof of Theorem 7.17, we can conclude that it constructs in the limit an infinite tableau $\mathcal{T}$ which has at least one open leaf goal under any instantiation for the free variables in $\mathcal{T}$.

We now fix the instantiation σ. Namely, σ should instantiate the free variables introduced by the *ext*-rule in such a way that every leaf goal of $\sigma\mathcal{T}$ contains *infinitely many* occurrences of literals of each ground instance of every clause in $\mathcal{C}$. This is possible because the *ext*-rule is applied infinitely often for each clause on every branch, and using a dovetailing process that lets each of the ground instantiations be used infinitely often.

There must now be a leaf goal $\mathcal{B}$ of $\mathcal{T}$, such that $\mathcal{B}$ is not closed by σ. As there are infinitely many occurrences of literals of each ground instance of every clause on $\mathcal{B}$, and every clause is finite, for every ground instance τC of every clause, there must be at least one literal $L_{\tau C} \in \tau C$, such that there are infinitely many ext-literals $L' \in \mathcal{B}$ with $\sigma L' = L_{\tau C}$.

Collect all these ext-literals $L_{\tau C}$ on $\mathcal{B}$ in a set $\mathcal{E}^\infty$. As we are dealing with ext-literals, the histories of literals in $\mathcal{E}^\infty$ are disjoint, so $\mathcal{E}^\infty$ is rich under σ. Now define $\mathcal{B}^\infty$ to contain all literals of $\mathcal{E}^\infty$ as well as all literals on $\mathcal{B}$ derived from literals in $\mathcal{E}^\infty$ alone.

As $\mathcal{B}$ is closed under *sup*-rule applications by fairness of the proof procedure, so is $\mathcal{B}^\infty$. Furthermore, $\mathcal{B}^\infty$ is rich, as can be seen by induction on the number n of literals in the history of a given literal L: For $n = 1$, L is an ext-literal, so $L \in \mathcal{E}^\infty$. Hence there are infinitely many copies of L with pairwise disjoint histories. For $n > 1$, L must be derived by an application of a *sup*-rule from literals with a history smaller than n. The induction hypothesis guarantees an infinite number of copies with pairwise disjoint histories of these literals in $\mathcal{B}^\infty$. The same rule application is obviously possible between these copies, and as $\mathcal{B}^\infty$ is closed under *sup* applications, one easily sees that there must be infinitely many copies of L.

We apply the model generation of Def. 7.12 on $\mathcal{B}^\infty$ and σ to obtain a set of rewrite rules $R = R_{\mathcal{B}^\infty,\sigma}$. As $\mathcal{B}^\infty$ and σ satisfy the preconditions of Lemma 7.21, every variable-irreducible instance of $\mathcal{B}^\infty$ is valid in R^*.

It now remains to show that every clause in $\mathcal{C}$ is valid in R^* to contradict the assumption that $\mathcal{C}$ is unsatisfiable. This is done as in the proof of Theorem 7.17, except that it now suffices to take expansions contributing to $\mathcal{E}^\infty$ into account. We must show that all ground instances of clauses in $\mathcal{C}$ are valid. Let τC be a ground instance of $C \in \mathcal{C}$, where τ is a ground substitution for the variables occurring in C. We now define a new substitution τ' such that $\tau' x$ is the normal form w.r.t. R of τx. This makes $\tau' x$ irreducible by R for all variables x of C. Now $\tau' C$ is obtained from τC by replacement of a number of subterms by other subterms equivalent under R^*. Thus $R^* \models \tau C$ iff $R^* \models \tau' C$. By construction of σ and $\mathcal{B}^\infty$, there must be a literal $L \in C$ such that $\theta L \in \mathcal{E}^\infty \subset \mathcal{B}^\infty$ for some renaming of variables θ, and such that $\tau' L = \sigma\theta L$. As θL carries no constraint, this makes $\tau' L$ a variable-irreducible instance of θL, so $R^* \models \tau' L$ and accordingly $R^* \models \tau' C$. □

7.4.2 Termination

The completeness proof just given is simpler than that of Degtyarev and Voronkov *although* our calculus is more restrictive. In contrast, our proof of the relative termination property is simpler than theirs, *because* our calculus is more restrictive. Indeed, we can prove this property with the histories alone, without needing arguments about the ordering restrictions expressed in the constraints.

Theorem 7.23 *Starting from a finite tableau $\mathcal{T}$, only a finite number of sup-rule applications is possible without intervening applications of the ext-rule.*

Proof. As the *sup*-rules do not introduce new branches, it suffices to show this property for each of the finitely many branches of $\mathcal{T}$. The *sup*-rules combine the disjoint history sets of used literals, so the size of the history of the resulting literal is the sum of the sizes of the used literals' histories. Only ext-literals have a history of size one.

We show by induction on n that only finitely many literals with a history of at most n literals can be derived. For $n = 1$, this is the case, since we start out with only finitely many ext-literals, and we do not get any new ones. For $n > 1$, a literal must be the result of a *sup*-application between literals of history size less than n. By induction hypothesis, there can be only finitely many of those. Also, there are only finitely many ways to apply a *sup*-rule between two given literals, because the rule applications are determined by the position p at which the terms are overlapped.

No history can get larger than the number of ext-literals on the branch, so one can only derive a finite number of new literals altogether. □

It should be remarked that the history restriction employed is rather strong. In a backtracking setting, this would correspond to discarding both used literals in every superposition step. The calculus of Voronkov and Degtyarev discards only the rewritten one, i.e. the first premise in our notation. As we have already mentioned, we choose the given variant because it is more restrictive, meaning that we have a stronger completeness result, without making the completeness proof more complicated.

On the other hand, our restriction to disjoint histories is so strong, that it prompts the question whether it is useful in practice. But of course, that question has to be asked of any restriction. We introduced the history restriction to get a calculus which enjoys the relative termination property just proven. Only experimentation can show which restriction is useful to ensure termination in practice. In fact, it is not even clear whether the termination property is of any practical value at all:

- At first sight, the termination property makes it easier to implement a fair proof procedure: One can apply the *sup*-rules exhaustively before resorting to further *ext*-expansions. However, one still needs a fair strategy to choose the next extension clause on a branch. If one can implement an intelligent procedure to do this, one should also be able to choose between extension and superposition. Or, vice versa, if it is sufficient to just put pending *ext*-expansions in a FIFO queue, why should it not be good enough to use the same queue for superposition steps?

- As Beckert has pointed out [Bec93, Bec94], it is crucial for the efficiency of equality reasoning to take universal variables into account (see Sect. 5.7). The reason for this is that many application domains naturally lead to universal unit equality axioms, like commutativity, associativity, idempotency, inverse function relationships ($\forall x.fgx = x$), etc. Introducing a new copy with rigid variables for every application of such axioms leads to similar redundancies as were shown for the simplification rules in Sect. 6.6. If universal variables are used, only one copy of these axioms ever needs to be introduced on a branch.

  It turns out that the superposition rules with universal variables correspond to a variant of unfailing Knuth-Bendix completion [BDP89], which does not terminate in general. UKBA behaves very well in practice, so it is probably not sensible to restrict it artificially only to enforce termination.

- The regularity restriction (see Sect. 5.8 and 6.8) requires literals introduced by rule applications to be new to their branches under the closing substitution. This is a very common and successful restriction to eliminate redundancy in proof search. The calculus with histories is not complete if we require regularity, see Sect. 7.4.3. It is not clear whether the calculus of Degtyarev and Voronkov is compatible with regularity. On the other hand, the simple calculus of Sect. 7.3 obviously *is*, since only one copy of each ground instance is needed.

To summarize, it seems that in an efficient implementation of a tableau calculus with superposition, the termination property is not really important, and maybe cannot even be sensibly maintained at all.

Of course, the termination property can be bestowed on any calculus by a simple trick: One takes an arbitrary fair strategy and codes it into the calculus. As every possible rule application gets scheduled at some point by a fair procedure, and extension with a clause is always possible, it follows that only finitely many *sup*-applications are performed in between.

Admittedly, it is nonsense to code the whole proof procedure into the calculus. But only experimentation can show how far one should go.

7.4.3 Regularity

In this section, we are going to consider the problems with the regularity restriction mentioned in the previous section in a little more detail. For this discussion, we do not need the technique for subsuming regularity presented in Sect. 6.8, because we are dealing only with clausal tableaux. We shall simply require that first, closing instantiations σ which lead to two equal literals on a branch under σ are not considered, and second, rule applications that lead to a repetition of literals under *every* instantiation are forbidden. How this should be implemented shall not be an issue here.

The most important effect of the regularity restriction in a first order calculus is that it prevents expanding the same clause twice with the same instantiation on one branch. We shall call this the 'economic instantiation' property.

One can easily check that the rigid basic superposition calculus without histories of Sect. 7.3 is compatible with the regularity restriction. Indeed, the completeness proof only requires *one* literal of every ground instance of each clause to be on a branch.

The situation is different for the calculus with histories. Regularity can still be required, if instances of literals with different histories are regarded as different in the regularity condition. But the condition would then be very weak, because the *ext*-rule introduces new literals with new histories each time, so we would not have the economic instantiation property. In other words, a sensible definition of regularity should not take histories into account. On the other hand, if we disregard histories in the regularity condition, we regain the economic instantiation property, but one can easily see that the calculus is no longer complete. This is reflected by the requirement of having infinitely many copies of literals on a branch in the completeness proof.

To summarize, if we take a useful definition of regularity, the calculus *without* histories is compatible with regularity, but does not have the relative termination property, while the calculus *with* histories has the relative termination property, but is incompatible with regularity.

As we remarked earlier, the history restriction used in our calculus is rather strong. This suggests the question, whether there is a restriction of our calculus which has the termination property and is compatible with a reasonably defined regularity restriction. The answer is yes, because the simple calculus is compatible with regularity and the trick mentioned at the end of the previous section allows us to endow it with the termination property. Of course, this is no interesting answer, as the restriction produced by coding the proof procedure into the calculus is not very natural. We thus reformulate our question:

> Is there a *natural* restriction of our calculus which has the termination property and is compatible with a reasonably defined regularity restriction?

A natural restriction would be one, for instance, that somehow reflects discarding rewritten or otherwise redundant literals. A reasonable regularity restriction should at least entail the economic instantiation property.

It is not stated in [DV98] whether the calculus of Degtyarev and Voronkov is compatible with regularity. One should remember that their calculus is destructive, so a suitable regularity condition should demand not only that instances of literals present on a branch are not duplicated by rule applications, but also that instances of literals that *were* present but have since been discarded are not duplicated. Otherwise, we would not get the economic instantiation property.

Reconsider the initial attempt at a calculus with histories mentioned at the beginning of Sect. 7.4 on page 114. We tried to find a completeness proof for that calculus (which corresponds closely to the one of [DV98]) that would be compatible with regularity. In particular, we considered a ground version of that calculus that does not need constraints and works only on ground literals with histories. We were able to show completeness of that ground calculus by the model generation technique without requiring infinitely many copies of literals. This means that the ground calculus is compatible with regularity. We shall not spell out the proof here, as it is similar to the ones already given. The main differences are as follows: The variable irreducibility restriction is obviously not needed for the ground case, instead we restrict model generation to literals with maximal histories. The model generation proof is adapted to generate a model in which all literals with maximal history are valid. From the validity of these in an open goal, we can then inductively infer the validity of all other literals.

Unfortunately, we have not succeeded in lifting this ground proof to a version with free variables. The restriction to literals with maximal history conflicts with the restriction to variable-irreducible instances which we need to cope with the non-variable-position restriction, leading to very tangled interdependencies which we were not able to resolve.

The conclusion is that we currently have no answer to the question posed above.

7.5 Using Dis-Unification Constraints for Superposition

In Sect. 7.4, we used history lists to emulate the deletion of literals from a branch in a non-destructive way that eliminates the need to backtrack over *sup*-rule applications. Another possibility to model deletion of literals is to use dis-unification (DU) constraints as in Sect. 6.5.1. We shall briefly discuss this possibility in this section.

Assume that the following rule application is possible between two literals:

$$sup\text{-}p \quad \frac{\begin{array}{c} s \doteq t \ll A \\ l \doteq r \ll B \end{array}}{s[r]_p \doteq t \ll A \,\&\, B \,\&\, C}$$

Where $C = (s|_p \equiv l \,\&\, s \succ t \,\&\, l \succ r)$ is the new constraint introduced by the rule application. One could now model the deletion of the rewritten literal $s \doteq t \ll A$ by *modifying* the constraint of that literal to $A \,\&\, !(B \,\&\, C)$:[5] this implies that for instantiations σ under which the superposition is possible, namely if σ satisfies $A \,\&\, B \,\&\, C$, the constraint of the deleted literal is no longer valid.

This method leads to a destructive calculus, since the constraints of literals are changed as the proof is expanded. Accordingly, more complicated techniques are required for a completeness proof. Such techniques are given by Nieuwenhuis and Rubio [NR95b, NR01] for resolution saturation calculi. Amongst other modifications, one needs to consider the persistent literals on a branch, instead of just the union of all goals as we did until now. A literal is called persistent, if it is introduced in some goal and never discarded by rule applications below that goal.

Also, one needs a rather complicated fairness condition: It is not sufficient to require (as in resolution saturation procedures) that all superpositions between persistent literals of a branch are eventually executed: Assume for instance that a superposition with K is applicable on some literal L, and that this superposition is needed to close the proof. There are situations where successive superposition steps with other literals can be applied on L, leading to a sequence of constrained literals $L', L'', \ldots$ with constraints getting more and more restrictive, but without becoming unsatisfiable. Neither the literal L or any of its descendants is then persistent on the branch, so superposition with K might never take place.

We have tried to fix the fairness condition (or rather the notion of persistency) to take account of this difficulty. But we still did not succeed in showing completeness of the resulting procedure. To give an intuition for the problem, let us mention that one still has to restrict model generation to variable-irreducible literals in order to cope with the non-variable-position restriction, while literals on the branch might be rewritten by superposition with literals which later turn out not to be variable irreducible.

This state of affairs is rather unsatisfying. One can easily construct a *ground* calculus which closely corresponds to the calculus with DU constraints. In this calculus, the *ext* rule introduces (guessed) ground instances of clauses instead of free variables, no constraints are needed, and the first premise of the superposition rules is actually deleted. We have been able to show completeness (without backtracking over superposition applications) of this destructive ground calculus. The proof even shows compatibility with regularity. But like for the ground calculus with histories mentioned in the previous section, we have not been able to lift this proof due to the afore-mentioned problems with the non-variable-position restriction.

[5] Remember that we use '!' to designate negation in constraints.

7.6 Tableaux with Basic Ordered Paramodulation

At the beginning of this chapter, we claimed that the model generation completeness proof can easily be adapted to variants of the calculus. In this section, we shall try to demonstrate how variations of calculi and completeness proofs can be carried over from known results for resolution-based calculi.

There is a more restrictive form of equality handling known in the resolution community as *basic ordered paramodulation* [BGLS95]. In comparison to basic superposition, the basicness restriction is strengthened: One forbids paramodulation below a position where a previous paramodulation step has taken place. The price to pay is that equations have to be applied on both sides of literals and not only the maximal side as for basic superposition. Still, basic ordered paramodulation seems to be very effective in practice [McC97].

Using constrained literals, one can easily enforce this stronger basicness restriction by introducing a new free variable in the equality handling rules. The *sup-p* rule becomes:[6]

$$\textit{par-p}\quad \frac{\begin{array}{c} s \doteq t \ll A \\ l \doteq r \ll B \end{array}}{s[X]_p \doteq t \ll X \equiv r \,\&\, s|_p \equiv l \,\&\, l \succ r \,\&\, A \,\&\, B}$$

where p is a position in s, $s|_p$ is not a variable,
and X is a new (free) variable.

Note how the constraint forces X to be instantiated with r, and that the restriction $s \succ t$ is gone.[7] The *par-n*-rule is exactly analogous. This modification is a straightforward adaptation of the formulation of basic ordered paramodulation using constraint inheritance given by Nieuwenhuis and Rubio in [NR01].

How do we show completeness of our modified calculus? We cite [NR01]:

> The completeness proof is an easy extension of the previous results by the model generation method. It suffices to modify the rule generation by requiring, when a rule $l \Rightarrow r$ is generated, that both l and r are irreducible by R_C, instead of only l as before, and to adapt the proof of Theorem 5.6 accordingly, which is straightforward.

All we need is a little 'signature mapping' to apply this to our situation: their Theorem 5.6 corresponds closely to our Lemma 7.15. They have R_C instead of our R_L because they have to work with ground clauses, where we can use literals. Otherwise, this statement applies exactly to our case.

In the definition of the model generation process (Def. 7.12 on page 109), let us replace condition 5 by

[6]We do not use the disjoint history restriction here in order to make things simpler to read. It should however be no problem to use that restriction with basic ordered paramodulation.

[7]It might seem that introducing a new free variable is not a good idea. But these ones are harmless, as there is no need to search for their instantiation. It is determined by the instantiations of the free variables in r. In fact, these new variables are universal as they are restricted to a fixed instantiation by the constraint.

5. *l and r are irreducible w.r.t. R_L.*

A close scrutiny of the proofs of Lemmas 7.13 and 7.14 satisfies us that they are still valid after this modification. And it is indeed quite straightforward to adapt the proof of Lemma 7.15 on page 110 ff.: for case 1, we drop the assumption that $s \succ t$, and infer that condition 5 must be violated as before. As we take a symmetric view of equations, we can now assume that it is s that is reducible by some rule in R_L. One then shows as before that a *sup-p* application is possible. Showing that L' is a variable-irreducible instance of some L'_0 is even simpler than before, because we do not need to account for variables in r_0. To show that σX is irreducible, note that $\sigma X = r$, and as $l \doteq r$ generates a rule, condition 5 guarantees that r is irreducible. Similar modifications apply for case 2. All this corresponds exactly to what needs to be done for resolution.

The only new and tableau-specific part is that σ has to provide an instantiation for the free variables X introduced in the paramodulation steps in such a way that the new constraints are satisfied. But fortunately, this is also easily done: in an induction over the superposition steps leading to the deduction of a literal, let $\sigma X := \sigma r_0$ for a free variable X introduced by a superposition with $l_0 \doteq r_0 \ll B$.

We think that this example is strong evidence in support of our claim that model generation completeness proofs are a good basis for adapting known results from resolution with superposition or paramodulation to a tableau setting.

7.7 Related Work

Using techniques based on ordered rewriting is not the only way to add efficient equality handling to a tableau prover. We believe that it is the most powerful way because of the success of such techniques in resolution provers like SPASS [Wei01] or Vampire [RV01].

The other commonly employed technique is to transform problems in a way that makes equality axioms redundant [Bra75, MS97, BGV97]. These transformation techniques analyze the problem, in order to find possible instances of paramodulation steps, and code the results of these paramodulations into the problem before the actual proof search starts. Classically, term orderings are not used, although [BGV97] introduce ordering constraints in some cases. The great advantage of transformation techniques is that one can add equality handling to a prover by a pre-processing step, instead of changing the actual prover. On the other hand, we expect a direct integration of equality handling into the calculus to be more efficient.

Another technique for equality handling in sequent based calculi has been proposed by Gallier and Snyder [GS89] under the name of *lazy paramodulation*. In this approach, equations are only applied on top level terms, but term orderings cannot be used. We have not investigated how this approach could be used in an incremental closure prover, but it is to be expected that syntactic unification constraints could be used again. Of course, one would not use the model generation technique to show completeness for such a calculus.

Letz and Stenz [LS02] have published an overview on the integration of equality handling into Billon's disconnection calculus [Bil96]. It is interesting to see that they also encounter problems with the regularity restriction.

7.8 Summary

In this chapter, we have shown how equality handling may be integrated into an incremental closure prover using ordered superposition/paramodulation rules with constraint propagation. We demonstrated how the completeness of such calculi can be shown using model generation techniques known from resolution calculi with only few additional tableau-specific ingredients. Though completeness of a similar calculus had previously been established in [DV97], using a different approach, our proof is much shorter, and we have demonstrated in Sect. 7.6 that it is easily adapted to related calculi.

In Sect. 7.4, we have shown how a termination property can be enforced for such calculi using a disjoint history restriction, and how completeness may be proved in presence of such a restriction. We have also briefly discussed the practical usefulness of the termination property in such calculi. In Sect. 7.4.3, we have pointed out certain problems in combining the termination property with a suitable regularity restriction.

The techniques described in this chapter have not yet been implemented. A good implementation of equality handling based on ordered rewriting is by no means a simple task: efficient satisfiability checks have to be implemented for ordering constraints, and these are significantly more complex than those for unification or dis-unification constraints. For fully automated use, a suitable term ordering has to be chosen heuristically. And maybe most importantly, new indexing structures are needed to quickly find possible superposition steps between the literals on a branch. The implementation and evaluation of the techniques proposed in this chapter is thus future work.

8 Future Work

In this short chapter, we shall identify some possible areas for future research. Concerning the basic framework of incremental closure presented in Chapters 4 and 5, some topics have already been suggested.

- Depending on the nature of the problems to be solved, the instantiation buffers in MergerSinks can become large. In that case, suitable indexing structures have to be developed, see Sect. 5.5. In particular, an adaptation of Graf's substitution trees [Gra96, Gra94] would seem to be a promising approach.
- Heuristics need to be developed for the two main indeterminisms in the incremental closure procedure, namely goal selection and formula selection, see Sect. 5.2 and 5.3. In a prover without backtracking, one can afford to spend more time on such heuristics than would be acceptable in a backtracking prover, because rule applications are not repeated.
- We mentioned the possibility of a multi processor implementation of the incremental closure approach at the end of Sect. 4.8. For problem domains were proof trees typically become large, but only few instantiations reach the top Merger objects, such an implementation might be worthwhile.

A possibility we have not yet mentioned is to include some sort of global redundancy notion into the proof procedure. As the whole tableau is always present, it might be possible to detect rule applications which are redundant with respect to a tableau, and not only with respect to a branch or goal. In some cases, such redundancy criteria might lead to termination of the proof procedure for satisfiable input formulae. In other words, one would obtain a decision procedure for some subclass of first order logic.

For the simplification rules of Chapter 6, the nost interesting open question is which simplification strategy to use. In particular, although the NNF hyper tableau strategy works quite well for many problems, a more goal-oriented strategy would be useful in some cases. It is also not clear how the potential cyclicity of simplification rules with universal variables is best dealt with, see Sect. 6.6. Weight functions and other term orderings should be investigated to find a good solution.

Concering the superposition based equality handling of Chapter 7, the next step is of course to experiment with an implementation. In particular, it would be interesting to see what impact various restrictions ensuring the relative termination property of Theorem 7.23 have both on performance of the prover and on implementation complexity. Maybe fairness can be achieved with a uniform approach for simplification and superposition rules.

An obvious extension of our results would be a version of the calculus that permits predicates other than equality and that does not require problems to be in clausal form. We expect this to be quite straight-forward.

Universal variables are known to be important for efficiency. It is expected that the given calculi and proofs can easily be adapted to incorporate universal variables, but of course this has to be checked in detail. We also plan to investigate how superposition-based equality handling can be incorporated into hyper tableaux [Bau98].

Another important field for research is building in associativity and commutativity or other common equational theories. We expect that this can be done in the same way as for resolution, see e.g. [NR97].

Finally, it would be (at least theoretically) interesting to find an answer to the question of Sect. 7.4.3, namely whether there is a natural restriction that ensures the termination property but is compatible with regularity.

In the introduction, we mentioned that the context of our work is formal program verification. This means that we are not only interested in fully automated theorem proving, but rather in an integration on automated and interactive deduction. The concepts for such an integration have already been developed [Gie98], and the prover of the KeY project [ABB$^+$00, ABB$^+$02] has reached a state where properties of programs in the JavaCard language can be proven interactively. It remains to build automated proof search for first order goals into the KeY prover, using the incremental closure technique.

9 Conclusion

We have described the *incremental closure* technique, which can be used to define a proof procedure for free variable tableaux without backtracking. The central idea is to incrementally compute instantiations that close sub-tableaux until one global instantiation is found that closes the whole tableau.

The practicality of this approach has been shown by experimentally comparing a prototypical implementation of the technique to a backtracking prover. The results showed that the incremental closure technique significantly shortens the time required for proof search in many cases.

It was shown that the incremental closure technique is compatible with many of the standard refinements known for backtracking proof procedures. We also introduced a family of powerful simplification rules which can be used to subsume some of the most successful techniques employed in state-of-the-art theorem provers.

We also showed how to integrate equality handling into the incremental closure framework, using rules based on ordered rewriting. We presented a method of proving completeness of such calculi which is simpler than previous approaches, and which permits the adaptation of results from resolution saturation procedures.

We hope to have convinced the reader that the incremental closure technique is an interesting alternative to the usual backtracking iterative deepening proof procedures for free variable tableaux, in that it is both efficient and open to refinements and modifications.

Bibliography

[ABB$^+$00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In Manuel Ojeda-Aciego, Inma P. de Guzmán, Gerhard Brewka, and Luís Moniz Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, volume 1919 of *LNCS*, pages 21–36. Springer-Verlag, October 2000.

[ABB$^+$02] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering. 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 2002, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer-Verlag, 2002.

[ABH$^+$98] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97–116. Kluwer, 1998.

[Ahr01] Wolfgang Ahrendt. Deduktive Fehlersuche in Abstrakten Datentypen, 2001. Dissertation (preversion, in German), University of Karlsruhe, available under `http://www.cs.chalmers.se/~ahrendt/cade02/diss.ps.gz`.

[Ahr02] Wolfgang Ahrendt. Deductive search for errors in free data type specifications using model generation. In Andrei Voronkov, editor, *Proc. 18th CADE, Copenhagen, Denmark*, volume 2392 of *LNCS*. Springer-Verlag, to appear 2002.

[And81] Peter B. Andrews. Theorem proving through general matings. *Journal of the ACM*, 28:193–214, 1981.

[Bau98] Peter Baumgartner. Hyper Tableaux — The Next Generation. In Harrie de Swart, editor, *Proc. International Conference on Automated Reasoning* *with Analytic Tableaux and Related Methods, Oosterwijk, The Netherlands*, number 1397 in LNCS, pages 60–76. Springer-Verlag, 1998.

[Bau00] Peter Baumgartner. FDPLL – a First-Order Davis-Putnam-Logemann-Loveland Procedure. In David McAllester, editor, *Automated Deduction, CADE-17*, LNAI. Springer-Verlag, 2000.

[BDP89] Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. Completion without failure. In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 1–30. Academic Press, New York, 1989.

[Bec93] Bernhard Beckert. Ein vervollständigungsbasiertes Verfahren zur Behandlung von Gleichheit im Tableaukalkül mit freien Variablen. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, July 1993.

[Bec94] Bernhard Beckert. A completion-based method for mixed universal and rigid E-unification. In Alan Bundy, editor, *Proc. 12th Conference on Automated Deduction CADE, Nancy/France*, LNAI 814, pages 678–692. Springer-Verlag, 1994.

[Bec98] Bernhard Beckert. *Integration und Uniformierung von Methoden des tableaubasierten Theorembeweisens.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, 1998.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In Isabelle Attali and Thomas P. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.

[Bec03] Bernhard Beckert. Depth-first proof search without backtracking for free-variable clausal tableaux. *Journal of Symbolic Computation*, 36:117–138, 2003.

[BEF99] Peter Baumgartner, Norbert Eisinger, and Ulrich Furbach. A confluent connection calculus. In Harald Ganzinger, editor, *Proc. 16th International Conference on Automated Deduction, CADE-16, Trento, Italy*, volume 1632 of *LNCS*, pages 329–343. Springer-Verlag, 1999.

[BEF00] Peter Baumgartner, Norbert Eisinger, and Ulrich Furbach. A confluent connection calculus. In Steffen Hölldobler, editor, *Intellectics and Computational Logic — Papers in Honor of Wolfgang Bibel*, volume 19 of *Applied Logic Series.* Kluwer, 2000.

[BF95] Matthias Baaz and Christian G. Fermüller. Non-elementary speedups between different versions of tableaux. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *Proc. 4th International Workshop, TABLEAUX'95, St. Goar, Germany*, volume 918 of *LNCS*, pages 217–230. Springer-Verlag, 1995.

[BFN96a] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In José Júlio Alferes, Luís Moniz Pereira, and Ewa Orłowska, editors, *Proc. European Workshop: Logics in Artificial Intelligence, JELIA*, volume 1126 of *LNCS*, pages 1–17. Springer-Verlag, 1996.

[BFN96b] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. Technical Report 8/96, Institute for Computer Science, University of Koblenz, Germany, 1996.

[BG90] Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplication. In Mark E. Stickel, editor, *Proceedings of CADE-10, Kaiserslautern, Germany*, volume 449 of *LNCS*, pages 427–441. Springer-Verlag, 1990.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[BGLS95] Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[BGV97] Leo Bachmair, Harald Ganzinger, and Andrei Voronkov. Elimination of equality via transformation with ordering constraints. Research Report MPI-I-97-2-012, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, December 1997.

[BH98] Bernhard Beckert and Reiner Hähnle. Analytic tableaux. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume I, chapter 1, pages 11–41. Kluwer, 1998.

[BHRM94] Bernhard Beckert, Reiner Hähnle, Anavai Ramesh, and Neil Murray. On anti-links. In Frank Pfenning, editor, *Proc. 5th International Conference on Logic Programming and Automated Reasoning, Kiev, Ukraine*, volume 822 of *LNCS*, pages 275–289. Springer-Verlag, 1994.

[BHS93] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. The *even more* liberalized δ-rule in free variable semantic tableaux. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Proceedings of the third Kurt Gödel Colloquium KGC'93, Brno, Czech Republic*, volume 713 of *LNCS*, pages 108–119. Springer-Verlag, August 1993.

[Bib87] Wolfgang Bibel. *Automated Theorem Proving.* Vieweg, Braunschweig, second revised edition, 1987.

[Bic84] Lubomir Bic. Execution of logic programs on a dataflow architecture. In *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA*, pages 290–296, 1984.

[Bil96] Jean-Paul Billon. The disconnection method: a confluent integration of unification in the analytic framework. In Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors, *Theorem Proving with Tableaux and Related Methods, 5th International Workshop, TABLEAUX'96, Terrasini, Palermo, Italy*, volume 1071 of *LNCS*, pages 110–126. Springer-Verlag, 1996.

[BP94] Bernhard Beckert and Joachim Posegga. lean$T^{A}P$: Lean tableau-based theorem proving. extended abstract. In Alan Bundy, editor, *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France*, volume 814 of *LNCS 814*, pages 793–797. Springer-Verlag, 1994.

[Bra75] D. Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4(4):412–430, 1975.

[BT03] Peter Baumgartner and Cesare Tinelli. The model evolution calculus. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction, CADE-19 (Miami, Florida, USA)*, Lecture Notes in Artificial Intelligence, pages 350–364. Springer, 2003.

[CK85] John S. Conery and Dennis F. Kibler. AND parallelism and nondeterminism in logic programs. *New Generation Computing*, 3(1):43–70, 1985.

[Clo87] William F. Clocksin. Principles of the DelPhi parallel inference machine. *The Computer Journal*, 30(5), 1987.

[Clo92] William F. Clocksin. The DelPhi multiprocessor inference machine. URL: `ftp://clip.dia.fi.upm.es/pub/papers/IJCSLP92-WS6/SessionA/William.Clocksin.paper.ps.Z`, 1992.

[CNA98] Domenico Cantone and Marianna Nicolosi Asmundo. A further and effective liberalization of the delta-rule in free variable semantic tableaux. In Maria Paola Bonacina and Ulrich Furbach, editors, *Second Int. Workshop on First-Order Theorem Proving, FTP'98.* Technische Universität, Wien (Austria), 1998.

[Com91] Hubert Comon. Disunification: a survey. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, chapter 9, pages 322–359. MIT Press, Cambridge, MA, USA, 1991.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[DM94] Marcello D'Agostino and Marco Mondadori. The taming of the cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.

[DV94] Anatoli Degtyarev and Andrei Voronkov. Equality elimination for semantic tableaux. Technical Report 90, Comp. Science Dept., Uppsala University, 1994.

[DV96] Anatoli Degtyarev and Andrei Voronkov. The undecidability of simultaneous rigid E-unification. *Theoretical Computer Science*, 166(1-2):291–300, October 1996.

[DV97] Anatoli Degtyarev and Andrei Voronkov. What you always wanted to know about rigid E-unification. Technical Report 143, Comp. Science Dept., Uppsala University, 1997.

[DV98] Anatoli Degtyarev and Andrei Voronkov. What you always wanted to know about rigid E-unification. *Journal of Automated Reasoning*, 20(1):47–80, 1998.

[DV01] Anatoli Degtyarev and Andrei Voronkov. Equality reasoning in sequent-based calculi. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 10, pages 611–706. Elsevier Science, 2001.

[Fit96] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

[GA99] Martin Giese and Wolfgang Ahrendt. Hilbert's ϵ-terms in automated theorem proving. In Neil V. Murray, editor, *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Saratoga Springs/NY, USA*, number 1617 in LNCS, pages 171–185. Springer-Verlag, 1999.

[GH03] Martin Giese and Reiner Hähnle. Tableaux + constraints. Also as Tech. Report RT-DIA-80-2003, Dipt. di Informatica e Automazione, Università degli Studi di Roma Tre, 2003.

[Gie98] Martin Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diploma Thesis, Fakultät für Informatik, Universität Karlsruhe, June 1998.

[Gie00a] Martin Giese. A first-order simplification rule with constraints. In Peter Baumgartner and Hantao Zhang, editors, *3rd Int. Workshop on First-Order Theorem Proving (FTP), St. Andrews, Scotland, TR 5/2000 Univ. of Koblenz*, pages 113–121, 2000.

[Gie00b] Martin Giese. Proof search without backtracking using instance streams, position paper. In Peter Baumgartner and Hantao Zhang, editors, *3rd Int. Workshop on First-Order Theorem Proving (FTP), St. Andrews, Scotland, TR 5/2000 Univ. of Koblenz*, pages 227–228, 2000.

[Gie01] Martin Giese. Incremental closure of free variable tableaux. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proc. Intl. Joint Conf. on Automated Reasoning, Siena, Italy*, volume 2083 of *LNCS*, pages 545–560. Springer-Verlag, 2001.

[Gie02] Martin Giese. A model generation style completeness proof for constraint tableaux with superposition. In Uwe Egly and Christian G. Fermüller, editors, *Proc. Intl. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, Copenhagen, Denmark*, volume 2381 of *LNCS*, pages 130–144. Springer-Verlag, 2002.

[Gie03] Martin Giese. Simplification rules for constrained formula tableaux. In Marta Cialdea Mayer and Fiora Pirri, editors, *Proc. Intl. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, Rome, Italy*, LNCS, pages 65–80. Springer, 2003.

[GJS97] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1997.

[Gra94] Peter Graf. Substitution tree indexing. Technical Report MPI-I-94-251, Universität Saarbrücken, 1994.

[Gra96] Peter Graf. *Term Indexing*, volume 1053 of *LNCS*. Springer-Verlag, 1996.

[GRS87] Jean H. Gallier, Stan Raatz, and Wayne Snyder. Theorem proving using rigid E-unification: Equational matings. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 338–346. IEEE Computer Society Press, 1987.

[GS89] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67:203–260, 1989.

[Häh01] Reiner Hähnle. Tableaux and related methods. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 3, pages 100–178. Elsevier Science, 2001.

[Hal86] Zahran Halim. A data-driven machine for OR-parallel evaluation of logic programs. *New Generation Computing*, 4(1):5–33, 1986.

[HS94] Reiner Hähnle and Peter H. Schmitt. The liberalized δ-rule in free variable semantic tableaux. *Journal of Automated Reasoning*, 13(2):211–222, October 1994.

[HS03] Reiner Hähnle and Niklas Sörensson. Fair constraint merging tableaux in lazy functional programming style. In Marta Cialdea Mayer and Fiora Pirri, editors, *Proc. Intl. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, Rome, Italy*, LNCS, pages 252–256. Springer, 2003.

[Kan63] Stig Kanger. A simplified proof method for elementary logic. *Computer Programming and Formal Systems*, pages 87–94, 1963. Reprinted as [Kan83].

[Kan83] Stig Kanger. A simplified proof method for elementary logic. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning 1: Classical Papers on Computational Logic 1957–1966*, pages 364–371. Springer-Verlag, Berlin, Heidelberg, 1983.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970. Reprinted as [KB83].

[KB83] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970*, pages 342–376. Springer-Verlag, Berlin, Heidelberg, 1983.

[KJ00] Boris Konev and Tudor Jebelean. Using meta-variables for natural deduction in Theorema. In Michael Kohlhase and Manfred Kerber, editors, *Proceedings of Calculemus-2000 Conference*. Electronic Notes in Computer Science, 2000.

[Kow74] Robert A. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proceedings of 6th IFIP Congress, Stockholm, Sweden*, pages 569–574. North-Holland, 1974.

[Küh97] Michael Kühn. Rigid hypertableaux. In *Proc. of KI '97: Advances in Artificial Intelligence*, volume 1303 of *LNAI*, pages 87–98. Springer-Verlag, 1997.

[KV01] Konstantin Korovin and Andrei Voronkov. Knuth-Bendix constraint solving is NP-complete. In *Proc. Intl. Conf. on Automata, Languages and Programming (ICALP)*, volume 2076 of *LNCS*, pages 979–992, 2001.

[Let99] Reinhold Letz. First-order tableau methods. In Marcello D'Agostino, Dov Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*, pages 125–196. Kluwer, Dordrecht, 1999.

[LS01a] Reinhold Letz and Gernot Stenz. DCTP: A Disconnection Calculus Theorem Prover. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR-2001), Siena, Italy*, volume 2083 of *LNAI*, pages 381–385. Springer, Berlin, June 2001.

[LS01b] Reinhold Letz and Gernot Stenz. Model elimination and connection tableau procedures. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 28, pages 2015–2114. Elsevier Science, 2001.

[LS02] Reinhold Letz and Gernot Stenz. Integration of equality reasoning into the disconnection calculus. In Uwe Egly and Christian G. Fermüller, editors, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-2002), Copenhagen, Denmark*, LNAI, pages 176–190. Springer, Berlin, July 2002.

[LSBB92] Reinhold Letz, Johann Schumann, Stephan Bayerl, and Wolfgang Bibel. SETHEO: A high-perfomance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[Mas97] Fabio Massacci. Simplification with renaming: A general proof technique for tableau and sequent-based provers. Technical Report 424, Computer Laboratory, Univ. of Cambridge (UK), 1997.

[Mas98] Fabio Massacci. Simplification: A general constraint propagation technique for propositional and modal tableaux. In Harrie de Swart, editor, *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Oosterwijk, The Netherlands*, volume 1397 of *LNCS*, pages 217–232. Springer-Verlag, 1998.

[McC97] William McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, December 1997.

[MR93] Neil V. Murray and Erik Rosenthal. Dissolution: Making paths vanish. *Journal of the ACM*, 40(3):504–535, 1993.

[MS97] Max Moser and Joachim Steinbach. STE-modification revisited. AR-Report AR-97-03, Institut für Informatik, München (Germany), 1997.

[NHRV01] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the evaluation of indexing techniques for theorem proving. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *International Joint Conference on Automated Reasoning, Siena, Italy*, volume 2083 of *LNCS*, pages 257–271. Springer-Verlag, 2001.

[NR95a] Robert Nieuwenhuis and Albert Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19(4):321–352, 1995.

[NR95b] Robert Nieuwenhuis and Albert Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19(4):321–351, 1995.

[NR97] Robert Nieuwenhuis and Albert Rubio. Paramodulation with built-in AC-theories and symbolic constraints. *Journal of Symbolic Computation*, 23(1):1–21, January 1997.

[NR99] Robert Nieuwenhuis and José Miguel Rivero. Solved forms for path ordering constraints. In Paliath Narendran and Michaël Rusinowitch, editors, *Proc. 10th International Conference on Rewriting Techniques and Applications (RTA), Trento, Italy*, volume 1631 of *LNCS*, pages 1–15. Springer-Verlag, 1999.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[ON02] Breanndán Ó Nualláin. Constraint tableaux. In *Position Papers presented at International Conference on Analytic Tableaux and Related Methods, Copenhagen, Denmark*, 2002.

[Pel86] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[Pel97] Nicolas Peltier. Simplifying and generalizing formulae in tableaux: pruning the search space and building models. In Didier Galmiche, editor, *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Pont-à-Mousson, France*, volume 1227 of *LNCS*, pages 313–327. Springer-Verlag, 1997.

[Pel99] Nicolas Peltier. Pruning the search space and extracting more models in tableaux. *Logic Journal of the IGPL*, 7(2):217–251, 1999. Available online at http://www3.oup.co.uk/igpl/Volume_07/Issue_02/.

[Pet94] Uwe Petermann. A complete connection calculus with rigid e-unification. In Craig MacNish, David Pearce, and Luís Moniz Pereira, editors, *Logics in Artificial Intelligence(JELIA)*, pages 152–166. Springer-Verlag, Berlin, Heidelberg, 1994.

[PW78] Mike Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Rei92] Wolfgang Reif. The KIV system: Systematic construction of verified software. In D. Kapur, editor, *Proc. 11th Conference on Automated Deduction, Albany/NY*, LNAI 607, pages 753–760. Springer-Verlag, 1992.

[RSV01] I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier Science, 2001.

[Rub94] Albert Rubio. *Automated deduction with ordering and equality constrained clauses.* PhD thesis, Technical University of Catalonia, Barcelona, Spain, 1994.

[RV01] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (System description). In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proc. Intl. Joint Conf. on Automated Reasoning, Siena, Italy*, volume 2083 of *LNCS*, pages 376–380. Springer-Verlag, 2001.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 135–150. Edinburgh University Press, Edinburgh, Scotland, 1969.

[Smu68] Raymond M. Smullyan. *First-Order Logic*, volume 43 of *Ergebnisse der Mathematik und ihrer Grenzgebiete.* Springer-Verlag, New York, 1968.

[SS97] Christian B. Suttner and Geoff Sutcliffe. The TPTP problem library — v2.1.0. Technical Report JCU-CS-97/8, Department of Computer Science, James Cook University, 15 December 1997.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *Proc. Formal Methods in Computer-Aided Design, Second International Conference, FMCAD'98, Palo Alto/CA, USA*, volume 1522 of *LNCS*, pages 82–99, 1998.

[Ste02] Gernot Stenz. DCTP 1.2 – system abstract. In Uwe Egly and Christian G. Fermüller, editors, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-2002), Copenhagen, Denmark*, LNAI, pages 335–339. Springer, Berlin, July 2002.

[Sti86] Mark E. Stickel. Schubert's steamroller problem: Formulations and solutions. *Journal of Automated Reasoning*, 2:89–101, 1986.

[UML01] Object Modeling Group. *Unified Modelling Language Specification, version 1.4*, September 2001.

[vE00] Jan van Eijck. LazyTAP — a lazy tableau theorem prover for FOL. Manuscript, Available online at: `http://www.cwi.nl/~jve/dynamo/papers/lazyTAP.ps.gz`, December 2000.

[vE01] Jan van Eijck. Constrained hyper tableaux. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *LNCS*, pages 232–246. Springer-Verlag, 2001.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[Wel47] B. L. Welch. The generalization of 'Student's' problem when several different populations are involved. *Biometrika*, 34:28–35, 1947.

[Wre90] Karen L. Wrench. *A distributed AND-OR parallel Prolog network.* PhD thesis, University of Cambridge, 1990. Summary as Tech. Report 212, Computer Lab., University of Cambridge.

Index

Curriculum Vitae – Martin Giese

Persönliche Daten	Geboren am	14/06/1970
	in	Berlin (West)
	Familienstand:	verheiratet seit dem 02/02/2002 mit Simone Giese, geb. John
	Nationalität	Deutsch
Schule	09/81 – 07/84	Thomas-Mann-Gymnasium Stutensee
	09/84 – 07/85	Collège Campra, Aix-en-Provence
	09/85 – 07/86	Lycée Cézanne, Aix-en-Provence
	08/86 – 06/90	Thomas-Mann-Gymnasium Stutensee
	16/05/90	Abitur mit Abschlußnote 1,0
Zivildienst	07/90 – 09/91	Zivildienstleistender bei der Firma ISB Bruchsal-Bretten
Studium	10/91 – 06/98	Studium der Informatik an der Universität Karlsruhe (TH)
	07/93	Vordiplom mit Gesamtnote „sehr gut“
	30/06/98	Abschluß der Diplomprüfung zum Thema „Integriertes automatisches und interaktives Beweisen: die Kalkülebene“ mit Gesamtnote 1,0
	08/98 – 08/00	Stipendiat im Graduiertenkolleg „Beherrschbarkeit komplexer Systeme“ der Univ. Karlsruhe (TH)
Auslandsaufenthalte	01/78 – 11/79	Weymouth, England
	07/84 – 07/86	Aix-en-Provence, Frankreich
Wettbewerbe	1990	Bundeswettbewerb Mathematik 1. Preis in der ersten Runde
	1997	ACM Southwestern European Programming Contest, Platz 7
Arbeitsverhältnisse	11/89 – 06/91	Freier Mitarbeiter der Software-Firma Advanced Applications Viczena GmbH
	04/92 – 06/95	Wissenschaftliche Hilfskraft am Institut für Telematik bei Herrn Prof. Dr. Krüger.
	02/95 – 06/98	Wissenschaftliche Hilfskraft am Institut für Logik, Komplexität und Deduktionssysteme bei Herrn Prof. Dr. Menzel
	seit 09/00	Wissenschaftlicher Mitarbeiter am Institut für Logik, Komplexität und Deduktionssysteme bei Herrn Prof. Dr. Menzel