Habilitationsschrift

Tableau and Connection Calculi.

Structure, Complexity, Implementation

Reinhold Letz

December 27, 1999

Contents

Introduction

In the last years considerable progress has been made in the development of tableau-based proof systems for automated deduction. While the tableau framework always was very influential in proof theory and in the *development* of logics, particularly non-classical ones, it had almost no influence on automated deduction in classical logic. This changed about ten years ago, when it was recognized that it is more natural to view automated deduction calculi like model elimination or the connection method as particular refinements of the tableau calculus. The central new feature of those refinements is the active use of connections as a control mechanism for guiding the proof search. In order to emphasize this, the term "connection tableaux" was introduced. This view had a very fruitful effect on the research in the area. In the meantime, many proof systems developed in automated deduction have been reformulated in tableau style. As a positive result of these activities, the similarities and differences between many calculi with formerly unclear relations could be identified. Furthermore, new calculi have been developed which are based on tableaux and integrate connections in different manners. Currently, some of the most powerful theorem proving systems are based on tableaux with connections. So, within the last decade, tableaux have become one of the favorite paradigms for automated deduction in classical logic, too. The crucial feature which accounts for this fact is the active integration of connections. In this work we will attempt to give a comprehensive presentation of the state of the art of tableau and connection calculi for classical first-order logic.

The material is organized in ten chapters. The first chapter provides the general background on first-order logic with function symbols; the syntax and the classical model-theoretic semantics of first-order logic are introduced and the fundamental properties of variable substitutions are described. Furthermore, we mention the most important normal forms of first-order logic and present basic concepts like Herbrand interpretations.

In the second chapter, we turn to the tableau system for first-order logic due to Smullyan and Fitting. Using uniform notation, first, tableau calculi for closed formulae are developed. We prove Hintikka's Lemma and the completeness of first-order tableaux by using a systematic tableau procedure. Furthermore, basic refinements of tableaux are considered like strictness, regularity and the Herbrand condition, and the properties of confluence and nondestructiveness are introduced, which are important for the classification of tableau-based proof systems. Then we turn to the crucial weakness of the traditional tableau systems with respect to proof search. It lies in the nature of the standard γ-rule, which enforces that instantiations have to be chosen too early. The standard approach to remedy this weakness is to permit free variables in a tableau which are treated as placeholders for terms, as so-called "rigid" variables; the instantiation of rigid variables then is guided by unification. Unfortunately, systematic procedures for free-variable tableaux cannot be devised as easily as for sentence tableaux. Therefore, typically, tableau enumeration procedures are used instead. We analyze the consequences of this design decision with respect to the use of tableaux as decision procedures for certain formula classes and concerning the possibility of model generation.

Following the tableau enumeration approach of free-variable tableaux, the crucial demand is to reduce the number of tableaux to be considered by the search procedure. The central such concept in clause logic is the notion of a connection, which can be employed in various ways to guide the tableau construction. In Chapter 3, we incorporate connections into tableaux. From here on, we concentrate on tableaux for formulae in clausal form. Clause logic permits a more condensed representation of tableaux and hence a simplification of the tableau rules. Two connection conditions of increasing restrictiveness are introduced which define the so-called connection tableaux. These calculi are compared with the previous tableau systems and with each other with respect to proof search and the preservation of properties like confluence and nondestructiveness.

In the fourth chapter, we illustrate the relation of connection tableaux with other calculi for automated deduction like model elimination and the connection calculi used in the connection method. Furthermore, we consider alternative approaches of integrating connections into tableaux that are preserving confluence and hence facilitate a limited possibility of model generation. First, we introduce and compare different variants of hyper tableaux. Hyper tableau systems have the general problem that they still need a version of the traditional problematic γ-rule. This rule can only be avoided for a restricted class of formulae. We also present the so-called disconnection method, which does not suffer from this weakness and which represents the first confluent integration of unification into a tableau branch saturation procedure. Both proof systems are nondestructive.

In Chapter 5, a number of refinements of tableau procedures are discussed which drastically reduce the number of permitted deductions and hence increase the suitability of the respective tableau systems for automatic proof search. On the one hand, we consider further restrictions on the structure of tableaux. On the other hand, a significant problem in proof search is that typically certain deductions are redundant in the presence of other ones. These *global* approaches of inter-tableau pruning form the second class of methods for redundancy elimination. This chapter also includes a completeness proof for structurally refined connection tableaux. Since connection tableaux are both nonconfluent and destructive, also a fundamentally different technique for proving completeness has to be used.

In Chapter 6 methods are developed which can produce significantly shorter tableau proofs. The techniques can be subdivided into three different classes. The mechanisms of the first type are centered around controlled integrations of the cut rule; these methods are also related to the use of lemmata. Second, so-called liberalizations of the δ-rule are mentioned, which may lead to even nonelementarily smaller tableau proofs. Finally, we consider an improvement which concerns tableaux with free variables, which typically are considered as rigid. We consider conditions under which free variables may be treated as universally quantified on the respective branch. This can lead to exponentially smaller proofs.

Chapter 7 contains a number of complexity results on the various calculi developed in this work and their relation to other important proof systems. In this chapter, the criterion for comparing calculi is their relation concerning the minimal complexities of proofs that can be generated by the calculi. We present a wealth of so-called *polynomial simulation* results between the considered calculi and with other calculi from automated deduction like resolution and semantic trees.

As a matter of fact, the classification of calculi according to polynomial simulatability must not be the only criterion for comparing calculi, since it completely abstracts from the problem of proof *search*. In Chapter 8, we concentrate on the complexities of search spaces. Since in general the search spaces are infinite, we study the sizes of the finite initial segments defined by the different types of iterative-deepening bounds, the inference, the depth, and the multiplicity bounds. It turns out that, for all three cases, one can obtain completeness results with respect to well-known complexity classes like NP or NEXPTIME.

In Chapter 9, we turn to the implementation of deduction systems based on tableaux with connections. Here, we concentrate on the techniques developed for the non-confluent and destructive connection tableau systems, since those are the most successful tableau-based systems in automated deduction. When considering the implementation of such systems, we have a very special situation. This is because such connection tableaux are very close to SLD-resolution, which is the basic inference system of the programming language Prolog. Consequently, one can exploit this proximity by using as much as possible from the implementation techniques developed for Prolog. One successful such approach is to extend abstract machine technology from the Horn case to the full clausal case. Another possibility consists in taking Prolog itself as a programming language, by which often reasonably efficient implementations of connection tableaux can be obtained with no or only very little implementational effort. Both of these approaches will be described in detail. However, both approaches suffer from some inflexibility problems. Therefore we also consider in detail an implementation approach which is not directly based on Prolog. The key idea for achieving high efficiency in this approach is the extensive re-use of the results of expensive operations.

The development of a redundancy elimination technique is one thing, its efficient implementation is another one. Fortunately, many of the refinements developed for connection tableaux may be formulated in a uniform general setting, as conditions on the instantiations of variables, so-called disequation constraints. In Chapter 10, we develop the general framework of disequation constraints including universal variables and normalization, and describe in detail how efficient constraint handlers may be implemented.

In the Conclusion, we summarize our work and sketch promising future extensions of tableau systems with connections.

Acknowledgements

I would like to thank Gernot Stenz and Andreas Wolf for valuable support and discussions during the completion of this work.

Chapter 1

Preliminaries

1.1 Classical First-Order Logic

The theory of first-order logic is a convenient and powerful formal abstraction from expressions and concepts occurring in natural language, and, most significantly, in mathematical discourse. In this section we present the syntax and semantics of first-order logic with function symbols. Furthermore, the central modification operation on first-order expressions is introduced, the replacement of variables, and some of its invariances are studied.

1.1.1 Syntax of First-Order Logic

Propositional logic deals with sentences and their composition, hence the alphabet of a propositional language consists of only three types of symbols, propositional variables, logical symbols, and punctuation symbols. First-order logic does not stop at the sentence level, it can express the *internal* structure of sentences. In first-order logic, the logical structure and content of assertions of the following form can be studied that have no natural formalization in propositional logic.

Example 1.1 If every person that is not rich has a rich father, then some rich person must have a rich grandfather.

In order to express such formulations, a first-order alphabet has to provide symbols for denoting objects, functions, and relations. Furthermore, it must be possible to make universal or existential assertions, hence we need quantifiers. Altogether, the alphabet or *signature* of a first-order language will be defined as consisting of six disjoint sets of symbols.

Definition 1.2 (First-order signature) A *first-order signature* is a pair $\Sigma = \langle A, a \rangle$ consisting of a denumerably infinite alphabet A and a partial mapping $a\colon A \longrightarrow \mathbb{N}_0$, associating natural numbers with certain symbols in A, called their *arities*, such that A can be partitioned into the following six pairwise disjoint sets of symbols.

1. An infinite set $\mathcal{V}$ of *variables*, without arities.

2. An infinite set of *function symbols*, all with arities such that there are infinitely many function symbols of every arity. Nullary function symbols are called *constants*.

3. An infinite set of *predicate symbols*, all with arities such that there are infinitely many predicate symbols of every arity.

4. A set of *connectives* consisting of five distinct symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$, the first one with arity 1 and all others binary. We call $\neg$ the *negation symbol*, $\wedge$ is the *conjunction symbol*, $\vee$ is the *disjunction symbol*, $\rightarrow$ is the *material implication symbol*, and $\leftrightarrow$ is the *material equivalence symbol*,

5. A set of *quantifiers* consisting of two distinct symbols $\forall$, called the *universal quantifier*, and $\exists$, called the *existential quantifier*, both with arity 2.

6. A set of *punctuation symbols* consisting of three distinct symbols without arities, which we denote with the symbols '(', ')', and ','.

Notation 1.3 Normally, we will denote variables and function symbols with lower-case letters and predicate symbols with upper-case letters. Preferably, we use for variables letters from 'u' onwards; for constants the letters 'a', 'b', 'c', 'd', and 'e'; for function symbols with arity ≥ 1 the letters 'f', 'g' and 'h'; and for predicate symbols the letters 'P', 'Q' and 'R'; nullary predicate symbols shall occasionally be denoted with lower-case letters. Optionally, subscripts will be used. We do not distinguish between symbols and unary strings consisting of symbols, the context will clear up possible ambiguities. We will always talk *about* symbols of first-order languages and never give examples of concrete expressions *within* a specific object language.

Given a first-order signature Σ, the corresponding *first-order language* is defined inductively[1] as a set of specific strings over the alphabet of the signature. In our presentation of first-order languages we use prefix notation for the representation of terms and atomic formulae, and infix notation for the binary connectives. Let in the following $\Sigma = \langle A, a \rangle$ be a fixed first-order signature.

Definition 1.4 (Term)

1. Every variable in A is said to be a *term over* Σ.

2. If f is an n-ary function symbol in A with $n \geq 0$ and $t_1, \ldots, t_n$ are terms over Σ, then the concatenation $f(t_1, \ldots, t_n)$ is a *term over* Σ.

[1] In inductive definitions we shall, conveniently, omit the explicit formulation of the necessity condition.

Definition 1.5 (Atomic formula) If P is (the unary string consisting of) an n-ary predicate symbol in A with $n \geq 0$ and $t_1, \ldots, t_n$ are terms over Σ, then the concatenation $P(t_1, \ldots, t_n)$ is an *atomic formula*, or *atom*, *over* Σ.

Notation 1.6 Terms of the form $a()$ and atoms of the form $P()$ are abbreviated by writing just a and P, respectively.

Definition 1.7 (Formula)

1. Every atom over Σ is a *formula over* Σ.

2. If Φ and Ψ are formulae over Σ and x is (the unary string consisting of) a variable in A, then the following concatenations are also *formulae over* Σ:
   $\neg\Phi$, called the *negation* of Φ,
   $(\Phi \wedge \Psi)$, called the *conjunction* of Φ *and* Ψ,
   $(\Phi \vee \Psi)$, called the *disjunction* of Φ *and* Ψ,
   $(\Phi \rightarrow \Psi)$, called the *material implication* of Ψ *by* Φ,
   $(\Phi \leftrightarrow \Psi)$, called the *material equivalence* of Φ *and* Ψ,
   $\forall x\Phi$, called the *universal quantification* of Φ *in* x, and
   $\exists x\Phi$, called the *existential quantification* of Φ *in* x.

Definition 1.8 ((Well-formed) expression) All terms and formulae over Σ are called *(well-formed) expressions over* Σ.

Definition 1.9 (First-order language) The set of all (well-formed) expressions over Σ is called the *first-order language over* Σ.

Definition 1.10 (Complement) The *complement* of any negated formula $\neg\Phi$ is Φ and the *complement* of any unnegated formula Φ is its negation $\neg\Phi$; we denote the complement of a formula Φ with $\sim\Phi$.

Definition 1.11 (Literal) Every atomic formula and every negation of an atomic formula is called a *literal*.

Recalling the assertion given in Example 1.1: "if every person that is not rich has a rich father, then some rich person must have a rich grandfather," a possible (abstracted) first-order formalization would be the following formula.

Example 1.12 $\forall x(\neg R(x) \rightarrow R(f(x))) \rightarrow \exists x(R(x) \wedge R(f(f(x))))$.

Definition 1.13 (Subexpression) If an expression Φ is the concatenation of strings $W_1, \ldots, W_n$, in concordance with the Definitions 1.4 to 1.7, then any expression among these strings is called an *immediate subexpression* of Φ. The sequence obtained by deleting all elements from $W_1, \ldots, W_n$ that are not expressions is called the *immediate subexpression sequence* of Φ. Among the strings $W_1, \ldots, W_n$ there is a unique string W whose symbol is a connective, a quantifier, a function symbol, or a predicate symbol; W is called the *dominating symbol* of Φ. An expression Ψ is said to be a *subexpression* of an expression Φ if the pair $\langle\Psi, \Phi\rangle$ is in the transitive closure of the immediate subexpression relation. Analogously, the notions of (immediate) subterms and (immediate) subformulae are defined.

Example 1.14 According to our conventions of denoting symbols and strings, a formula of the form $P(x, f(a, y), x)$ has the immediate subexpression sequence $x, f(a, y), x$; the immediate subexpressions x and $f(a, y)$; the subexpressions x, $f(a, y)$, a, and y; and, lastly, P as dominating symbol.

We have to provide a means for addressing different occurrences of symbols and subexpressions in an expression E. One could simply address occurrences by giving the first and last word positions in E. Although this way occurrences of symbols and subexpressions in an expression could be uniquely determined, this notation has the disadvantage that whenever expressions are modified, e.g., by concatenating them or by replacing an occurrence of a subexpression, then the addresses of the occurrences may change completely. We will use a notation which is more robust concerning concatenations of and replacements in expressions. This notation is motivated by a *symbol tree* representation of logical expressions, as displayed in Figure 1.1.

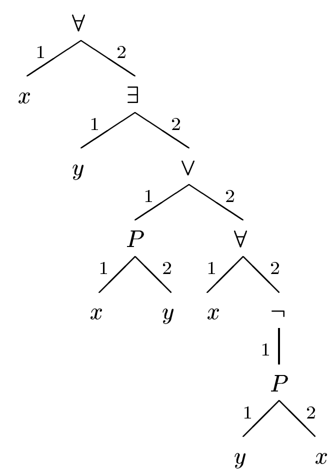


Figure 1.1: Symbol tree of the formula $\forall x \exists y (P(x, y) \lor \forall x \neg P(y, x))$.

Each occurrence of a symbol or a subexpression in an expression can be uniquely determined by a *sequence of natural numbers* that encodes the edges to be followed in the symbol tree. Formally, tree positions can be defined as follows.

Definition 1.15 (Position) For any expression E,

1. if s is the dominating symbol of an expression E, then the *position* both of E and of the dominating occurrence of s *in* E is the empty sequence $\emptyset$.

2. if $E_1, \ldots, E_n$ is the immediate subexpression sequence of E and if $p_1, \ldots, p_n$ is the position of an occurrence of an expression or a symbol W in E_i, $1 \leq i \leq n$, then the *position* of that occurrence of W *in* E is the sequence $i, p_1, \ldots, p_n$.

An occurrence of a symbol or an expression W with position $p_1, \ldots, p_n$ in an expression E is denoted with ${}^{p_1,\ldots,p_n}W$.

For example, the occurrences of the variable x in the formula $\forall x \exists y(P(x,y) \vee \forall x \neg P(y,x))$ are ${}^{1}x$, ${}^{2,2,1,1}x$, ${}^{2,2,2,1}x$, and ${}^{2,2,2,2,1,2}x$. Since it is essential to associate variable occurrences in an expression with occurrences of quantifiers, we need the concept of the scope of a quantifier occurrence.

Definition 1.16 (Scope of a quantifier occurrence) If ${}^{p_1,\ldots,p_m}\mathcal{Q}$ is the occurrence of a quantifier in a formula Φ, then the occurrence of the respective quantification ${}^{p_1,\ldots,p_m}\mathcal{Q}x\Psi$ is called the *scope* of ${}^{p_1,\ldots,p_m}\mathcal{Q}$ *in* Φ; every occurrence of the structure ${}^{p_1,\ldots,p_m,p_{m+1},\ldots,p_n}W$ $(m \leq n)$ of symbols or expressions in Φ is said to be *in the scope* of ${}^{p_1,\ldots,p_m}\mathcal{Q}$ *in* Φ.

Referring to the formula in Figure 1.1, the occurrence ${}^{2,2,1}P(x,y)$ is in the scope of only one quantifier occurrence, namely, ${}^{\emptyset}\forall$, whereas ${}^{2,2,2,2,1}P(y,x)$ is in the scope of both occurrences of the universal quantifier.

Definition 1.17 (Bound and free variable occurrence) If an occurrence of a variable ${}^{p_1,\ldots,p_m,p_{m+1},\ldots,p_n}x$, $m < n$, in an expression Φ is in the scope of a quantifier occurrence ${}^{p_1,\ldots,p_m}\mathcal{Q}$, then that variable occurrence is called a *bound occurrence* of x in Φ; the variable occurrence is said to be *bound by* the rightmost such quantifier occurrence in the string notation of Φ, i.e., by the one with the greatest index $m < n$. A variable occurrence is called *free* in an expression if it is not bound by some quantifier occurrence in the expression.

Accordingly, the rightmost occurrence ${}^{2,2,2,2,1,2}x$ of x in the formula in Figure 1.1 is bound by the universal quantifier at position $2, 2, 2$. Note that every occurrence of a variable in a well-formed expression is bound by at most one quantifier occurrence in the expression.

Definition 1.18 (Closed and ground expression, sentence) If an expression does not contain variables, it is called *ground*, and if it does not contain free variables, it is termed *closed*. Closed formulae are called *sentences*.

Definition 1.19 (Closures of a formula) Let Φ be a formula and $\{x_1, \ldots, x_n\}$ the set of free variables of Φ, then the sentence $\forall x_1 \cdots \forall x_n \Phi$ is called a *universal closure* of Φ, and the sentence $\exists x_1 \cdots \exists x_n \Phi$ is called an *existential closure* of Φ.

Notation 1.20 In order to gain readability, we shall normally spare brackets. As usual, we permit to omit outermost brackets. Furthermore, for arbitrary binary connectives $\circ_1, \circ_2$, any formula of the shape $\Phi \circ_1 (\Psi \circ_2 \Xi)$ may be abbreviated by writing just $\Phi \circ_1 \Psi \circ_2 \Xi$ (right bracketing). Accordingly, if brackets are missing, the dominating infix connective is always the leftmost one.

1.1.2 Semantics of Classical First-Order Logic

Now we are going to present the classical model-theoretic semantics of first-order logic due to [Tarski, 1936]. In contrast to propositional logic, where it is sufficient to work with Boolean valuations and where the atomic formulae can be treated as the basic meaningful units, the richer structure of the first-order language requires a finer analysis. In first-order logic the basic semantic components are the denotations of the terms, a collection of objects termed *universe*.

Definition 1.21 (Universe) Any non-empty collection[2] of objects is called a *universe*.

The function symbols and the predicate symbols of the signature of a first-order language are then interpreted as functions and relations over such a universe.

Notation 1.22 For every universe $\mathcal{U}$, we denote with $\mathcal{U}_\mathcal{F}$ the collection of mappings $\bigcup_{n\in\mathbb{N}_0} \mathcal{U}^n \longrightarrow \mathcal{U}$, and with $\mathcal{U}_\mathcal{P}$ the collection of relations $\bigcup_{n\in\mathbb{N}_0} \mathfrak{P}(\mathcal{U}^n)$ with $\mathfrak{P}(\mathcal{U}^n)$ being the power set of $\mathcal{U}$. Note that any nullary mapping in $\mathcal{U}_\mathcal{F}$ is from the singleton set $\{\emptyset\}$ to $\mathcal{U}$, and hence, subsequently, will be identified with the single element in its image. Any nullary relation in $\mathcal{U}_\mathcal{P}$ is just an element of the two-element set $\{\emptyset, \{\emptyset\}\}$ $(= \{0, 1\}$, according to the Zermelo-Fraenkel definition of natural numbers). We call the sets $\emptyset$ and $\{\emptyset\}$ *truth values*, and abbreviate them with $\bot$ and $\top$, respectively.

This way the mapping of atomic formulae to truth values as performed for the case of propositional logic is captured as a special case by the more general framework developed now. In the following, we denote with $\mathcal{L}$ a first-order language, with $\mathcal{V}$, $\mathcal{F}$, and $\mathcal{P}$ the sets of variables, function symbols, and predicate symbols in the signature of $\mathcal{L}$, respectively, and with $\mathcal{T}$ and $\mathcal{W}$ the sets of terms and formulae in $\mathcal{L}$, respectively.

Definition 1.23 (First-order structure, interpretation) A *(first-order) structure* is a pair $\langle\mathcal{L},\mathcal{U}\rangle$ consisting of a first-order language $\mathcal{L}$ and a universe $\mathcal{U}$. An *interpretation for* a first-order structure $\langle\mathcal{L},\mathcal{U}\rangle$ is a mapping $\mathcal{I}\colon \mathcal{F} \cup \mathcal{P} \longrightarrow \mathcal{U}_\mathcal{F} \cup \mathcal{U}_\mathcal{P}$ such that

1. $\mathcal{I}$ maps every n-ary function symbol in $\mathcal{F}$ to an n-ary function in $\mathcal{U}_\mathcal{F}$.

2. $\mathcal{I}$ maps every n-ary predicate symbol in $\mathcal{P}$ to an n-ary relation in $\mathcal{U}_\mathcal{P}$.

[2]Whenever the term 'collection' will be used, no restriction is made with respect to the cardinality of an aggregation, whereas the term 'set' indicates that only denumerably many elements are contained.

Since formulae may contain free variables, the notion of variable assignments is be needed.

Definition 1.24 (Variable assignment) A *variable assignment from* a first-order language $\mathcal{L}$ *to* a universe $\mathcal{U}$ is a mapping $\mathcal{A}\colon \mathcal{V} \longrightarrow \mathcal{U}$.

Once an interpretation and a variable assignment have been fixed, the meaning of any term and any formula in the language is uniquely determined.

Definition 1.25 (Term assignment) Let $\mathcal{I}$ be an interpretation for a structure $\langle\mathcal{L},\mathcal{U}\rangle$, and let $\mathcal{A}$ be a variable assignment from $\mathcal{L}$ to $\mathcal{U}$. The *term assignment* of $\mathcal{I}$ *and* $\mathcal{A}$ is the mapping $\mathcal{I}^{\mathcal{A}}\colon \mathcal{T} \longrightarrow \mathcal{U}$ defined as follows.

1. For every variable x in $\mathcal{V}$: $\mathcal{I}^{\mathcal{A}}(x) = \mathcal{A}(x)$.

2. If f is a function symbol of arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms, then

$$\mathcal{I}^{\mathcal{A}}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathcal{I}^{\mathcal{A}}(t_1), \ldots, \mathcal{I}^{\mathcal{A}}(t_n)).$$

Finally, we come to the assignment of truth values to formulae, which is defined by simultaneous induction.

Definition 1.26 (Formula assignment) Let $\mathcal{I}$ be an interpretation for a structure $\langle\mathcal{L},\mathcal{U}\rangle$, and let $\mathcal{A}$ be a variable assignment from $\mathcal{L}$ to $\mathcal{U}$. The *formula assignment* of $\mathcal{I}$ *and* $\mathcal{A}$ is the mapping $\mathcal{I}^{\mathcal{A}}\colon \mathcal{W} \longrightarrow \{\top, \bot\}$ defined as follows. Let Φ and Ψ denote arbitrary formulae of $\mathcal{L}$.

1. For any nullary predicate symbol P in the signature of $\mathcal{L}$: $\mathcal{I}^{\mathcal{A}}(P) = \mathcal{I}(P)$.

2. If P is a predicate symbol of arity $n > 0$ and $t_1, \ldots, t_n$ are terms, then

$$\mathcal{I}^{\mathcal{A}}(P(t_1, \ldots, t_n)) = \begin{cases} \top & \text{if } \langle\mathcal{I}^{\mathcal{A}}(t_1), \ldots, \mathcal{I}^{\mathcal{A}}(t_n)\rangle \in \mathcal{I}(P) \\ \bot & \text{otherwise.} \end{cases}$$

3. $$\mathcal{I}^{\mathcal{A}}((\Phi \vee \Psi)) = \begin{cases} \top & \text{if } \mathcal{I}^{\mathcal{A}}(\Phi) = \top \quad \text{or} \quad \mathcal{I}^{\mathcal{A}}(\Psi) = \top \\ \bot & \text{otherwise.} \end{cases}$$

4. $$\mathcal{I}^{\mathcal{A}}(\neg\Phi) = \begin{cases} \top & \text{if } \mathcal{I}^{\mathcal{A}}(\Phi) = \bot \\ \bot & \text{otherwise.} \end{cases}$$

5. $$\mathcal{I}^{\mathcal{A}}((\Phi \wedge \Psi)) = \mathcal{I}^{\mathcal{A}}(\neg(\neg\Phi \vee \neg\Psi)).$$

6. $$\mathcal{I}^{\mathcal{A}}((\Phi \rightarrow \Psi)) = \mathcal{I}^{\mathcal{A}}((\neg\Phi \vee \Psi)).$$

7. $$\mathcal{I}^{\mathcal{A}}((\Phi \leftrightarrow \Psi)) = \mathcal{I}^{\mathcal{A}}(((\Phi \rightarrow \Psi) \wedge (\Psi \rightarrow \Phi))).$$

8. A variable assignment is called an *x-variant* of a variable assignment if both assignments differ at most in the value of the variable x.

$$\mathcal{I}^{\mathcal{A}}(\forall x \Phi) = \begin{cases} \top & \text{if } \mathcal{I}^{\mathcal{A}'}(\Phi) = \top \text{ for all } x\text{-variants } \mathcal{A}' \text{ of } \mathcal{A} \\ \bot & \text{otherwise.} \end{cases}$$

9. $$\mathcal{I}^{\mathcal{A}}(\exists x \Phi) = \mathcal{I}^{\mathcal{A}}(\neg \forall x \neg \Phi).$$

We extend the definition to sets S of formulae by setting $\mathcal{I}^{\mathcal{A}}(S) = \top$ if and only if $\mathcal{I}^{\mathcal{A}}(\Phi) = \top$, for all formulae $\Phi \in S$.

Particularly interesting is the case of interpretations for sentences, i.e., closed formulae. From the definition of formula assignments (items 8 and 9) it follows that, for any sentence and any interpretation $\mathcal{I}$, the respective formula assignments are all identical, and hence do not depend on the variable assignments. Consequently, for sentences, we shall speak of *the* formula assignment of an interpretation $\mathcal{I}$, and write it $\mathcal{I}$, too. Possible ambiguities between an interpretation and the corresponding formula assignment will be clarified by the context.

To comprehend the manner in which formula assignments give meaning to expressions, see Example 1.27. The example illustrates how formulae are interpreted in which an occurrence of a variable is in the scopes of different quantifier occurrences. Loosely speaking, Definition 1.26 guarantees that variable assignments obey "dynamic binding" rules (in terms of programming languages), in the sense that a variable assignment to a variable x for an expression Φ is *overwritten* by a variable assignment to the same variable x in a subexpression of Φ.

Example 1.27 Two sentences $\Phi = \forall x(\exists x P(x) \wedge Q(x))$ and $\Psi = \forall x \exists x(P(x) \wedge Q(x))$. Given a universe $\mathcal{U} = \{u_1, u_2\}$, let an interpretation $\mathcal{I}(P) = \mathcal{I}(Q) = \{u_1\}$. Then $\mathcal{I}(\Phi) = \bot$ and $\mathcal{I}(\Psi) = \top$.

The central semantic notion is that of a model.

Definition 1.28 (Model) Let $\mathcal{I}$ be an interpretation for a structure $\langle \mathcal{L}, \mathcal{U} \rangle$, A a collection of variable assignments from $\mathcal{L}$ to $\mathcal{U}$, and Φ a first-order formula. We say that $\mathcal{I}$ is an *A-model for* Φ if $\mathcal{I}^{\mathcal{A}}(\Phi) = \top$, for every variable assignment $\mathcal{A} \in A$; if $\mathcal{I}$ is an A-model for Φ and A is the collection of all variable assignments, then $\mathcal{I}$ is called a *model for* Φ. If $\mathcal{I}$ is an (A-)model for every formula in a set of first-order formulae S, then we also call $\mathcal{I}$ an *(A-)model* for S.

The notions of satisfiability and validity abstract from the consideration of specific models.

Definition 1.29 (Satisfiability, validity) Let Γ be a (set of) formula(e) of a first-order language $\mathcal{L}$ (and A a collection of variable assignments). The set Γ is called *(A-)satisfiable* if there exists an (A-)model for Γ. We call Γ *valid* if every interpretation is a model for Γ.

Definition 1.30 (Implication, equivalence) Let Γ and Δ be two (sets of) first-order formulae.

1. We say that Γ *implies* Δ, written $\Gamma \models \Delta$, if every model for Γ is a model for Δ; obviously, if $\Gamma = \emptyset$, then Δ is valid, and we simply write $\models \Delta$.

2. Γ *strongly implies* Δ if, for every universe $\mathcal{U}$ and every variable assignment $\mathcal{A}$ from $\mathcal{L}$ to $\mathcal{U}$: every $\{\mathcal{A}\}$-model for Γ is an $\{\mathcal{A}\}$-model for Δ.

If Γ and Δ (strongly) imply each other, they are called *(strongly) equivalent.*

Note that according to this definition any first-order formula is equivalent to any-one of its universal closures. Obviously, for (sets of) *sentences*, implication and strong implication coincide. Furthermore, the notion of material (object-level) implication and the strong (meta-level) implication concept of first-order formulae are related as follows.

Theorem 1.31 (Implication Theorem) Given two first-order formulae Φ and Ψ, Φ strongly implies Ψ if and only if the formula $\Phi \rightarrow \Psi$ is valid.

Proof For the "if"-part, assume $\Phi \rightarrow \Psi$ be valid. Let $\mathcal{A}$ be any variable assignment and $\mathcal{I}$ an arbitrary $\{\mathcal{A}\}$-model for Φ. By Definition 1.26, $\mathcal{I}^{\mathcal{A}}(\Phi) = \bot$ or $\mathcal{I}^{\mathcal{A}}(\Psi) = \top$. By assumption, $\mathcal{I}^{\mathcal{A}}(\Phi) = \top$; hence $\mathcal{I}^{\mathcal{A}}(\Psi) = \top$, and $\mathcal{I}$ is an $\{\mathcal{A}\}$-model for Ψ. For the "only-if"-part, suppose that Φ strongly implies Ψ. Let $\mathcal{A}$ be any variable assignment and $\mathcal{I}$ an arbitrary interpretation. Now, either $\mathcal{I}^{\mathcal{A}}(\Phi) = \bot$; then, by Definition 1.26, $\mathcal{I}^{\mathcal{A}}(\Phi \rightarrow \Psi) = \top$. Or, $\mathcal{I}^{\mathcal{A}}(\Phi) = \top$; in this case, by assumption, $\mathcal{I}(\Psi) = \top$; hence $\mathcal{I}^{\mathcal{A}}(\Phi \rightarrow \Psi) = \top$. Consequently, in either case $\mathcal{I}$ is an $\{\mathcal{A}\}$-model for $\Phi \rightarrow \Psi$. $\square$

It is obvious that strongly equivalent formulae can be substituted for each other in any context without changing the meaning of the context.

Lemma 1.32 (Replacement Lemma) Given two strongly equivalent formulae F and G and any formula Φ with F as subformula, if the formula Ψ can be obtained from Φ by replacing an occurrence of F in Φ with G, then Φ and Ψ are strongly equivalent.

Another more subtle useful replacement property is the following.

Lemma 1.33 If $\models F \rightarrow G$, then $\models \forall x F \rightarrow \forall x G$.

Proof Assume $\models F \rightarrow G$. Let $\mathcal{I}$ be any interpretation and $\mathcal{A}$ any variable assignment with $\mathcal{I}^{\mathcal{A}}(\forall x F) = \top$. Then, for all x-variants $\mathcal{A}'$ of $\mathcal{A}$: $\mathcal{I}^{\mathcal{A}'}(F) = \top$ and, by assumption, $\mathcal{I}^{\mathcal{A}'}(G) = \top$. Consequently, $\mathcal{I}^{\mathcal{A}}(\forall x G) = \top$. $\square$

The subsequently listed basic strong equivalences between first-order formulae can also be demonstrated easily.

Proposition 1.34 Let F, G, and H be arbitrary first-order formulae. All formulae of the following structures are valid.

(1) $\neg\neg F \leftrightarrow F$

(2) $\neg(F \wedge G) \leftrightarrow (\neg F \vee \neg G)$ *(De Morgan law for $\wedge$)*

(3) $\neg(F \vee G) \leftrightarrow (\neg F \wedge \neg G)$ *(De Morgan law for $\vee$)*

(4) $(F \vee (G \wedge H)) \leftrightarrow ((F \vee G) \wedge (F \vee H))$ *($\vee$-distributivity)*

(5) $(F \wedge (G \vee H)) \leftrightarrow ((F \wedge G) \vee (F \wedge H))$ *($\wedge$-distributivity)*

(6) $\neg\exists x F \leftrightarrow \forall x \neg F$ *($\exists\forall$-conversion)*

(7) $\neg\forall x F \leftrightarrow \exists x \neg F$ *($\forall\exists$-conversion)*

(8) $\forall x(F \wedge G) \leftrightarrow (\forall x F \wedge \forall x G)$ *($\forall\wedge$-distributivity)*

(9) $\exists x(F \vee G) \leftrightarrow (\exists x F \vee \exists x G)$ *($\exists\vee$-distributivity)*

We conclude this part with proving a technically useful property of variable assignments.

Definition 1.35 Two variable assignments $\mathcal{A}$ and $\mathcal{B}$ are said to *overlap on* a set of variables V if for all $x \in V : \mathcal{A}(x) = \mathcal{B}(x)$.

Notation 1.36 For any mapping[3] f, its modification by changing the value of x to u, i.e., $(f \setminus \{\langle x, f(x)\rangle\}) \cup \{\langle x, u\rangle\}$, will be denoted with f^x_u.

Proposition 1.37 Let Φ be a formula of a first-order language $\mathcal{L}$ with V being the set of free variables in Φ, and $\mathcal{U}$ a universe. Then, for any two variable assignments $\mathcal{A}$ and $\mathcal{B}$ from $\mathcal{L}$ to $\mathcal{U}$ that overlap on V:

(1) $\mathcal{I}^{\mathcal{A}}(\Phi) = \mathcal{I}^{\mathcal{B}}(\Phi)$, *and*

(2) if $\Phi = \exists x \Psi$, *then* $\{u \in \mathcal{U} \mid \mathcal{I}^{\mathcal{A}^x_u}(\Psi) = \top\} = \{u \in \mathcal{U} \mid \mathcal{I}^{\mathcal{B}^x_u}(\Psi) = \top\}$.

Proof (1) is obvious from Definition 1.26 of formula assignments. For (2), consider an arbitrary element $u \in \mathcal{U}$ with $\mathcal{I}^{\mathcal{A}^x_u}(\Psi) = \top$. Since $\mathcal{A}^x_u$ and $\mathcal{B}^x_u$ overlap on $V \cup \{x\}$, by (1), $\mathcal{I}^{\mathcal{B}^x_u}(\Psi) = \top$, which proves the set inclusion in one direction. The reverse direction holds by symmetry. $\square$

[3] Mappings are considered as sets of ordered pairs.

1.1.3 Variable Substitutions

The concept of variable substitutions, which we shall introduce next, is the basic modification operation performed on logical expressions. Let in the following denote $\mathcal{T}$ the set of terms and $\mathcal{V}$ the set of variables of a first-order language.

Definition 1.38 ((Variable) substitution) A *(variable) substitution* is any mapping $\sigma \colon V \longrightarrow \mathcal{T}$ where V is a finite subset of $\mathcal{V}$ and $x \neq \sigma(x)$, for every x in the domain of σ. A substitution is called *ground* if no variables occur in the terms of its range.

Definition 1.39 (Binding) Any element $\langle x, t\rangle$ of a substitution, abbreviated x/t, is called a *binding*. We say that a binding x/t is *proper* if the variable x does not occur in the term t.

Now, we consider the application of substitutions to logical expressions.

Definition 1.40 (Instance) If Φ is any expression and σ is a substitution, then the *σ-instance of* Φ, written $\Phi\sigma$, is the expression obtained from Φ by simultaneously replacing every occurrence of each variable $x \in \text{domain}(\sigma)$ that is free in Φ by the term $\sigma(x)$. If Φ and Ψ are expressions and there is a substitution σ with $\Psi = \Phi\sigma$, then Ψ is called an *instance of* Φ. Similarly, if S is a (collection of) set(s) of formulae, then $S\sigma$ denotes the (collection of the) set(s) of σ-instances of its elements.

As a matter of fact, bound variable occurrences are not replaced. Furthermore, we are interested in substitutions which preserve the models of a formula. In order to preserve modelhood for arbitrary logical expressions, the following property is sufficient.

Definition 1.41 (Free substitution) A substitution σ is said to be *free for* an expression Φ provided, for every free occurrence ${}^{s}x$ of a variable in Φ, all variable occurrences in ${}^{s}x\sigma$ are free in $\Phi\sigma$.

While no bound variable occurrence can vanish when a substitution is applied, for free substitutions, no additional bound variable occurrences are imported. This means that the following proposition holds.

Proposition 1.42 A substitution σ is free for an expression Φ if and only if any variable occurs bound at the same positions in Φ and in $\Phi\sigma$.

Bringing in additional bound variables can lead to unsoundness, as shown with the following example.

Example 1.43 Consider a formula Φ of the form $\exists x(P(x,y,z) \land \neg P(y,y,z))$ and the substitutions $\sigma_1 = \{y/z\}$ and $\sigma_2 = \{y/x\}$. While σ_1 is free for Φ and $\Phi \models \Phi\sigma_1 = \exists x(P(x,z,z) \land \neg P(z,z,z))$, σ_2 is not, and indeed $\Phi \not\models \Phi\sigma_2 = \exists x(P(x,x,z) \land \neg P(x,x,z))$.

The following fundamental result relates substitutions and interpretations.

Notation 1.44 If $\mathcal{I}$ is an interpretation, $\mathcal{A}$ a variable assignment, and $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ a substitution, then the variable assignment $\mathcal{A}^{x_1}_{\mathcal{I}^{\mathcal{A}}(t_1)} \cdots {}^{x_n}_{\mathcal{I}^{\mathcal{A}}(t_n)}$ (using Notation 1.36) will be denoted with $\mathcal{A}\sigma_{\mathcal{I}}$. If the underlying interpretation is clear from the context, we will sometimes omit the subscript and simply write $\mathcal{A}\sigma$.

Lemma 1.45 If σ is a substitution that is free for a first-order expression E, then, for any interpretation $\mathcal{I}$ and any variable assignment $\mathcal{A}$: $\mathcal{I}^{\mathcal{A}}(E\sigma) = \mathcal{I}^{\mathcal{A}\sigma}(E)$.

Proof The proof is by induction on the structural complexity of the expression E. First, for any term, the result is immediate from the Definition 1.25 of term assignments. The cases of quantifier-free formula are also straightforward from items (1) – (7) of the Definition 1.26 of formula assignments. We consider the case of a universal formula $\forall x F$ in more detail. $\mathcal{I}^{\mathcal{A}}(\forall x F\sigma) = \top$ if and only if (by item (8) of formula assignment) for all x-variants $\mathcal{A}'$ of $\mathcal{A}$: $\top = \mathcal{I}^{\mathcal{A}'}(F\sigma) =$ (by the induction hypothesis) $\mathcal{I}^{\mathcal{A}'\sigma}(F)$ iff (since σ is assumed to be free for F) for all x-variants $\mathcal{A}\sigma'$ of $\mathcal{A}\sigma$: $\mathcal{I}^{\mathcal{A}\sigma'}(F) = \top$ iff (by item (8) of formula assignment) $\mathcal{I}^{\mathcal{A}\sigma}(\forall x F) = \top$. The existential case is similar. □

Now we can state a very general soundness result for the application of substitutions to logical expressions. Let $\mathcal{V}$ denote the set of variables of the underlying first-order language.

Proposition 1.46 (Substitution soundness) Given a first-order formula Φ and a substitution $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ that is free for Φ, let $V \subseteq \mathcal{V}$ be any set of variables containing $x_1, \ldots, x_n$ and A any collection of all variable assignments that overlap on $\mathcal{V} \setminus V$. If an interpretation $\mathcal{I}$ is an A-model for Φ, then $\mathcal{I}$ is an A-model for $\Phi\sigma$.

Proof Consider an arbitrary variable assignment $\mathcal{A} \in A$. Now, by assumption, A contains all variable assignments that overlap on $\mathcal{V} \setminus V$ where V is any superset of $\{x_1, \ldots, x_n\}$. Therefore, $\mathcal{A}\sigma \in A$ and hence $\mathcal{I}^{\mathcal{A}}(\Phi\sigma) = \top$. Since σ is free for Φ, Lemma 1.45 can be applied which yields that $\mathcal{I}^{\mathcal{A}\sigma}(\Phi) = \mathcal{I}^{\mathcal{A}}(\Phi\sigma) = \top$. □

As a special instance of this proposition we obtain the following corollary (simply set $V = \mathcal{V}$ and A will be the collection of all variable assignments).

Corollary 1.47 For any formula Φ and any substitution σ which is free for Φ : $\Phi \models \Phi\sigma$.

Definition 1.48 (Composition of substitutions) Assume σ and τ to be substitutions. Let σ' be the substitution obtained from the set $\{\langle x, t\tau\rangle \mid x/t \in \sigma\}$ by removing all pairs for which $x = t\tau$, and let τ' be that subset of τ which contains no binding x/t with $x \in \text{domain}(\sigma)$. The substitution $\sigma' \cup \tau'$, written $\sigma\tau$, is called the *composition* of σ and τ.

Proposition 1.49 Let σ, τ and θ be arbitrary substitutions and Φ any logical expression such that σ is free for Φ.

1. *$\sigma\emptyset = \emptyset\sigma = \sigma$, for the empty substitution $\emptyset$.*
2. *$(\Phi\sigma)\tau = \Phi(\sigma\tau)$.*
3. *$(\sigma\tau)\theta = \sigma(\tau\theta)$.*

Proof (1) is immediate. For (2) consider any free occurrence ${}^s x$ of a variable x in Φ. We distinguish three cases. First, $x \notin \text{domain}(\sigma)$ and $x \notin \text{domain}(\tau)$; then ${}^s(x\sigma)\tau = {}^s x = {}^s x(\sigma\tau)$. If, secondly, $x \notin \text{domain}(\sigma)$ but $x \in \text{domain}(\tau)$, then ${}^s(x\sigma)\tau = {}^s x\tau = {}^s x(\sigma\tau)$. Lastly, assume $x \in \text{domain}(\sigma)$; as σ was assumed free for Φ, no variable occurrence in ${}^s x\sigma$ is bound in $\Phi\sigma$, therefore ${}^s(x\sigma)\tau = {}^s x(\sigma\tau)$. Since ${}^s x$ was chosen arbitrary and only free variable occurrences were modified, we have the result for Φ. For (3) let x be any variable. The repeated application of (2) yields that $x((\sigma\tau)\theta) = (x(\sigma\tau))\theta = ((x\sigma)\tau)\theta = (x\sigma)(\tau\theta) = x(\sigma(\tau\theta))$. This means that the substitutions $(\sigma\tau)\theta$ and $\sigma(\tau\theta)$ map every variable to the same term, hence they are identical. □

Summarizing these results, we have that $\emptyset$ acts as a left and right identity for composition; (2) expresses that under the given assumption substitution application and composition permute; and (3), the associativity of substitution composition, permits to omit parentheses when writing a composition of substitutions. As a consequence of (1) and (3), the set of substitutions with the composition operation forms a semi-group.

1.2 Normal Forms and Normal Form Transformations

A *logical problem* for a first-order language consists in the task of determining whether a relation holds between certain first-order expressions. For an *efficient solution* of a logical problem, it is very important to know whether it is possible to restrict attention to a proper sublanguage of the first-order language. This is because certain sublanguages permit the application of more efficient solution techniques than available for the full first-order format. For classical logic, this can be strongly exploited by using prenex and Skolem forms.

1.2.1 Prenex and Skolem Formulae

Definition 1.50 (Prenex form) A first-order formulae Φ is said to be a *prenex formula* or in *prenex form* if it has the structure $\mathcal{Q}_1 x_1 \cdots \mathcal{Q}_n x_n F$, $n \geq 0$, where the $\mathcal{Q}_i$, $1 \leq i \leq n$, are quantifiers and F is quantifier-free. We call F the *matrix* of Φ.

Proposition 1.51 For every first-order formula Φ there is a formula in prenex form which is strongly equivalent to Φ.

Proof We give a constructive method to transform any formula Φ over the connectives $\neg$, $\wedge$, and $\vee$ into prenex form—by the definition of formula assignment, the connectives $\leftrightarrow$ and $\rightarrow$ can be eliminated before, without affecting strong equivalence. If $\mathcal{Q}$ is any quantifier, $\forall$ or $\exists$, with $\bar{\mathcal{Q}}$ we denote the quantifier $\exists$ respectively $\forall$. Now, for any formula which is not in prenex form, one of the following two cases holds.

1. Φ has a subformula of the structure $\neg\mathcal{Q}xF$; then, by Proposition 1.34(6) and (7), and the Replacement Lemma (Lemma 1.32), the formula Ψ obtained from Φ by substituting all occurrences of $\neg\mathcal{Q}xF$ in Φ by $\bar{\mathcal{Q}}x\neg F$ is strongly equivalent to Φ.

2. Φ has a subformula Ψ of the structure $(\mathcal{Q}xF \circ G)$ or $(G \circ \mathcal{Q}xF)$ where $\circ$ is $\wedge$ or $\vee$; let y be a variable not occurring in Φ, then, clearly Ψ and $\Psi' = \mathcal{Q}y(F\{x/y\} \circ G)$ or $\mathcal{Q}y(G \circ F\{x/y\})$, respectively, are strongly equivalent; therefore, by the Replacement Lemma, the formula obtained by replacing all occurrences of Ψ in Φ by Ψ' is strongly equivalent to Φ.

Consequently, in either case one can move quantifiers in front without affecting strong equivalence, and after finitely many iterations prenex form is achieved. $\square$

It is obvious that, except for the case of formulae containing $\leftrightarrow$, the time needed for carrying out this procedure is bounded by a polynomial in the size of the input, and the resulting prenex formula has less than double the size of the initial formula. The removal of $\leftrightarrow$, however, can lead to an exponential increase of the formula size (see [Reckhow, 1976]).

Definition 1.52 (Skolem form) A first-order formula Φ is said to be a *Skolem formula* or in *Skolem form* if it has the form $\forall x_1 \cdots \forall x_n F$ and F is quantifier-free.

The possibility of transforming any first-order formula into Skolem form is fundamental for the field of automated deduction. This is because the removal of existential quantifiers facilitates a particularly efficient computational treatment of first-order formulae. Furthermore, the single step in which an existential quantifier is removed, occurs as a basic component of *any* calculus for full first-order logic.

Definition 1.53 (Skolemization) Let S be a set of formulae containing a formula Φ with the structure $\forall y_1 \cdots \forall y_m \exists y F$, $m \geq 0$, and $x_1, \ldots, x_n$ the variables that are free in $\exists y F$. If f is an n-ary function symbol not occurring in any formula of S (we say that f is *new to* S), then the formula $\forall y_1 \cdots \forall y_m (F\{y/f(x_1, \ldots, x_n)\})$ is named a *Skolemization* of Φ *wrt.* S.

We have introduced a general form of Skolemization which is applicable to arbitrary, not necessarily closed, sets of first-order formulae in prenex form. This is necessary for the free-variable tableau systems developed in Section 4. When moving to a Skolemization of a formula, for any variable assignment $\mathcal{A}$, the collection of $\{\mathcal{A}\}$-models does not increase.

Proposition 1.54 Given a formula Φ of a first-order language $\mathcal{L}$, if Ψ is a Skolemization of Φ wrt. a set of formulae S, then Ψ strongly implies Φ.

Proof Let $\Phi = \forall y_1 \cdots \forall y_m \exists y F$. First, we show that, for *any* term t, the subformula $\exists y F$ of Φ is strongly implied by $F' = F\{y/t\}$. Let $\mathcal{A}$ be any variable assignment and $\mathcal{I}$ any $\{\mathcal{A}\}$-model for F'. By Lemma 1.45, $\mathcal{I}^{\mathcal{A}\{y/t\}}(F) = \top$. Since $\mathcal{A}\{y/t\}$ is an y-variant of $\mathcal{A}$, by the definition of formula assignments, $\mathcal{I}^{\mathcal{A}}(\exists y F) = \top$. Then, a repeated application of Lemma 1.33 yields that $\Psi = \forall y_1 \cdots \forall y_m F'$ strongly implies Φ. □

When moving to a Skolemization of a formula, the collection of models may decrease, however. Consequently, for the transformation of formulae into Skolem form, equivalence must be sacrificed, but the preservation of A-satisfiability can be guaranteed, for any collection A of variable assignments.

Proposition 1.55 Let Ψ be a Skolemization of a formula Φ wrt. a set of formulae S and A any collection of variable assignments. If S is A-satisfiable, then $S \cup \{\Psi\}$ is A-satisfiable.

Proof By assumption, $\Phi \in S$ has the structure $\forall y_1 \cdots \forall y_m \exists y F$ ($m \geq 0$); Ψ has the form $\forall y_1 \cdots \forall y_m F'$ where $F' = F\{y/f(x_1, \ldots, x_n)\}$; f is an n-ary function symbol new to S; and $x_1, \ldots, x_n$ are the free variables in $\exists y F$. Now let A be any collection of variable assignments that has an A-model $\mathcal{I}$ for S. $\mathcal{I}$ need not be an A-model for Ψ, but we show that with a modification of merely the meaning of the function symbol f an A-model for $S \cup \{\Psi\}$ can be specified. First, we define a total and disjoint partition P on the collection of variable assignments A by grouping together all elements in A that overlap on the variables $x_1, \ldots, x_n$. By Proposition 1.37, for any two variable assignments $\mathcal{B}$ and $\mathcal{C}$ in any element of P: $\{u \in \mathcal{U} \mid \mathcal{I}^{\mathcal{B}^y_u}(F) = \top\} = \{u \in \mathcal{U} \mid \mathcal{I}^{\mathcal{C}^y_u}(F) = \top\}$, i.e., for any element of the partition P, the collection of objects "with the property" F is unique; we abbreviate with $\mathcal{U}_{u_1,\ldots,u_n}(F)$ the collection of objects determined by the variable assignments that map $x_1, \ldots, x_n$ to the objects $u_1, \ldots, u_n$, respectively. By the assumption of $\mathcal{I}$ being an A-model for Φ, none of these collections of objects is empty. In order to be able to identify elements in those possibly nondenumerable collections, which is necessary to define a mapping, we have to assume the existence of a well-ordering[4] $\prec$ on $\mathcal{U}$. For any collection $M \subseteq \mathcal{U}$, let μM denote the smallest element modulo $\prec$. Now we can define a total n-ary mapping f: $\mathcal{U}^n \longrightarrow \mathcal{U}$ by setting $\mathsf{f}(u_1, \ldots, u_n) = \mu \mathcal{U}_{u_1,\ldots,u_n}(F)$ and an interpretation $\mathcal{I}_\Psi = \mathcal{I}^f_{\mathsf{f}}$ (using Notation 1.36). Since the symbol f does not occur in any formula of S,

[4] A total ordering $\prec$ on a collection of objects S is a *well-ordering on S* if every non-empty subcollection M of objects from S has a smallest element modulo $\prec$. Note that supposing the existence of a well-ordering amounts to assuming the *axiom of choice* (for further equivalent formulations of the axiom of choice, consult, for example, [Krivine, 1971]).

$\mathcal{I}_\Psi$ is an A-model for S. To realize that $\mathcal{I}_\Psi$ is an A-model for Ψ, too, consider an arbitrary variable assignment $\mathcal{A} \in A$. Clearly, $\mathcal{I}_\Psi^\mathcal{A}(\exists y F) = \top$. Let $P_\mathcal{A}$ be the element of the partition P that contains $\mathcal{A}$. Define the variable assignment $\mathcal{A}' = \mathcal{A}^y_{\mathsf{f}(\mathcal{A}(x_1),\ldots,\mathcal{A}(x_n))}$. $\mathcal{I}_\Psi^{\mathcal{A}'}(F) = \top$ and hence $\mathcal{I}_\Psi^{\mathcal{A}'}(F\{y/f(x_1,\ldots,x_n)\}) = \top$. Now $\mathcal{A}$ and $\mathcal{A}'$ are identical except for the value of y, but y does not occur free in $F\{y/f(x_1,\ldots,x_n)\}$, therefore, $\mathcal{I}_\Psi^\mathcal{A}(F\{y/f(x_1,\ldots,x_n)\}) = \top$. □

Theorem 1.56 (Skolemization Theorem) Given a formula Φ of a first-order language $\mathcal{L}$, let Ψ be a Skolemization of Φ wrt. a set of formulae S and A any collection of variable assignments. S is A-satisfiable if and only if $S \cup \{\Psi\}$ is A-satisfiable.

Proof Immediate from the Propositions 1.54 and 1.55. □

Concerning the space and time complexity involved in a transformation into Skolem form, the following estimate can be formulated.

Proposition 1.57 Given a prenex formula Φ of a first-order language $\mathcal{L}$, if Ψ is a Skolem formula obtained from Φ via a sequence of Skolemizations, then $\mathrm{size}(\Psi)$, *i.e., the length of the string Ψ, is smaller than* $\mathrm{size}(\Phi)^2$, *and the run time of the Skolemization procedure is polynomially bounded by the size of Φ.*

Proof Every variable occurrence in Φ is bound by exactly one quantifier occurrence in Φ, and every variable occurrence in an inserted Skolem term is bound by a universal quantifier. This entails that, throughout the sequence of Skolemization steps, whenever a variable occurrence is replaced by a Skolem term, then no variable occurrence *within* an inserted Skolem term is substituted afterwards. Moreover, the arity of each inserted Skolem function is bounded by the number of free variables in Φ plus the number of variables in the quantifier prefix of Φ. Therefore, the output size is quadratically bounded by the input size. Since in the Skolemization operation merely variable replacements are performed, any deterministic execution of the Skolemization procedure can be done in polynomial time. □

Prenexing and Skolemization only work for classical logic, but not for intuitionistic or other logics. In those cases, more sophisticated methods are needed to encode the nesting of the connectives and quantifiers. Some of those are considered in [Wallen, 1989], [Ohlbach, 1991].

1.2.2 Herbrand Interpretations

The standard theorem proving procedures are based on the following obvious proposition.

Proposition 1.58 Given a set of sentences Γ and a sentence F, $\Gamma \models F$ if and only if $\Gamma \cup \{\neg F\}$ is unsatisfiable.

Accordingly, the problem of determining whether a sentence is logically implied by a set of sentences can be reformulated as an unsatisfiability problem. Demonstrating the unsatisfiability of a set of formulae of a first-order language $\mathcal{L}$, however, means to prove, for any universe $\mathcal{U}$, that no interpretation for $\langle\mathcal{L},\mathcal{U}\rangle$ is a model for the set of formulae. A further fundamental result for the efficient computational treatment of first-order logic is that, for formulae in Skolem form, it is sufficient to examine only the interpretations for one particular domain, the *Herbrand universe* of the set of formulae. Subsequently, let $\mathcal{L}$ denote a first-order language and $a_{\mathcal{L}}$ a fixed constant in the signature of $\mathcal{L}$.

Definition 1.59 (Herbrand universe) Let S be (a set of) formula(e) of $\mathcal{L}$. With S_C we denote the set of constants occurring in (formulae of) S. The *constant base* of S is S_C if S_C is non-empty, and the singleton set $\{a_{\mathcal{L}}\}$ if $S_C = \emptyset$. The *function base* S_F of S is the set of function symbols occurring in (formulae of) S with arities > 0. Then the *Herbrand universe* of S is the set of terms defined inductively as follows.

1. Every element of the constant base of S is in the *Herbrand universe* of S.
2. If $t_1, \ldots, t_n$ are in the Herbrand universe of S and f is an n-ary function symbol in the function base of S, then the term $f(t_1, \ldots, t_n)$ is in the *Herbrand universe* of S.

Definition 1.60 (Herbrand interpretation) Given a (set of) formula(e) S of a first-order language $\mathcal{L}$ with Herbrand universe $\mathcal{U}$. A *Herbrand interpretation for S* is an interpretation $\mathcal{I}$ for the pair $\langle\mathcal{L},\mathcal{U}\rangle$ meeting the following properties.

1. $\mathcal{I}$ maps every constant in S_C to itself.
2. $\mathcal{I}$ maps every function symbol f in S_F with arity $n > 0$ to the n-ary function that maps every n-tuple of terms $\langle t_1, \ldots, t_n\rangle \in \mathcal{U}^n$ to the term $f(t_1, \ldots, t_n)$.

Proposition 1.61 For any (set of) first-order formulae S in Skolem form, if S has a model, then it has a Herbrand model.

Proof Let $\mathcal{I}'$ be an interpretation with arbitrary universe $\mathcal{U}'$ which is a model for S, and let $\mathcal{U}$ denote the Herbrand universe of S. First, we define a mapping $h\colon \mathcal{U} \longrightarrow \mathcal{U}'$, as follows.

1. For every constant $c \in \mathcal{U}$: $h(c) = \mathcal{I}'(c)$.
2. For every term $f(t_1, \ldots, t_n) \in \mathcal{U}$:
   $h(f(t_1, \ldots, t_n)) = \mathcal{I}'(f)(h(t_1), \ldots, h(t_n))$.

Next, we define a Herbrand interpretation $\mathcal{I}$ for S.

3. For every n-ary predicate symbol P, $n \geq 0$, and any n-tuple of objects $\langle t_1, \ldots, t_n\rangle \in \mathcal{U}^n$: $\langle t_1, \ldots, t_n\rangle \in \mathcal{I}(P)$ if and only if $\langle h(t_1), \ldots, h(t_n)\rangle \in \mathcal{I}'(P)$.

Now let $\mathcal{A}$ be an arbitrary variable assignment $\mathcal{A}$ from $\mathcal{L}$ to $\mathcal{U}$. With $\mathcal{A}'$ we denote the functional composition of $\mathcal{A}$ and h. It can be verified easily by induction on the construction of formulae that $\mathfrak{I}'^{\mathcal{A}'}(S) = \top$ entails $\mathfrak{I}^{\mathcal{A}}(S) = \top$. The induction base is evident from the definition of $\mathcal{I}$, and the induction step follows from Definition 1.26. Consequently, $\mathcal{I}$ is a model for S. □

For formulae in Skolem form, the Herbrand universe is always rich enough to be used as a *representative* for any other universe, and the question whether a model exists can always be solved by restricting attention to Herbrand interpretations. For formulae that are not in Skolem form, this does not work, as illustrated with the following simple example.

Example 1.62 The formula $\exists x(P(x) \land \neg P(a))$ is satisfiable, but it has no Herbrand model.

The fact that Herbrand interpretations are sufficient for characterizing modelhood in the case of Skolem formulae can be used for proving the *Löwenheim-Skolem theorem.*

Theorem 1.63 (Löwenheim-Skolem theorem) Every satisfiable (set of) first-order formula(e) S has a model with a countable universe.

Proof Given any satisfiable (set of) first-order formula(e) S, let S' be a (set of) first-order formula(e) obtained from S by prenexing and Skolemization.[5] By Propositions 1.51 and 1.55, S' must be satisfiable, too. Then, by Proposition 1.61, there exists a Herbrand model $\mathcal{I}$ for S' with a countable Herbrand universe, since obviously every Herbrand universe is countable. By Propositions 1.51 and 1.54, $\mathcal{I}$ must be a model for S. □

Working with Herbrand interpretations has the advantage that interpretations can be represented in a very elegant manner.

Definition 1.64 (Herbrand base) Given a (set of) formula(e) S of a first-order language $\mathcal{L}$ with Herbrand universe $\mathcal{U}$. The *predicate base* S_P of S is the set of predicate symbols occurring in (formulae of) S. The *Herbrand base* of S, written $\mathcal{B}_S$, is the set of all atomic formulae $P(t_1, \ldots, t_n)$, $n \geq 0$, with $P \in S_P$ and $t_i \in \mathcal{U}$, for every $1 \leq i \leq n$.

Notation 1.65 Every Herbrand interpretation $\mathcal{H}$ of a (set of) formula(e) S can be uniquely represented by a subset H of the Herbrand base $\mathcal{B}_S$ of S by defining

$$\mathcal{H}(L) = \begin{cases} \top & \text{if } L \in H \\ \bot & \text{otherwise} \end{cases}$$

for any ground atom $L \in \mathcal{B}_S$. We shall exploit this fact and occasionally use subsets of the Herbrand base for denoting Herbrand interpretations.

[5] If S is an infinite set of formulae and the Herbrand universe of S already contains almost all function symbols of a certain arity, then it may be necessary to move to an extended first-order language $\mathcal{L}'$ that contains enough function symbols of every arity.

1.2.3 Formulae in Clausal Form

After prenexing and Skolemizing a formula, it is a standard technique in automated deduction to transform the resulting formula into *clausal form.*

Definition 1.66 (Clause) Any formula c of the form $\forall x_1 \cdots \forall x_n(L_1 \vee \cdots \vee L_m)$, with $m \geq 1$ and the L_i being literals, is a *clause.* Each literal L_i is said to be *(contained) in c.*

Definition 1.67 (Clausal form)

1. Any clause is *in clausal form.*
2. If F is in clausal form and c is a clause, then $c \wedge F$ is *in clausal form.*

Proposition 1.68 For any first-order formula Φ in Skolem form there exists a strongly equivalent formula Ψ in clausal form.

Proof Let F be the matrix of a first-order formula Φ in Skolem form. We perform the following four equivalence preserving operations. First, by items 7 and 6 of Definition 1.26 of formula assignment, successively, the connectives $\leftrightarrow$ and $\rightarrow$ are removed. Secondly, the negation signs are pushed immediately before atomic formulae, using recursively Proposition 1.34(1) and de Morgan's laws (2) and (3). Finally, apply $\vee$-distributivity from left to right until no conjunction is dominated by a disjunction. The resulting formula is in clausal form. □

It can easily be verified that, even for matrices not containing $\leftrightarrow$, the given transformation may lead to an exponential increase of the formula size. In fact, there exists no equivalence preserving polynomial transformation of a matrix into clausal form, even if $\leftrightarrow$ does not occur in the matrix (see [Reckhow, 1976]). But there are polynomial transformations if logical equivalence is sacrificed (see [Eder, 1985, Plaisted and Greenbaum, 1986, Boy de la Tour, 1990]). Those transformations are satisfiability and unsatisfiability preserving, and the transformed formula logically implies the source formula, so that the typical logical problems—unsatisfiability detection or model finding if possible—can be solved by considering the transformed formula.

Chapter 2

Tableau Systems

2.1 First-Order Sentence Tableaux

The *tableau method* was introduced by Beth in [Beth, 1955, Beth, 1959] and elaborated by Hintikka in [Hintikka, 1955] and others, but the most influential standard format was given by Smullyan in [Smullyan, 1968] (cf. Section 2 in the first chapter of this book). Therefore, the tableau calculus for closed first-order formulae, which is developed in this section, will be essentially Smullyan's.

2.1.1 Quantifier Elimination in Unifying Notation

The tableau method for propositional logic exploits the fact that all propositional formulae that are not literals can be partitioned into two syntactic types, a *conjunctive type*, called the *α-type*, or a *disjunctive type*, named the *β-type*. Accordingly, only two inference rules are needed, the α- and the β-rule. This uniformity extends to the first-order language, in that just two more syntactic types are needed to capture all first-order formulae, the *universal type*, called the *γ-type*, and the *existential type*, named the *δ-type*. Likewise, just two further inference rules will be needed, called the γ- and the δ-rule.

Altogether, this results in the following classification and decomposition schema for first-order *sentences*—arbitrary first-order formulae containing free variables will be treated in the next section. To any first-order sentence F of any *connective* type (α or β) a *sequence* of sentences different from F will be assigned, called the α- or *β-subformulae sequence* of F, respectively, as defined in Table 2.1, for all formulae over the connectives $\neg$, $\vee$, $\wedge$ and $\rightarrow$. Note that, by exploitation of the associativity of the connectives $\vee$ and $\wedge$, we permit subformulae sequences of more than two formulae. This straightforward generalization speeds up the decomposition of formulae.

While the subformula sequence and hence the decomposition of any formula of a connective type is always finite, this cannot be achieved in general. A first-order sentence F of any *quantification* type (γ or δ) has possibly infinitely many

Conjunctive		Disjunctive	
α	α-subformulae sequence	β	β-subformulae sequence
$\neg\neg F$	F		
$F_1 \wedge \cdots \wedge F_n$	$F_1, \ldots, F_n$	$F_1 \vee \cdots \vee F_n$	$F_1, \ldots, F_n$
$\neg(F_1 \vee \cdots \vee F_n)$	$\neg F_1, \ldots, \neg F_n$	$\neg(F_1 \wedge \cdots \wedge F_n)$	$\neg F_1, \ldots, \neg F_n$
$\neg(F \to G)$	$F, \neg G$	$F \to G$	$\neg F, G$

Table 2.1: Connective types and α-, β-subformulae sequences.

γ- or δ*-subformulae*, respectively, as defined in Table 2.2 where t ranges over the set of ground terms and c over the set of constants of a first-order language.

Universal		Existential	
γ	γ-subformulae	δ	δ-subformulae
$\forall xF$	$F\{x/t\}$	$\exists xF$	$F\{x/c\}$
$\neg\exists xF$	$\neg F\{x/t\}$	$\neg\forall xF$	$\neg F\{x/c\}$

Table 2.2: Quantification types and γ-, δ-subformulae.

Definition 2.1 Any α-, β-, γ-, or δ-subformula F' of a formula F is named an *immediate tableau subformula* of F. A formula F' is called a *tableau subformula* of F if the pair $\langle F', F\rangle$ is in the transitive closure of the immediate tableau subformula relation.

Obviously, the decomposition schema guarantees that all tableau subformulae of a sentence are sentences. Moreover, the decomposition rules have the following fundamental proof-theoretic property.

Definition 2.2 (Formula complexity) The *formula complexity* of a formula F is the number of occurrences of formulae in F.

Proposition 2.3 Every tableau subformula of a formula F has a smaller formula complexity than F.

This assures that there can be no infinite decomposition sequences.

Notation 2.4 We shall often use suggestive meta-symbols for naming formulae of a certain type. Thus, a formula of the α- or β-type will be denoted with 'α' or 'β', and the formulae in its subformula sequence with 'α_1',…,'α_n' or 'β_1',…,'β_n', respectively; for a formula of the γ- or δ-type and its subformula wrt a term t, we will write 'γ' or 'δ' and '$\gamma(t)$' or '$\delta(t)$', respectively.

As in the propositional case, first-order tableaux are particular *formula trees*, i.e, ordered trees with the nodes labelled with formulae. We do not formally introduce trees, and we permit trees to be infinite. Trees will be viewed as *downwardly* growing from the root. As the *depth* of a tree or a tableau we take the maximal number of edges on a branch. Furthermore, the following abbreviations will be used.

Definition 2.5 If a formula is the label of a node on a branch B of a formula tree T, we say that F *appears* or *is on* B and *in* T. With $B \oplus F_1 \mid \cdots \mid F_n$ we mean the result of attaching $n > 0$ new successor nodes $N_1, \ldots, N_n$, in this order, fanning out of the end of B and labelled with the formulae $F_1, \ldots, F_n$, respectively. Any such sequence $N_1, \ldots, N_n$ is termed a *node family* in T. We shall often treat the branch B of a formula tree as the set of formulae appearing on B. All nodes above a node N on a branch are called its *ancestors*, the ancestor immediately above N is termed its *predecessor*. If two nodes in a tableau are labelled with complementary formulae, we shall also call the nodes *complementary*.

Based on the developed formula decomposition schema, first-order tableaux for *sentences* are defined inductively.

Definition 2.6 (Sentence tableau) (inductive) Let S be any set of sentences of a first-order language $\mathcal{L}$.

- Every one-node formula tree labelled with a formula from S is a *sentence tableau for* S.
- If B is a branch of a sentence tableau T for S, then the formula trees obtained from the following five expansion rules are all *sentence tableaux for* S:
  - ($\boldsymbol{\alpha}$) $B \oplus \alpha_i$, if α_i is an α-subformula of a formula α on B,[1]
  - ($\boldsymbol{\beta}$) $B \oplus \beta_1 \mid \cdots \mid \beta_n$, if $\beta_1, \ldots, \beta_n$ is the β-subformula sequence of a formula β on B,
  - ($\boldsymbol{\gamma}$) $B \oplus \gamma(t)$, if t is a ground term and a formula γ occurs on B,
  - ($\boldsymbol{\delta}$) $B \oplus \delta(c)$, if c is a constant new to S and to the formulae in T, and a formula δ appears on B,
  - (F) $B \oplus F$, for any formula $F \in S$.

If T is a sentence tableau for a singleton set $\{\Phi\}$, we also say that T is a sentence tableau for the *formula* Φ. When a decomposition rule is performed on a formula F at a node N, we say that F or N *is used.*

Obviously, if the input set contains just one formula, the *formula rule* denoted with (F) can be omitted. The inference rules of sentence tableaux for formulae are summarized in Figure 2.1.

[1] Note that this rule is slightly more flexible than the standard α-rule as presented in [Smullyan, 1968] according to which B has to be modified to $B \oplus \alpha_1 \oplus \cdots \oplus \alpha_n$ in a single inference step. The need for this will become clear at the end of the section when we introduce the regularity refinement.

$$\frac{\alpha}{\alpha_i} \qquad \frac{\beta}{\beta_1 \mid \cdots \mid \beta_n} \qquad \frac{\gamma}{\gamma(t)} \qquad \frac{\delta}{\delta(c)}$$

for any ground term t (rule γ); for any new constant c (rule δ)

Figure 2.1: Inference rules of sentence tableaux for formulae.

Definition 2.7 (Closed tableau) A branch B of a tableau is called *(atomically) closed* if an (atomic) formula F and its negation appear on B, otherwise the branch is termed *(atomically) open*. Similarly, a node N is called *(atomically) closed* if all branches through N are (atomically) closed, and *(atomically) open* otherwise. Finally, a tableau is termed *(atomically) closed* if its root node is (atomically) closed, otherwise the tableau is called *(atomically) open*.

$$(1)\ \neg\exists x(\forall y\forall z P(y,f(x,y,z)) \rightarrow (\forall y P(y,f(x,y,x)) \wedge \forall y\exists z P(g(y),z)))$$

$\boldsymbol{\gamma}(1)$

$$(2)\ \neg(\forall y\forall z P(y,f(a,y,z)) \rightarrow (\forall y P(y,f(a,y,a)) \wedge \forall y\exists z P(g(y),z)))$$

$\boldsymbol{\alpha}(2)$

$$(3)\ \forall y\forall z P(y,f(a,y,z))$$

$\boldsymbol{\alpha}(2)$

$$(4)\ \neg(\forall y P(y,f(a,y,a)) \wedge \forall y\exists z P(g(y),z))$$

$\boldsymbol{\beta}(4)$

Left branch:

$(5)\ \neg\forall y P(y,f(a,y,a))$

$\boldsymbol{\delta}(5)$

$(7)\ \neg P(b,f(a,b,a))$

$\boldsymbol{\gamma}(3)$

$(8)\ \forall z P(b,f(a,b,z))$

$\boldsymbol{\gamma}(8)$

$(9)\ P(b,f(a,b,a))$

Right branch:

$(6)\ \neg\forall y\exists z P(g(y),z)$

$\boldsymbol{\delta}(6)$

$(10)\ \neg\exists z P(g(b),z)$

$\boldsymbol{\gamma}(10)$

$(11)\ \neg P(g(b),f(a,g(b),a))$

$\boldsymbol{\gamma}(3)$

$(12)\ \forall z P(g(b),f(a,g(b),z))$

$\boldsymbol{\gamma}(12)$

$(13)\ P(g(b),f(a,g(b),a))$

Figure 2.2: An atomically closed sentence tableau.

In Figure 2.2, a larger sentence tableau for a first-order sentence is displayed that illustrates the application of each tableau rule. For every tableau expansion step, the respective type of tableau expansion rule and the used ancestor node are annotated at the connecting vertices. Note that all branches of the tableau are atomically closed. A closed sentence tableau for a set of sentences S represents a correct proof of the unsatisfiability of S. The correctness of the tableau approach as a proof method for first-order sentences is based on the fact that the decomposition rules are satisfiability preserving.

Proposition 2.8 Let S be any satisfiable set of first-order sentences.

(1) If $\alpha \in S$, then $S \cup \{\alpha_i\}$ is satisfiable, for every α-subformula α_i of α.

(2) If $\beta \in S$, then $S \cup \{\beta_i\}$ is satisfiable, for some β-subformula β_i of β.

(3) If $\gamma \in S$, then $S \cup \{\gamma(t)\}$ is satisfiable, for any ground term t.

(4) If $\delta \in S$, then $S \cup \{\delta(c)\}$ is satisfiable, for any constant c that is new to S.

Proof Items (1) and (2) are immediate from the definition of formula assignment; (3) is a consequence of the soundness of substitution application (Proposition 1.46); lastly, since δ is assumed as closed and c is new to S, $\delta(c)$ is a Skolemization of δ wrt S, hence (4) follows from Proposition 1.55. □

Proposition 2.9 (Soundness of sentence tableaux) *If a set of sentences S is satisfiable, then every sentence tableau for S has an open branch.*

Proof We use the following notation. A branch of a tableau for a set of formulae S is called *satisfiable* if $S \cup B$ is satisfiable where B is the set of formulae on the branch. Clearly, every satisfiable branch must be open. We prove, by induction on the number of tableau expansion steps, that every sentence tableau for a satisfiable set of sentences S has a satisfiable branch. The induction base is evident. For the induction step, consider any tableau T for S generated with $n+1$ expansion steps. Let T' be a tableau for S from which T can be obtained by a single expansion step. By the induction assumption, T' has a satisfiable branch B. Now, either T contains B, in which case T' has a satisfiable branch. Or B is expanded; in this case, Proposition 2.8 guarantees that one of the new branches in T' is satisfiable. □

A fundamental proof-theoretic advantage of the tableau method over *synthetic* proof systems like axiomatic calculi [Hilbert and Ackermann, 1928] is the *analyticity* of the decomposition rules. The formulae in a tableaux are in the reflexive-transitive closure of the tableau subformula relation on the input set. For certain formula classes, this permits the generation of decision procedures based on tableaux, which will be discussed below.

2.1.2 Completeness of Sentence Tableaux

First-order logic differs from propositional logic in that there are no decision procedures for the logical status of a set of formulae, but merely *semi-decision* procedures. More precisely, there exist effective mechanical methods for verifying the unsatisfiability of sets of first-order formulae (or the logical validity of first-order formulae[2]), whereas, when subscribing to *Church's Thesis*, the satisfiability of sets of first-order formulae (or the non-validity of first-order formulae) cannot be effectively recognized.[3]

The tableau calculus represents such an effective mechanical proof method. In this part, we will provide a completeness proof of sentence tableaux. An essential concept used in this proof is that of a *downward saturated* set of sentences.

Definition 2.10 (Downward saturated set) Let S be a set of first-order sentences and $\mathcal{U}$ the Herbrand universe of S. The set S is called *downward saturated* provided:

1. if S contains an α, then it contains all its α-subformulae,
2. if S contains a β, then it contains at least one of its β-subformulae,
3. if S contains a γ, then it contains all $\gamma(t)$ with $t \in \mathcal{U}$,
4. if S contains a δ, then it contains a $\delta(c)$ with c being a constant in $\mathcal{U}$.

Definition 2.11 (Hintikka set) By an *(atomic) Hintikka set* we mean a downward saturated set which does not contain an (atomic) formula and its negation.

Lemma 2.12 (Hintikka's Lemma) Every atomic Hintikka set (and hence every Hintikka set) is satisfiable.

Proof Let S be an atomic Hintikka set. We show that some Herbrand interpretation for S is a model for S. Let H denote the set of ground atoms in S, which defines a Herbrand interpretation $\mathcal{H}$, using Notation 1.65. We prove, by induction on the formula complexity, that $\mathcal{H}$ is a model for all formulae in S. The induction base is evident. For the induction step, assume that $\mathcal{H}(F) = \top$, for all formulae F in S with formula complexity $< n$. First, since S does not contain an atomic formula and its negation, $\mathcal{H}$ is a model for all literals in S. Now consider any non-literal formula $F \in S$ with formula complexity n. The formula complexity of every tableau subformula of F is $< n$.

1. If F is an α, then, by the definition of downward saturation, every α_i is in S. Since, by the induction assumption, $\mathcal{H}(\alpha_i) = \top$, $\mathcal{H}(F) = \top$.
2. If F is a β, then, again by the definition of downward saturation, some β_i is in S. By the induction assumption, $\mathcal{H}(\beta_i) = \top$, therefore, $\mathcal{H}(F) = \top$.
3. If F is a $\gamma = \forall x F'$, by the downward saturatedness of S and the induction assumption, $\mathcal{H}(\gamma(t)) = \top$, for any term t in the Herbrand universe $\mathcal{U}$ of S. Since $\mathcal{U}$ is the universe of $\mathcal{H}$ and since $\mathcal{H}$ (being a Herbrand interpretation) maps every term t to itself, for all variable assignments $\mathcal{A}$ to $\mathcal{U}$, $\mathcal{H}^{\mathcal{A}}(F') = \mathcal{H}(F'\{x/\mathcal{A}(x)\}) = \top$. Therefore, $\mathcal{H}(F) = \top$.

[2] This result was first demonstrated by Gödel in [Gödel, 1930].

[3] Thus settling the *undecidability* of first-order logic, which was proved by Church in [Church, 1936] and Turing in [Turing, 1936].

4. Finally, if F is a δ, by downward saturation and the induction assumption $\mathcal{H}(\delta(c)) = \top$, for some constant $c \in \mathcal{T}$, therefore $\mathcal{H}(F) = \top$. □

After these preliminaries, we can come back to tableaux. The tableau calculus is indeterministic, i.e., many possible expansion steps are possible in a certain situation. We are now going to demonstrate that the tableau construction can be made completely deterministic and yet it can be guaranteed that the tableau will eventually close if the set of input formulae is unsatisfiable. Such tableaux are called *systematic tableaux*. In order to make the expansion deterministic, we have to determine,

1. from where the next formula has to be taken, and
2. for the case of the quantifier rules, to which closed term the respective variable has to be instantiated.

Furthermore, since systematic tableaux shall be introduced for the most general case in which the set of input formulae may be infinite, we have to provide means for making sure that any formula in the set will be taken into account in the tableau construction, if necessary.

For the node selection, we equip the nodes of tableaux with an additional number label, expressing whether the formula at the node can be used for a tableau expansion step or not. If a node carries a number label, then the formula at the node will be a possible candidate for a tableau expansion step, otherwise not.

Definition 2.13 (Usable node) If a tableau T contains nodes with number labels, then from all the nodes labelled with the smallest number the leftmost one with minimal tableau depth is called *the usable node* of T; otherwise T has *no usable node*.

For the term selection needed in the quantifier rules, we employ a total ordering on the set of closed terms. Both selection functions together can be used to uniquely determine the next tableau expansion steps. Concerning the fairness problem in case the set of input formulae be infinite, we use an additional total ordering on the formulae.

Definition 2.14 (Systematic tableau (sequence)) (inductive) Let π be a mapping from $\mathbb{N}_0$ onto the set of ground terms and $\prec$ a total ordering on the set of formulae of a first-order language $\mathcal{L}$, respectively, and S any set of closed first-order formulae. The *systematic tableau sequence* of S *wrt* π *and* $\prec$ is the following sequence $\mathcal{T}$ of tableaux for S. Let Φ be the smallest formula in S modulo $\prec$.

- The one-node tableau T_0 with root formula Φ and number label 0 is the first element of $\mathcal{T}$.

- If T_n is the n-th element in $\mathcal{T}$ and has nodes with number labels, let N be the usable node of T_n with formula F and number k. Furthermore, if some formula in S is not on some branch passing through N, let G denote the smallest such formula modulo $\prec$. Now expand each open branch B passing through N to:

  1. B [$\oplus G$] $\oplus \alpha_1 \oplus \cdots \oplus \alpha_n$, if F is of type α with α-subformula sequence $\alpha_1, \ldots, \alpha_n$,
  2. B [$\oplus G$] $\oplus \beta_1 \mid \cdots \mid \beta_n$, if F is of type β with β-subformula sequence $\beta_1, \ldots, \beta_n$,
  3. B [$\oplus G$] $\oplus \gamma(\pi(k))$, if F is of type γ,
  4. B [$\oplus G$] $\oplus \delta(c)$ if F is of type δ and c is the smallest constant modulo π not occurring in T.

  Then give every newly attached node the number label 0 if its formula label is not a literal. Next, remove the number labels from all nodes that have become atomically closed through the expansion steps. Finally, if F is not of type γ, remove the number label from N; otherwise change the number k at N to $k+1$. The tableau resulting from the entire operation is the $n+1$-st element of the sequence $\mathcal{T}$.

- If T_n has no usable node, it is the last element of $\mathcal{T}$.

In Figure 2.3, a closed systematic tableau is shown with $\pi(0) = a$ and $\pi(1) = b$. The following structural property of sentence tableaux plays an important role. We formulate it generically, for any system of tableau inference rules.

Definition 2.15 (Nondestructiveness) A tableau calculus C is called *nondestructive* if, whenever a tableau T can be deduced from a tableau T' according to the inference rules of C, then T' is an initial segment of T; otherwise C is called destructive.

Since obviously the calculus of sentence tableaux is nondestructive, one can form the (tree) union of all the tableaux in a systematic tableau sequence.

Definition 2.16 (Saturated systematic tableau) Let $\mathcal{T}$ be a systematic tableau sequence for a set of first-order formulae S. With T^* we denote the smallest formula tree containing all tableaux in $\mathcal{T}$ as initial segments; T^* is called a *saturated systematic tableau* of S.

Proposition 2.17 For any (atomically) open branch B of a saturated systematic tableau, the set of formulae on B is a(n atomic) Hintikka set.

Proof Let B be any (atomically) open branch of a saturated systematic tableau. According to the definition of systematic tableau, it is guaranteed that the branch B satisfies the following condition: for any formula F on B,

$$(1)\ \exists y \exists z \forall x (P(x,y) \land (P(z,x) \to \neg P(y,y)))$$

$$\Big|\ \boldsymbol{\delta}(1)$$

$$(2)\ \exists z \forall x (P(x,a) \land (P(z,x) \to \neg P(a,a)))$$

$$\Big|\ \boldsymbol{\delta}(2)$$

$$(3)\ \forall x (P(x,a) \land (P(b,x) \to \neg P(a,a)))$$

$$\Big|\ \boldsymbol{\gamma}(3)$$

$$(4)\ (P(a,a) \land (P(b,a) \to \neg P(a,a)))$$

$$\Big|\ \boldsymbol{\alpha}(4)$$

$$(5)\ P(a,a)$$

$$\Big|\ \boldsymbol{\alpha}(4)$$

$$(6)\ (P(b,a) \to \neg P(a,a))$$

$$\boldsymbol{\beta}(6)$$

$$(7)\ \neg P(b,a) \qquad\qquad (8)\ \neg P(a,a)$$

$$\Big|\ \boldsymbol{\gamma}(3)$$

$$(9)\ (P(b,a) \land (P(b,b) \to \neg P(a,a)))$$

$$\Big|\ \boldsymbol{\alpha}(9)$$

$$(10)\ P(b,a)$$

$$\Big|\ \boldsymbol{\alpha}(9)$$

$$(11)\ (P(b,b) \to \neg P(a,a))$$

Figure 2.3: An atomically closed systematic tableau.

1. if F is of type α, then all α-subformulae of F must be on B.
2. if F is of type β, then some β-subformula of F must be on B.
3. if F is of type γ, then all γ-subformulae of F must be on B.
4. if F is of type δ, then some δ-subformulae of F must be on B.

So the set S of formulae on B is downward saturated. Since B is (atomically) open, no (atomic) formula and its negation are in S, hence S is a(n atomic) Hintikka set. □

Now the refutational completeness of tableaux is straightforward.

Theorem 2.18 (Completeness of sentence tableau) If S is an unsatisfiable set of first-order sentences, then there exists a finite atomically closed sentence tableau for S.

Proof Let T be a saturated systematic tableau for S. First, we show that T must be atomically closed. Assume, indirectly, that T contains an atomically open branch B. Then, by Proposition 2.17, there would exist an atomic Hintikka set for the set S' of formulae on B and, by Hintikka's Lemma, a model $\mathcal{I}$ for S'. Now, by the definition of saturated systematic tableau, $S \subseteq S'$, hence $\mathcal{I}$ would be a model for S, contradicting the unsatisfiability assumption. This proves that every branch of T must be atomically closed. In order to recognize the finiteness of T, note that the closedness of any branch of a systematic tableau entails that it cannot have a branch of infinite length. Since the branching rate of any tableau is finite, (by König's Lemma) T must be finite. $\square$

The generality of our systematic tableau procedure permits an easy proof of a further fundamental property of first-order logic.

Theorem 2.19 (Compactness Theorem) Any unsatisfiable set of first-order sentences has a finite unsatisfiable subset.

Proof Let S be any unsatisfiable set of first-order sentences. By Theorem 2.18, there exists a finite closed tableau T for S. Let S' be the set of formulae in S appearing in T. S' is finite and, by the soundness of tableaux, S' must be unsatisfiable. $\square$

Sentence tableaux can also be used to illustrate the basic Herbrand-type property of first-order logic that with any unsatisfiable set of prenex formulae one can associate unsatisfiable sets of ground formulae as follows.

Definition 2.20 A tableau is called *quantifier preferring* if on any branch all applications of quantifier rules precede applications of the connective rules. Such a tableau begins with a single branch containing only quantifier rule applications up to a node N below which only connective rules are applied; the set of formulae on this branch up to the node N is called *the initial set* of the tableau, and the set of ground formulae in the initial set is termed *the initial ground set* of T.

It is evident that we can reorganize any tableau for a set of prenex formulae in such a way that it is quantifier preferring, without increasing its size or affecting its closedness. None of those properties is guaranteed to hold for sets containing formulae which are not in prenex form.

Proposition 2.21 If T is a closed quantifier preferring tableau for a set S of first-order formulae, then the initial ground set of T is unsatisfiable.

Since, for any unsatisfiable set S of prenex formulae, a closed quantifier preferring tableau exists, we can associate with S the collection of the initial ground sets of all closed quantifier preferring sentence tableaux for S. The sets in this collection, in particular the ones with minimal complexity, play an important rôle as a complexity measure.

Definition 2.22 (Herbrand complexity) The *Herbrand complexity* of an unsatisfiable set S of prenex formulae is the minimum of the complexities[4] of the initial ground sets of all closed quantifier preferring sentence tableaux for S.

Since the quantifier rules of tableaux are not specific to the tableau calculus, the Herbrand complexity can be used as a calculus-independent refutation complexity measure for unsatisfiable sets of formulae. This measure may also be extended to formulae which are not in prenex form, by working with transformations of the formulae in prenex form (see also [Baaz and Leitsch, 1992]).

Next, we come to an important proof-theoretic virtue of sentence tableaux, which we introduce generically for any system of tableau rules.

Definition 2.23 (Confluence) A tableau calculus C is called *proof confluent* or just *confluent* (for a class of formulae) if, for any unsatisfiable set S of formulae (from the class), from any tableau T for S constructed with the rules of C a closed tableau for S can be constructed with the rules of C.

Loosely speaking, a confluent (tableau) calculus does never run into dead ends.[5]

Proposition 2.24 Sentence tableaux are confluent for first-order sentences.

Proof Let T be any sentence tableau generated for an unsatisfiable set of sentences S. By the completeness of sentence tableaux, for any branch B in T, there exists a closed sentence tableau T_B for $S \cup B$. At the leaf of any branch B of T, simply repeat the construction of T_B. □

In the subsequent sections, we shall introduce tableau calculi and procedures that are not confluent and for which no systematic procedures of the type presented in this section exist. Nonconfluence may have strong consequences on the termination behaviour and the functionality of a calculus, particularly, when one is interested in decision procedures or in model generation for (sublanguages of) first-order logic. As we shall see, the lack of confluence may also require completely different approaches towards proving completeness.

2.1.3 Refinements of Tableaux

The calculus of sentence tableaux permits the performance of certain inference steps that are redundant in the sense that they do not contribute to the closing of the tableau. In order to avoid such redundancies, one can restrict the tableau rules and/or impose conditions on the tableau structure. First, we discuss the notion of strictness which is a refinement of the tableau rules. Adapted to our framework, it reads as follows.

[4] As the complexity of a set of formulae one may take the cardinality of the set plus the number of occurrences of symbols in the elements of the set.

[5] Note that the notion of confluence used here slightly differs from its definition in the area of term rewriting (see, e.g., [Huet, 1980].

Definition 2.25 (Strict tableau) A tableau construction is *strict* if

- every β- and δ-node is used only once on a branch and
- for any α-node, any occurrence of an α_i in α is used only once a branch.

The strictness condition is motivated by the definition of systematic tableaux, since obviously any systematic tableau construction is strict. Consequently, the strictness condition is completeness-preserving. Strictness can also be implemented very efficiently by simply labelling nodes (or occurrences of tableau subformulae at nodes) as already used on a branch. But strictness does not perform optimal redundancy elimination, since it does not prevent that one and the same formula may appear twice on a branch. This is particularly detrimental if it happens in a β-rule application where new branches with new proof obligations are produced, which obviously is completely useless. A stronger tableau restriction concerning the connective rules is achieved with the following *structural* condition.

Definition 2.26 (Regularity) A formula tree is called *regular* if on no branch a formula appears more than once.

The main reason why regularity has not been used in the traditional presentation of tableaux lies in the different definition of the α-rule here and there. We permit that only *one* α-subformula can be attached, whereas the traditional format requires to append *all* α-subformulae at once, one below the other. It is straightforward to realize that regularity is not compatible with the traditional definition of the α-rule. An obvious example is the unsatisfiable formula $(p \land q) \land (p \land \neg q)$, for which no closed regular tableau exists if the traditional α-rule is used. Since (wrt. the connective rules) the regularity condition is a more powerful mechanism of avoiding suboptimal proofs than the strictness condition, we have generalized the α-rule in order to achieve compatibility with regularity.[6] The following demonstrates that tableaux which are irregular can be safely ignored.

Procedure 2.27 (Removal of irregularities) Given any tableau T, repeat the following operation, until the resulting formula tree is regular.

- Select a node N in T with an ancestor node N' such that both nodes are labelled with the same literal. Remove the edges originating in the predecessor N'' of N and replace them with the edges originating in N.

Proposition 2.28 Every closed sentence tableau of minimal size[7] is regular.

[6] This is an interesting illustration of the fact that an unfortunate presentation of inference rules can block certain obvious pruning mechanisms.

[7] A precise measure for the size of a tableau is given in Definition 7.3. But this result is compatible with *any* reasonable measure.

Proof We show the contraposition, i.e., that every closed irregular sentence tableau T is not minimal in size. Let T be any closed irregular sentence tableau for a set S. Obtain a formula tree T' by performing Procedure 2.27 on T. Clearly, T' is a sentence tableau for S, it is smaller than T, and it is closed. □

In order to integrate the δ-rule restriction of strictness, we call a tableau *strictly regular* if it is strict and regular. The regularity restriction can easily be integrated into systematic tableaux, by simply omitting the attachment of nodes $B \oplus F_1 \mid \cdots \mid F_n$ if one of the F_i is already on the branch B.

A further fundamental refinement of sentence tableau concerns the γ-rule.

Definition 2.29 (Herbrand tableau) Herbrand tableaux are defined like sentence tableaux, but with the γ-rule replaced by the *Herbrand γ-rule*:

$(\gamma_{\mathcal{H}})$ $$\frac{\gamma}{\gamma(t)}$$ where t is from the Herbrand universe of the branch.

The Herbrand restriction on the γ-rule may significantly improve the termination behaviour of sentence tableaux, as illustrated with the formula F given in Example 2.30. The formula is satisfiable. But unfortunately, infinitely large sentence tableau can be constructed for F, as shown in Figure 2.30, since the γ-rule can be applied again and again using the formula (4). Any strict Herbrand tableau construction terminates, since the number of ground terms that can be selected for (4) is finite.

Example 2.30 $F = \neg\forall x(\exists y P(x,y) \rightarrow \exists y P(y,x))$

The Herbrand restriction on tableaux is as reasonable as regularity, since it preserves minimal proof size.

Proposition 2.31 For every (atomically) closed sentence tableau T for a set S, there exists a(n atomically) closed Herbrand tableau T' for S with less or equal size than T.

Proof Without increasing the size, we can rearrange T in such a way that all formula rule applications are performed first. Now consider any γ-step in the tableau that is not Herbrand, attaching a formula $\gamma(t)$ to the leaf N of a branch B. Replace any occurrence of t below N with a constant from the Herbrand universe of B. Obviously, the modified formula tree is (atomically) closed and does not increase in size. It is straightforward to recognize that the formula tree is a sentence tableau for S. Finitely many applications of this operation produce a(n atomically) closed Herbrand tableau T' for S equal or smaller in size than T. □

Proposition 2.32 Strictly regular Herbrand tableaux are confluent for first-order sentences.

$$
\begin{array}{l}
(1)\ \neg\forall x(\exists y P(x,y) \rightarrow \exists y P(y,x)) \\
\qquad \mid\ \boldsymbol{\delta}(1) \\
(2)\ \neg(\exists y P(a,y) \rightarrow \exists y P(y,a)) \\
\qquad \mid\ \boldsymbol{\alpha}(2) \\
(3)\ \exists y P(a,y) \\
\qquad \mid\ \boldsymbol{\alpha}(2) \\
(4)\ \neg\exists y P(y,a) \\
\qquad \mid\ \boldsymbol{\delta}(3) \\
(5)\ P(a,b) \\
\qquad \mid\ \boldsymbol{\gamma}(4) \\
(6)\ \neg P(a,a) \\
\qquad \mid\ \boldsymbol{\gamma}(4) \\
(7)\ \neg P(b,a) \\
\qquad \mid\ \boldsymbol{\gamma}(4) \\
(8)\ \neg P(c,a) \\
\qquad \vdots
\end{array}
\qquad\qquad
\begin{array}{l}
(1)\ \neg\forall x(\exists y P(x,y) \rightarrow \exists y P(y,x)) \\
\qquad \mid\ \boldsymbol{\delta}(1) \\
(2)\ \neg(\exists y P(a,y) \rightarrow \exists y P(y,a)) \\
\qquad \mid\ \boldsymbol{\alpha}(2) \\
(3)\ \exists y P(a,y) \\
\qquad \mid\ \boldsymbol{\alpha}(2) \\
(4)\ \neg\exists y P(y,a) \\
\qquad \mid\ \boldsymbol{\delta}(3) \\
(5)\ P(a,b) \\
\qquad \mid\ \boldsymbol{\gamma}_{\mathcal{H}}(4) \\
(6)\ \neg P(a,a) \\
\qquad \mid\ \boldsymbol{\gamma}_{\mathcal{H}}(4) \\
(7)\ \neg P(b,a) \\
\text{all Herbrand terms} \\
\text{selected for (4)}
\end{array}
$$

Figure 2.4: Sentence and Herband tableau for Example 2.30.

Proof Let T be any strictly regular Herbrand tableau for an unsatisfiable set of sentences S. By the completeness of sentence tableaux, there exists a closed sentence tableau T' for S. Simply repeat the construction of T', at any leaf of T. Now modify the resulting sentence tableau, as described in the proofs of Propositions 2.28 and 2.31. The procedure results in a closed strictly regular Herbrand tableaux T'' for S. Since the modification operation is performed from the leaves towards the root, it does not affect the inital tree T, hence T'' is as desired. □

The Herbrand tableau rule also has an effect on the *systematic* tableau construction. Since the Herbrand universe may increase during branch expansion, the enumeration of ground terms must be organized differently.

Definition 2.33 (Systematic Herbrand tableau) Systematic Herbrand tableaux are defined like systematic tableaux except that the γ-rule application is controlled differently. Whenever a γ at a node N is selected, for any atomically open branch B through N, select the smallest term t (modulo the ordering π) from the Herbrand universe of B that has not been selected at N on B; if all terms from the Herbrand universe of B have already been selected at N on B, γ cannot be used for expanding the current leaf of B.

In particular, this entails that, for different branches, different Herbrand terms may be selected in the systematic tableau construction. Imposing the Herbrand restriction on systematic tableaux preserves completeness, since Proposition 2.17 also holds for Herbrand tableaux.

Proposition 2.34 Every (atomically) open branch B of a saturated regular systematic Herbrand tableau is a(n atomic) Hintikka set, moreover, the set of atoms on B defines a Herbrand model for B.

Proof See the proof of Proposition 2.17. □

Herbrand tableaux provide a higher functionality than sentence tableaux, since a larger class of first-order formulae can be decided.

Definition 2.35 (Weak Skolem, datalogic form) A sentence Φ is said to be in *weak Skolem form* if Φ has no tableau subformula of type γ that has a tableau subformula of type δ. A sentence Φ is said to be in *(weak) datalogic form* if Φ is in (weak) Skolem form, respectively, and Φ has no function symbol of arity > 0.

The set of weak datalogic formulae is a generalization of the Bernays-Schönfinkel class [Bernays and Schönfinkel, 1928].

Proposition 2.36 Every strictly regular Herbrand tableau for any finite set S of weak datalogic formulae is finite.

Proof The formula structure and the tableau rules guarantee that only δ-formulae can appear on a branch which occur as subformulae in the elements of S. Since S is assumed as finite, this entails that the number of δ-formulae on any branch must be finite. Because of the strictness condition on the δ-rule, only finitely many new constants can occur on a branch. Since no function symbols of arity > 0 occur in the elements of S, the Herbrand universe of any branch must be finite, and hence the set of formulae occurring on a branch. Regularity then assures that also the length of any branch must be finite. □

Both properties demonstrate that Herband tableaux are decision procedures for the class.

Proposition 2.37 Given any finite set S of weak datalogic formulae, any regular systematic Herbrand tableau construction terminates,

- *either with a closed tableau if S is unsatisfiable,*
- *or with an open branch which defines a Herbrand model for S.*

2.2 Free-variable Tableaux

The tableau approach is traditionally useful as an elegant format for *presenting* proofs. With the increasing importance of automatic deduction, however, the question arises whether the tableau paradigm is also suited for proof *search*. In principle, systematic tableau procedures could be used for this purpose. But systematic procedures, even regular Herbrand ones, are still too inefficient for a broad application. As an illustration, see the tableau displayed in Figure 2.2, which is not systematic. A systematic tableau would be much larger. The essential difference concerns the applications of the γ-rule. Consider, e.g., the γ-step from node (10) $\neg\exists z P(g(b), z)$ to node (11) $\neg P(g(b), f(a, g(b), a))$ in which the "right" substitution $\{z/f(a, g(b), a))\}$ has been selected. Since a systematic procedure has to enumerate *all* (Herbrand) instances in a systematic and therefore "blind" manner, it would normally perform the substitution $\{z/f(a, g(b), a))\}$ much later. The obvious weakness of the γ-rule is that it enforces to perform ground instantiations too early, at a time when it is not clear whether the substitution will contribute to the closing of a branch. The natural approach for overcoming this problem is to postpone the term selection completely by permitting free variables in a tableau and to determine the instances later when they can be used to immediately close a branch. The free variables are then treated in a *rigid* manner, i.e., they are not being considered universally quantified but as placeholders for arbitrary (ground) terms. This view of free variables dates back to work of Prawitz [Prawitz, 1960], was applied by Bibel [Bibel, 1981] and Andrews [Andrews, 1981], and incorporated into tableaux, for example, by Fitting [Fitting, 1990] (see also [Reeves, 1987]). In this section, we will investigate this approach. Closure of a branch means producing two *complementary* formulae, i.e., a formula and its negation, on the branch. Since we can confine ourselves to atomic closures, the problem can be reduced to finding a substitution σ such that for two literals K and L on the branch: $K\sigma = \sim L\sigma$. So one has to integrate *unification* into the tableau calculus.

2.2.1 Unification

Unification is one of the most successful advances in automated deduction, because it permits to make instantiation optimal with respect to generality. Unification will be introduced here for arbitrary finite sets of quantifier-free expressions.

Definition 2.38 (Unifier) For any finite set S of quantifier-free expressions and any substitution σ, if $|S\sigma| = 1$,[8] then σ is called a *unifier for* S. If a unifier exists for a set S, then S is called *unifiable*.

Subsequently, we will always assume that S denotes finite sets of quantifier-free expressions. The general notion of a unifier can be subclassified in certain useful ways.

[8] With $|M|$ we denote the cardinality of a set M.

Definition 2.39 (Most general unifier) If σ and τ are substitutions and there is a substitution θ such that $\tau = \sigma\theta$, then we say that σ is *more general* than τ. A unifier for a set S is called a *most general unifier*, MGU for short, if σ is more general than any unifier for S.

Most general unifiers have the nice property that any unifier for two atoms can be generated from a most general unifier by further composition. This qualifies MGUs as a useful instantiation vehicle in many inference systems. The central unifier concept in automated deduction, however, is the following.

Definition 2.40 (Minimal unifier) If a unifier σ for a set S has the property that for every unifier τ for S: $|\sigma| \leq |\tau|$, then we say that σ is a *minimal unifier for S*.

For a minimal unifier the number of substituted variables is minimal.

Example 2.41 Given the set of terms $S = \{x, f(y)\}$, the two substitutions $\sigma = \{y/x, x/f(x)\}$ and $\tau = \{x/f(y)\}$ are both MGUs for S, but only τ is a minimal unifier.

In fact, every minimal unifier is a most general unifier, as will be shown in the Unification Theorem (Theorem 2.50) below. How can we a find a minimal unifier for a given set? For this purpose, the procedurally oriented concept of a *computed unifier* will be developed.

Definition 2.42 (Disagreement set) Let S be a finite set of quantifier-free expressions. A *disagreement set* of S is any two-element set $\{E_1, E_2\}$ of expressions such that the dominating symbols of E_1 and E_2 are distinct and E_1 and E_2 occur at the same position as subexpressions in two of the expressions in S.

Example 2.43 The set of terms $S = \{x, g(a, y, u), g(z, b, v)\}$ has the following disagreement sets: $\{a, z\}$, $\{y, b\}$, $\{u, v\}$, $\{x, g(a, y, u)\}$, $\{x, g(z, b, v)\}$.

Obviously, a set of expressions S has a disagreement set if and only if $|S| > 1$. The following facts immediately follow from the above definitions.

Proposition 2.44 If σ is a unifier for a set S and D is a disagreement set of S, then σ unifies D, each member of D is a term, and D contains a variable which does not occur in the other term of D.

The last item of the proposition expresses that any binding formed from any disagreement set of a unifiable set must be a proper binding. Operationally, the examination whether a binding is proper is called the *occurs-check*. A particularly useful technical tool for proving the Unification Theorem below is the Decomposition Lemma.

Lemma 2.45 (Decomposition Lemma) Let σ be a unifier for a set S with $|S| > 1$ and let $\{x, t\}$ be any disagreement set of S with $x \neq x\sigma$. If $\tau = \sigma \setminus \{x/x\sigma\}$, then $\sigma = \{x/t\}\tau$.

Proof First, since σ unifies any disagreement set of S, $x\sigma = t\sigma$. By Proposition 2.44, x does not occur in t, which gives us $t\sigma = t\tau$. Consequently, $x\sigma = t\tau$ and $x \neq t\tau$. Furthermore, $x \notin \text{domain}(\tau)$, and by the composition of substitutions, $\{x/t\}\tau = \{x/t\tau\} \cup \tau$. Putting all this together yields the chain $\{x/t\}\tau = \{x/t\tau\} \cup \tau = \{x/x\sigma\} \cup \tau = \sigma$. □

Now we shall introduce a concept which captures the elementary operation performed when making a set of expressions equal by instantiation. It works by eliminating exactly one variable x from all expressions of the set and by replacing this variable with another term t from a disagreement set $\{x, t\}$ of S provided that x does not occur in t.

Definition 2.46 (Variable elimination) If S is a finite set of expressions such that from the elements of one of its disagreement sets a proper binding x/t can be formed, then $S\{x/t\}$ is said to be *obtainable from* S by a *variable elimination wrt* x/t.

Proposition 2.47 Let S be any finite set of quantifier-free expressions and let V_S be the set of variables occurring in S.

1. *If S is unifiable, so are all sets obtainable from S by a variable elimination.*
2. *Only finitely many sets can be obtained from S by a variable elimination.*
3. *If S' has been obtained from S by a variable elimination wrt a binding $\{x/t\}$ and $V_{S'}$ is the set of variables occurring in S', then $|S'| \leq |S|$ and $V_{S'} = V_S \setminus \{x\}$.*
4. *The transitive closure of the relation*

$$\{\langle S', S\rangle \mid S' \text{ can be obtained from } S \text{ by a variable elimination step}\}$$

is well-founded where S and S' are arbitrary finite sets of quantifier-free expressions, i.e., there are no infinite sequences of successive variable elimination steps.

Proof For the proof of (1), let $S' = S\{x/t\}$ be obtained from S by a variable elimination wrt to the binding x/t composed from a disagreement set of S, and suppose σ unifies S. Since σ unifies every disagreement set of S, it follows that $x\sigma = t\sigma$. Let $\tau = \sigma \setminus \{x/x\sigma\}$. By the Decomposition Lemma (Lemma 2.45), we have $\{x/t\}\tau = \sigma$. Therefore, $S(\{x/t\}\tau) = (S\{x/t\})\tau = S'\tau$. Hence τ unifies S'. For (2) note that since there are only finitely many disagreement sets of S and each of them is finite, only finitely many proper bindings are induced, and hence only finitely many sets can be obtained by a variable elimination. To recognize (3), let $S' = S\{x/t\}$ be any set obtained from S by a variable elimination. Then S' is the result of replacing any occurrence of x in S by the term t. Therefore, $|S'| \leq |S|$, and, since x/t is proper and t already occurs in S, we get $V_{S'} = V_S \setminus \{x\}$. Lastly, (4) is an immediate consequence of (3). □

Now we are able to introduce the important notion of a computed unifier, which is defined by induction on the cardinality of the unifier.

Definition 2.48 (Computed unifier) (inductive)

1. $\emptyset$ is a (the only) *computed unifier for* any singleton set of quantifier-free expressions.
2. If a substitution σ of cardinality n is a computed unifier for a finite set S' and S' can be obtained from S by a variable elimination wrt a binding x/t, then the substitution $\sigma \cup \{x/t\sigma\} = \{x/t\}\sigma$ of cardinality $n+1$ is a *computed unifier for* S.

The definition of a computed unifier can be seen as a declarative specification of an algorithm for *really computing* a unifier for a given set of expressions, which we will present now using a procedural notation. The procedure is a generalization of the algorithm given by Robinson in [Robinson, 1965].[9]

Definition 2.49 (Unification algorithm) Let S be any finite set of quantifier-free expressions. $\sigma_0 = \emptyset$, $S_0 = S$, and $k = 0$. Then go to 1.

1. If $|S_k| = 1$, output σ_k as a computed unifier for S. Otherwise select a disagreement set D_k of S_k and go to 2.
2. If D_k contains a proper binding, choose one, say x/t; then set $\sigma_{k+1} = \sigma_k\{x/t\}$, set $S_{k+1} = S_k\{x/t\}$, increment k by 1 and go to 1. Otherwise output "not unifiable".

Note that the unification algorithm is a nondeterministic procedure. This is because there may be several different choices for a disagreement set and for a binding. Evidently, the unification procedure can be directly read off from the definition of a computed unifier: it just successively performs variable elimination operations, until either there are no variable elimination steps possible, or the resulting set is a singleton set. Conversely, the notion of a computed unifier is an adequate declarative specification of the unification algorithm. It follows immediately from Proposition 2.47 (1) and (4) that each unifier output of the unification algorithm is indeed a computed unifier and that the procedure terminates, respectively.

We shall demonstrate now that the notions of a minimal and a computed unifier coincide, and that both of them are most general unifiers.

Theorem 2.50 (Unification Theorem) Let S be any unifiable finite set of quantifier-free expressions.

1. *If σ is a minimal unifier for S, then σ is a computed unifier for S.*
2. *If σ is a computed unifier for S, then σ is a minimal unifier for S.*

[9] Historically, the first unification procedure was given by Herbrand in [Herbrand, 1930].

3. If σ is a computed unifier for S, then σ is an MGU *for S.*

Proof We will prove (1) to (3) by induction on the cardinalities of the respective unifiers. First, note that $\emptyset$ is the only minimal and computed unifier for any singleton set of quantifier-free expressions S and that $\emptyset$ is an MGU for S. Assume the result to hold for any set of expressions with minimal and computed unifiers of cardinalities $\leq n$. For the induction step, suppose S has only minimal or computed unifiers of cardinality $> n \geq 0$. Let σ be an arbitrary unifier for S and x/t any proper binding from a disagreement set of S with $x \neq x\sigma$ (which exists by Proposition 2.44). Let $S' = S\{x/t\}$ and set $\tau = \sigma \setminus \{x/x\sigma\}$, which is a unifier for S', by the Decomposition Lemma (Lemma 2.45). For the proof of (1), let σ be a minimal unifier for S. We first show that τ is minimal for S'. If θ' is any minimal unifier for S', then $\theta = \{x/t\}\theta'$ is a unifier for S and all variables in domain(θ') occur in S'. Therefore, the Decomposition Lemma can be applied yielding that $\theta' = \theta \setminus \{x/x\theta\}$. And from the chain $|\theta'| = |\theta| - 1 \geq |\sigma| - 1 = |\tau|$ it follows that τ is a minimal unifier for S'. Since $|\tau| \leq n$, by the induction assumption, τ is a computed unifier for S'. Hence, by definition, $\sigma = \{x/t\}\tau$ is a computed unifier for S. For (2) and (3), let σ be a computed unifier for S. Then, by definition, τ is a computed unifier for S'. Let θ be an arbitrary unifier for S. Since x is in some disagreement set of S, either $x \in$ domain(θ) or there is a variable y and $y/x \in \theta$. Define

$$\eta = \begin{cases} \theta & \text{if } x \in \text{domain}(\theta) \\ \theta\{x/y\} & \text{otherwise.} \end{cases}$$

Since $x \in$ domain(η), the Decomposition Lemma yields that if $\eta' = \eta \setminus \{x/x\eta\}$, then $\{x/t\}\eta' = \eta$, and η' is a unifier for S'. The minimality of σ can be recognized as follows. By the induction assumption, τ is minimal for S'. Then consider the chain

$$|\theta| = |\eta| = |\eta'| + 1 \geq |\tau| + 1 = |\sigma|.$$

For (3), note that τ is an MGU for S', by the induction assumption, i.e., there is a substitution γ: $\eta' = \tau\gamma$. On the other hand, $\theta = \theta\{x/y\}\{y/x\}$, hence there is a substitution ν: $\theta = \eta\nu$. This gives us the chain

$$S\theta = S\eta\nu = S\{x/t\}\eta'\nu = S\{x/t\}\tau\gamma\nu = S\sigma\gamma\nu$$

demonstrating that σ is an MGU for S. This completes the proof of the Unification Theorem. □

Concerning terminology, notions are treated differently in the literature (see [Lassez et al., 1988] for a comparison). We have chosen a highly indeterministic presentation of the unification algorithm, it is even permitted to select between alternative disagreement sets. Furthermore, we have stressed the importance of minimal unifiers. Therefore our Unification Theorem is stronger than normally presented, it also states that *each* minimal unifier indeed can be computed by the unification algorithm.

Polynomial unification

Unification is the central ingredient applied in all advanced proof systems for first-order logic. As a consequence, the complexity of unification is a lower bound for the complexity of each advanced calculus. While the cardinality of a most general unifier σ for a set of expressions S is always bounded by the number of variables in S, the range of the unifier may contain terms with a size exponential in the size of the initial expressions. Of course, this would also entail that $S\sigma$ contains expressions with an exponential size. The following class of examples demonstrates this fact.

Example 2.51 If P is an $(n+1)$-ary predicate symbol and f a binary function symbol, then, for every $n > 1$, define S_n as the set containing the atomic formulae

$$P(x_1, x_2, \ldots, x_n, x_n), \text{ and}$$
$$P(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}), x_n).$$

Obviously, any unifier for an S_n must contain a binding x_n/t such that the number of symbol occurrences in t is greater than 2^n. As a consequence, we have the problem of exponential space and, therefore, also of exponential time, when working with such structures. Different solutions have been proposed for doing unification polynomially. In [Paterson and Wegman, 1978], a linear unification algorithm is presented. Furthermore, a number of "almost" linear algorithms have been developed, for example, in [Huet, 1976] and [Martelli and Montanari, 1976, Martelli and Montanari, 1982]. Similar to the early approach in [Herbrand, 1930], all of the mentioned efficient algorithms reduce the unification problem to the problem of solving a set of equations. However, all of those procedures—particularly the one in [Paterson and Wegman, 1978]—need sophisticated and expensive additional data structures, which render them not optimal for small or average sized expressions. Therefore, Corbin and Bidoit rehabilitated Robinson's exponential algorithm by improving it with little additional data structure up to a quadratic worst-case complexity [Corbin and Bidoit, 1983, Letz, 1993a]. This algorithm turns out to be very efficient in practice.

We cannot treat polynomial unification in detail here, but we give the essential two ideas contained in any of the mentioned polynomial unification algorithms.

1. The representation of expressions has to be generalized from strings or trees to directed acyclic graphs, *dags* for short. This way, the space complexity can be reduced from exponential to linear, as shown with the directed acyclic graph representing the term $x_n\sigma$ in the example above:

$$\underbrace{f \rightrightarrows f \rightrightarrows \cdots f \rightrightarrows}_{n-\text{times}} x_0$$

2. In order to reduce the time complexity—which may still be exponential even if graphs are used, since there may be exponentially many paths through a graph—, the following will work. One must remember

- which pairs of expressions have already been unified in the graph (e.g., during the unification of $x_n\sigma$ with itself at the last argument positions of the atoms),
- and in occurs-checks: for which expressions the occurrence of the respective variable was already checked (e.g., during the check whether x_n occurs in $f(x_{n-1}, x_{n-1})\sigma$ at the n-th argument positions of the atoms),

and one must not repeat those operations. How sophisticated this is organized determines whether the worst-case complexity can be reduced to linear or just to quadratic time.

2.2.2 Generalized Quantifier Rules

Using the unification concept, the γ-rule of sentence tableaux can be modified in such a way that instantiations of γ-formulae are delayed until a branch can be immediately closed. Two further modifications have to be performed. On the one hand, since now free variables occur in the tableau, one has to generalize the δ-rule to full Skolemization in order to preserve soundness.

Example 2.52 Consider the satisfiable formula $\exists y(\neg P(x, y) \wedge P(x, x))$. An application of the δ-rule of sentence tableaux would result in an unsatisfiable formula $\neg P(x, a) \wedge P(x, x)$.

One the other hand, substitutions have to be applied to the formulae in a tableau. With $T\sigma$ we denote the result of applying a substitution σ to the formulae in a tableau T. Before defining tableau with unification, we introduce a tableau system in which arbitrary substitutions can be applied. This system will serve as a very general reference system, which also subsumes sentence tableaux,

Definition 2.53 (General free-variable tableau) General free-variable tableaux are defined as sentence tableaux are, but with the γ- and the δ-rule replaced by the following three rules. Let B be (the set of formulae on) the actual tableau branch and S the set of input sentences of the current tableau T.

$(\gamma *)$ $\quad \dfrac{\gamma}{\gamma(t)}$ where t is any term of the language $\mathcal{L}$ and $\{x/t\}$ is free for $\gamma(x)$

(δ^+) $\quad \dfrac{\delta}{\delta(f(x_1, \ldots, x_n))}$ where f is new to S and T, and $x_1, \ldots, x_n$ are the free variables in δ,

(S) $\quad$ Modify T to $T\sigma$ where σ is free for all formulae in T.

The δ^+-rule [Hähnle and Schmitt, 1994, Fitting, 1996] we use here is already an improvement of the original δ-rule used in [Fitting, 1990]. The difference between both rules will be discussed in Section 6.2. The additional *substitution rule* denoted with (S), which is now needed to achieve closure of certain branches, differs strongly from the tableau rules presented up to now. While all those rules were conservative in the sense that the initial tableau was not modified but just expanded, the substitution rule is *destructive*. This has severe consequences on free-variable tableaux, both proof-theoretically and concerning the functionality of the calculus, which will be discussed below.

But how do we know that the calculus of general free-variable tableau produces correct proofs? It is clear that the method of proving the correctness of sentence tableaux (using Proposition 2.8) will not work. In free-variable tableaux, branches cannot be treated separately, because they may share free variables. As an example, consider a tableau T with the two branches $P(x) \oplus \neg P(a)$ and $Q(x) \oplus \neg Q(b)$ which cannot be closed using the rules of general free-variable tableaux, although both branches are unsatisfiable. The notion of satisfiability is too coarse for free-variable tableaux. What will work here is the following finer notion which was developed in [Hähnle and Schmitt, 1994] and also used in [Fitting, 1996].

*Definition 2.54 ($\forall$-**satisfiability**))* A collection $\mathcal{C}$ of sets of first-order formulae is called $\forall$-***satisfiable*** if there is an interpretation $\mathcal{I}$ such that, for every variable assignment $\mathcal{A}$, $\mathcal{I}$ is an $\{\mathcal{A}\}$-model for some element of $\mathcal{C}$.

It is evident that, for closed first-order formulae, $\forall$-satisfiability of a collection coincides with ordinary satisfiability of some element of the collection. In order to illustrate the difference of this concept for formulae with free variables, consider the tableau T mentionend above. The collection consisting of the two sets of formulae $\{P(x), \neg P(a)\}$ and $\{Q(x), \neg Q(b)\}$ is $\forall$-satisfiable (set, e.g., $\mathcal{U} = \{0, 1\}$, $\mathcal{I}(P) = \{0\}$, $\mathcal{I}(Q) = \{1\}$, $\mathcal{I}(a) = 1$, and $\mathcal{I}(b) = 0$).

We now give a generalized version of Proposition 2.8 which can be used to prove correctness of both sentence tableaux and general free-variable tableaux.

Proposition 2.55 Let $C' = C \cup \{S\}$ be a $\forall$-satisfiable collection of sets of first-order formulae.

1. *If $\alpha \in S$, then $C \cup \{S \cup \{\alpha_i\}\}$ is $\forall$-satisfiable, for every α-subformula α_i of α.*

2. *If $\beta \in S$, then $C \cup \{S \cup \{\beta_1\}, \ldots, S \cup \{\beta_n\}\}$ is $\forall$-satisfiable where $\beta_1, \ldots, \beta_n$ is the β-subformula sequence of β.*

3. *If $\forall x F = \gamma \in S$, then $C \cup \{S \cup \{\gamma(t)\}\}$ is $\forall$-satisfiable, for any term t of the language $\mathcal{L}$ provided $\{x/t\}$ is free for F.*

4. *If $\delta \in S$, then $C \cup \{S \cup \{\delta(t)\}\}$ is $\forall$-satisfiable for any Skolemization $\delta(t)$ of δ wrt $\bigcup C'$.*

5. *$C'\sigma$ is $\forall$-satisfiable, for any substitution σ which is free for all formulae in $\bigcup C'$.*

Proof By the definition of $\forall$-satisfiability, there is an interpretation $\mathcal{I}$ such that, for every variable assignment $\mathcal{A}$, $\mathcal{I}$ is an $\{\mathcal{A}\}$-model for some member of C'. The non-trivial case for proving items (1) to (4) is the one in which S is $\{\mathcal{A}\}$-satisfied by $\mathcal{I}$ and no element of C is. Let A be the collection of all variable assignments, for which this holds. Items (1) and (2) are immediate from the definition of formula assignment. For (3), let $\mathcal{A}$ be an arbitrary element from A. Then, be item (8) of formula assignments, $\mathcal{I}^{\mathcal{A}'}(F) = \top$, for all x-variants of $\mathcal{A}$. Since $\mathcal{A}\sigma$ is an x-variant of $\mathcal{A}$, $\mathcal{I}^{\mathcal{A}\sigma}(F) = \top$. Now σ is free for F, therefore, Lemma 1.45 can be applied yielding that $\mathcal{I}^{\mathcal{A}}(F\sigma) = \mathcal{I}^{\mathcal{A}\sigma}(F)$. Item (4): since $\delta(t)$ is a Skolemization of δ wrt $\bigcup C'$, by Proposition 1.55, there exists an A-model $\mathcal{I}'$ for $S \cup \{\delta(t)\}$ which is identical to $\mathcal{I}$ except for the interpretation of the new function symbol f in $\delta(t)$. Since f does not occur in C, for all variable assignment $\mathcal{A} \notin A$, some element of C is $\{\mathcal{A}\}$-satisfied by $\mathcal{I}'$. Consequently, $\mathcal{I}'$ is a $\forall$-model for $C \cup \{S \cup \{\delta(t)\}\}$. For (5), let $\mathcal{A}$ be any variable assignment. Consider its modification $\mathcal{A}\sigma$. Since C' is assumed as $\forall$-satisfiable, $\mathcal{I}^{\mathcal{A}\sigma}(S') = \top$, for some $S' \in C'$. Now σ is free for F, therefore, again by Lemma 1.45, $\mathcal{I}^{\mathcal{A}}(S\sigma) = \mathcal{I}^{\mathcal{A}\sigma}(S)$. □

Proposition 2.56 (Soundness of general free-variable tableaux) If a set of formulae S is satisfiable, then every general free-variable tableau for S has an open branch.

Proof First, note that the satisfiability of a set of formulae S entails the $\forall$-satisfiability of the collection $\{S\}$. Then the proof is by induction on the number of inference steps, using Proposition 2.55 on the collection of the sets of formulae on the branches of a general free-variable tableau. □

The completeness of general free-variable tableaux is trivial, because the calculus is obviously a generalization of the calculus of sentence tableaux. So general free-variable tableaux are only relevant as a common framework but not as a calculus supporting the *finding* of proofs. What we are interested in is to apply a substitution only if this immediately leads to the closure of a branch, and we will even restrict this to an atomic closure.

Definition 2.57 (Free-variable tableau) Free-variable tableaux with atomic closure, or just *free-variable tableaux*, are defined as general free-variable tableaux, but with the following two modifications. The γ^*-rule is replaced with the weaker γ'-rule and the substitution rule (S) is replaced with the weaker *closure rule* (C)

(γ')	$\dfrac{\gamma}{\gamma(x)}$	where x is a variable new to S and T,
(C)	Modify T to $T\sigma$	if two literals K and L are on a branch such that σ is a minimal unifier for $\{K, {\sim}L\}$.

Note that the applied substitution will be automatically free for the formulae in the tableau. This is because the γ'-rule guarantees that no variable occurs

$$(1)\ \neg\exists x(\forall y\forall z P(y,f(x,y,z)) \rightarrow (\forall y P(y,f(x,y,x)) \wedge \forall y\exists z P(g(y),z)))$$

$$\Big|\ \gamma'(1)$$

$$(2)\ \neg(\forall y\forall z P(y,f(x_1,y,z)) \rightarrow (\forall y P(y,f(x_1,y,x_1)) \wedge \forall y\exists z P(g(y),z)))$$

$$\Big|\ \alpha(2)$$

$$(3)\ \forall y\forall z P(y,f(x_1,y,z))$$

$$\Big|\ \alpha(2)$$

$$(4)\ \neg(\forall y P(y,f(x_1,y,x_1)) \wedge \forall y\exists z P(g(y),z))$$

$$\beta(4)$$

$$\begin{array}{ll}
(5)\ \neg\forall y P(y,f(x_1,y,x_1)) & (6)\ \neg\forall y\exists z P(g(y),z) \\
\quad\Big|\ \delta'(5) & \quad\Big|\ \delta'(6) \\
(7)\ \neg P(h(x_1),f(x_1,h(x_1),x_1)) & (10)\ \neg\exists z P(g(b),z) \\
\quad\Big|\ \gamma'(3) & \quad\Big|\ \gamma'(10) \\
(8)\ \forall z P(y_1,f(x_1,y_1,z)) & (11)\ \neg P(g(b),z_2) \\
\quad\Big|\ \gamma'(8) & \quad\Big|\ \gamma'(3) \\
(9)\ P(y_1,f(x_1,y_1,z_1)) & (12)\ \forall z P(y_2,f(x_1,y_2,z)) \\
\sigma_1 = \{y_1/h(x_1), z_1/x_1\} & \quad\Big|\ \gamma'(12) \\
 & (13)\ P(y_2,f(x_1,y_2,z_3)) \\
 & \sigma_2 = \{y_2/g(b), z_2/f(x_1,g(b),z_3)\}
\end{array}$$

Figure 2.5: Closed free-variable tableau.

bound and free in formulae of the tableau and, since K and L are quantifier-free, the minimal unifier σ has only free variables in the terms of its range.

Let us now consider an example. It is apparent that the destructive modifications render it more difficult to represent a free-variable tableau deduction. We solve this problem by not applying the substitutions $\sigma_1, \ldots, \sigma_n$ explicitly to the tableau T, but by annotating them below the nodes at the respective leaves. The represented tableau then is $T\sigma_1 \cdots \sigma_n$. In Figure 2.5, a free-variable tableau for the same first-order sentence is displayed for which in Figure 2.2 a sentence tableau is displayed. Comparing both tableaux, we can observe that it is much easier to find the closed free-variable tableau than the closed sentence tableau. The substitutions that close the branches need not be blindly guessed, they can be automatically computed from the respective pairs of literals to be unified, viz. (7) and (9) on the left and (11) and (13) on the right branch.

2.2.3 Completeness of Free-Variable Tableaux

The completeness of free-variable tableaux is not difficult to prove for formulae in Skolem or weak Skolem form[10], since the construction of any atomically closed sentence tableau can be simulated step by step by the calculus of free-variable tableaux. This is evident, because only the γ-rule has a different effect for this class of formulae. The simulation then proceeds by simply delaying the instantiations of γ-formulae and performing the substitutions later by using the closure rule. Unfortunately, for general formulae, no identical simulation of sentence tableaux is possible, as becomes clear when comparing Figure 2.5 with Figure 2.2. The problem is that more complex Skolem functions may be necessary in free-variable tableaux. But modulo such a modification, a so-called *Skolem variant*, a tree-isomorphic simulation exists.

Definition 2.58 (Skolem variant of a sentence tableau) (inductive)

1. Any sentence tableau T is a *Skolem variant* of itself.
2. If c is a constant introduced by a δ-rule application on a branch B of a Skolem variant T' of a sentence tableau T and t is any ground term whose dominating function symbol is new to B, then the formula tree obtained from replacing any occurrence of c in T' by t is a Skolem variant of T.

It is clear that Skolem variants preserve the closedness of a formula tree.

Lemma 2.59 Any Skolem variant of a(n atomically) closed sentence tableau is (atomically) closed.

Another problem is that the order in which a free-variable tableau is constructed can influence the arity of the Skolem functions in δ^+-rules. Consider, for example, a tableau consisting of a left branch $P(x) \oplus \neg P(a)$ and a right branch $\exists y(Q(x, y) \wedge \neg Q(a, y))$. If we decide to close the left branch first using the unifier $\{x/a\}$, then the performance of the δ^+-rule on the instantiated right branch will produce a Skolem constant. If the right branch is selected first, then we have to introduce a complex Skolem term $f(x)$, since x is still free. So, in the presence of δ-formulae, different orders of constructing a free-variable tableau can make a difference in the final tableau, as opposed to sentence tableaux which are completely independent of the construction order. As a matter of fact, we want completeness of free-variable tableaux independent of the construction order. The order of construction is formalized with the notion of a *(branch) selection function.*

Definition 2.60 (Selection function) A *(branch) selection function* ϕ is a mapping assigning an open branch to every tableau T which is not atomically closed. Let ϕ be a branch selection function and let $T_1, \ldots, T_n$ be a sequence of *successive* tableaux, i.e., each T_{i+1}, can be obtained from T_i by a tableau inference step. The tableau T_n is said to be *(constructed) according to* ϕ if each T_{i+1} can be obtained from T_i by performing an inference on the branch $\phi(T_i)$.

[10] I.e., in which no tableau subformula of type γ has a tableau subformula of type δ.

Lemma 2.61 Let T' be any atomically closed sentence tableau. Then, for any branch selection function ϕ, there exists an atomically closed free-variable tableau T for S constructed according to ϕ such that T is more general than a Skolem variant of T'; and if every formula $F \in S$ is in weak Skolem form, then T is even more general than T'.

Proof Let T' be any atomically closed sentence tableau and ϕ any branch selection function. We define sequences $T_1, \ldots, T_m$ of free-variable tableaux which correspond to initial segments of T' as follows. T_1 is the one-node initial tableau of T'. Let T_i be the i-th element of such a sequence $T_1, \ldots, T_m$, $1 \leq i < m$, and B the inital segment of the branch in T' which corresponds to the selected branch $\phi(T_i)$ in T_i, i.e., B and $\phi(T_i)$ are paths from the root to the same tree position.

1. If B is atomically open, then some expansion step has been performed in the construction of T' to expand B. T_{i+1} is the result of performing a corresponding free-variable tableau expansion step on $\phi(T_i)$.

2. If B is atomically closed (and $\phi(T_i)$ is atomically open), then two complementary literals must be on B. Let K and L be the corresponding literals on $\phi(T_i)$. T_{i+1} is the result of applying a minimal unifier of $\{K, {\sim}L\}$ to T_i.

We show by induction on the sequence length that any of the tableaux in such a sequence is more general than an initial segment of some Skolem variant of T'. The induction base is evident. For the induction step, let T_i be more general than an initial segment T_i^{Sk} of a Skolem variant of T'. For case (1), we consider first the subcase in which B is not expanded by a δ-step. Then an expansion of $\phi(T_i)$ corresponding to the sentence tableau expansion of B is possible and produces a tableau T_{i+1} that is more general than the respective expansion of T_i^{Sk}, which is an initial segment of some Skolem variant of T'. The subcase of δ-expansion is the problematic one, since one (possibly) has to move to another Skolem variant of T'. Let $\delta(f(x_1, \ldots, x_n))$, $n \geq 0$, be the formula by which T_i was expanded. Each variable x_j, $1 \leq j \leq n$, has been introduced in T_i by a γ'-step using a node N_j. If $t_1, \ldots, t_n$ are the respective ground terms at the same term positions in T_i^{Sk}, then let T_{i+1}^{Sk} be the formula tree obtained by expanding the branch corresponding to B with $\delta(f(t_1, \ldots, t_n))$. By construction, T_{i+1} is more general than T_{i+1}^{Sk}, which is an initial segment of a Skolem variant of T'. In case (2), $\phi(T_i)$ is atomically open, but B is atomically closed. By the induction assumption, T_i is more general than T_i^{Sk}, which has the branch atomically closed. Therefore, there exists a minimal unifier for the literals K and the complement of L on $\phi(T_i)$, and $T_{i+1} = T_i\sigma$ is more general than T_i^{Sk}. Now any such sequence $T_1, \ldots, T_m$ must be of finite length, since in each simulation step a different node position of T is either expanded or closed, i.e., m is less or equal to the number of nodes of T'. Consequently, $T = T_m$ is an atomically closed free-variable tableau for S that is more general than a Skolem variant of T'. Finally, if S is in weak Skolem form, no free variable can occur in a δ-formula in a free-variable tableau for S. In this case, one can always use the same Skolem constants in the construction of T and T'

and never has to move to a proper Skolem variant of T'. Then T is more general than T'. □

From this lemma immediately follows the refutational completeness of free-variable tableaux.

Theorem 2.62 (Free-variable tableau completeness) If S is an unsatisfiable set of first-order sentences, then there exists a finite atomically closed free-variable tableau for S.

2.2.4 Proof Procedures for Free-Variable Tableaux

We have proven the completeness of free-variable tableaux via a simulation of sentence tableaux instead of providing an independent completeness proof. The advantage of this approach is that we are assured that, for any atomic sentence tableau proof, there is a free-variable tableau proof of the same tree size. The disadvantage of this completeness proof, however, is that it is proof-theoretically weaker than the one given for sentence tableaux, since we do not specify *how to systematically construct* a closed free-variable tableau, as it is done with the systematic sentence tableau procedure. The simple reason for this is the following. Since the calculus of free-variable tableaux is destructive, in general, the (tree) union of the tableaux in a successive tableau sequence cannot be performed. The fundamental proof-theoretic difference from sentence tableaux is that with the application of substitutions to tableaux the paradigm of saturating a branch (possibly up to a Hintikka set) is lost. A notion of *saturated systematic free-variable tableau* can only be defined for the fragment of the calculus without the closure rule. Completeness could then be shown in the standard way by using any one-to-one association between the set of variables and the set of all ground terms which is then applied to the saturated tableaux at the end (cf. p. 195 in [Fitting, 1996]). This is proof-theoretically possible, but useless for efficient proof search, because the employment of a fixed association between variables and ground terms degrades free-variable tableaux to sentence tableaux. The question is whether there exists a systematic procedure for free-variable tableaux of the same type and functionality as for sentence tableaux but with variable instantiations guided by unification? The problem can at best be recognized with an example.

Consider the formula on top of Figure 2.6. Since the formula is a satisfiable datalogic formula, any regular Herbrand tableau construction will terminate with an open branch which is a Hintikka set. Let us contrast this with the behaviour of free-variable tableaux. Referring to the figure, after eight steps we have produced the displayed two-branch tableau. What shall we do next? If we close the left branch by unifying $\neg P(x_1, y_1, v_1)$ and the complement of $\neg P(c, a, b)$, the applied unifier blocks the immediate closure of the right branch. We could proceed and try another four γ'-steps, producing a similar situation than before. Since always new free variables are imported by the γ'-rule, the procedure never terminates, even if we only permit regular tableaux. How do we know when to stop and how can we produce a model? In fact, no systematic procedure for free-variable

$$
\begin{array}{c}
(1)\ \neg P(a,b,c) \wedge P(c,a,b) \wedge \forall x \forall y \forall v \forall w (P(x,y,v) \rightarrow P(y,x,w)) \\
\big| \, \boldsymbol{\alpha}(1) \\
(2)\ \neg P(a,b,c) \\
\big| \, \boldsymbol{\alpha}(1) \\
(3)\ P(c,a,b) \\
\big| \, \boldsymbol{\alpha}(1) \\
(4)\ \forall x \forall y \forall v \forall w (P(x,y,v) \rightarrow P(y,x,w)) \\
\vdots \, 4{*}\boldsymbol{\gamma}' \\
(8)\ P(x_1,y_1,v_1) \rightarrow P(y_1,x_1,w_1) \\
\boldsymbol{\beta}(8) \\
(9)\ \neg P(x_1,y_1,v_1) \qquad\qquad (10)\ P(y_1,x_1,w_1)
\end{array}
$$

Figure 2.6: Free-variable tableau for a datalogic formula (see Definition 2.35).

tableaux has been devised up to now that both is guided by unification and has the same functionality as sentence tableaux. It was only very recently that such a procedure has been proposed for the restricted class of formulae in clausal form [Billon, 1996]. This procedure, which is based on a nondestructive variant of free-variable tableaux, is described in Section 4.2.3.

But we will further pursue the destructive line and discuss a radically different paradigm of searching for tableau proofs. Instead of saturation of a single tableau, one considers *all* tableaux that can be constructed. If all existing tableaux are enumerated in a fair manner, for any unsatisfiable input sentence, one will eventually find a closed free-variable tableau. The fair enumeration is facilitated by the fact that the set of all existing tableaux can be organized in the form of a tree.

Definition 2.63 (Search tree) Let S be a set of sentences, C a tableau calculus, and ϕ a branch selection function. The *search tree of C and ϕ for S* is a tree $\mathcal{T}$ with its non-root nodes labelled with tableaux defined as follows.

1. The root of $\mathcal{T}$ consists of a single unlabelled node.
2. The successors of the root are labelled with all single-node tableaux for S.
3. Every non-leaf node $\mathcal{N}$ in $\mathcal{T}$ labelled with a tableau T has as many successor nodes as there are successful applications of a single inference step in C applied to the branch in T selected by ϕ, and the successor nodes of $\mathcal{N}$ in $\mathcal{T}$ are labelled with the respective resulting tableaux.

If the input set is finite, the search tree branches finitely, and a fair enumeration can be achieved by simply inspecting the search tree levelwise from the top to the leaves. Any closed tableau will eventually be found after finitely many steps. In practice, this could be implemented as a procedure which explicitly constructs competitive tableaux and thus investigates the search tree in a *breadth-first* manner. The *explicit* enumeration of tableaux, however, suffers from two severe disadvantages. The first one is that, due to the branching rate of the search tree, an enormous amount of space is needed to store all tableaux. The second disadvantage is that the cost for adding new tableaux increases during the proof process as the sizes of the proof objects increase. In contrast, for resolution procedures mainly the *number* of new proof objects (clauses) is generally considered the critical parameter. These weaknesses give sufficient reason why in practice no-one has succeeded with an explicit tableau enumeration approach up to now.

The customary and successful paradigm therefore is to perform tableau enumeration in an implicit manner, using *iterative deepening search* procedures. With this approach, iteratively increasing finite initial segments of a search tree are explored. Although, according to this methodology, initial parts of the search tree are explored several times, no significant efficiency is lost if the initial segments increase exponentially [Korf, 1985]. Due to the construction process of tableaux from the root to the leaves, many tableaux have identical or structurally identical subparts. This motivates one to explore finite initial segments of the search tree in a *depth-first* manner by strongly exploiting *structure sharing* techniques and *backtracking*. Using this approach, at each time only one tableau is kept in memory, which is extended following the branches of the search tree, and truncated when a leaf node of the respective initial segment of the search tree is reached. The advantage is that, due to the application of Prolog techniques, very high inference rates can be achieved this way (see [Stickel, 1988], [Letz et al., 1992], or [Beckert and Posegga, 1994]). The respective initial segments are determined by so-called *completeness bounds*.

Definition 2.64 (Completeness bound) A *size bound* is a total mapping s assigning to any tableau T a nonnegative integer n, the *s-size* of T. A size bound s is called a *completeness bound* for a tableau calculus C if, for any finite set S of formulae and any $n \geq 0$, there are only finitely many C-tableaux with s-size less or equal to n.

The finiteness condition qualifies completeness bounds as suitable for iterative deepening search. Given a completeness bound s and an iterative deepening level with size limit n, an implicit deduction enumeration procedure works as follows. Whenever an inference step is applied to a tableau, it is checked whether the s-size of the new tableau is $\leq n$, otherwise backtracking is performed. A common methodology of developing completeness bounds for the strict (or strictly regular) free-variable tableau calculus C is to limit the application of γ'-steps in certain ways (see also [Fitting, 1996]). We give three concrete examples. First, one may simply limit the application of γ'-steps permitted in the tableau (1) or on each branch of the tableau (2). Another variant (3) is the so-called *multiplicity* bound which has also been used in other frameworks [Prawitz, 1960] and [Bibel, 1987]. The natural definition of this bound is for finite sets S of formulae in Skolem form and for tableaux in which every $F \in S$ is used in the tableau only once at the beginning. Then, with multiplicity n, to each γ-node in the tableau, at most n γ'-steps are permitted.

It is obvious that all mentioned size bounds are completeness bounds for the tableau calculus C, that is, for any finite input set of formulae S: for every n, there are only finitely many C-tableaux of size n for S, and if S is unsatisfiable, then, for some n, there is a closed C-tableau with size n.

Interestingly, one can make complexity assessments about the problem of determining whether a tableau with a certain limit exists. For example, for the bounds (1) and (3), one can demonstrate that, for some finite input set S, the recognition problem of the existence of a closed tableau for S with a certain limit is complete for the complexity class Σ_2^p in the *polynomial hierarchy* [Garey and Johnson, 1979]. We will consider this topic later in Chapter 8.

A general disadvantage of completeness bounds of the γ-type is that they are too uniform to be useful in practice. Normally, the first initial segment of the search tree containing a closed tableau with size n may have an astronomic size, with the obvious consequence that in practice a proof will not be found. In the next section, we shall mention completeness bounds in which normally the first proof is in a much smaller initial segment.

We conclude this section with mentioning an obvious method for reducing the effort for finding closed free-variable tableaux. In fact, it is not necessary to consider *all* free-variable tableaux in an initial segment of a search tree. Since only the closure rule is destructive, we can work with the following refined calculus which, at least concerning the tableau expansion rules, is deterministic, similar to the systematic tableau procedures.

Definition 2.65 (Expansion-deterministic free-variable tableau) Expansion-deterministic free-variable tableaux are defined like systematic sentence tableaux, but with the respective free-variable rules plus the closure rule.

So the only way indeterminism can occur in this calculus is by the application of closure steps. In order to minimize the search effort, one should even prefer the closure rule (if applicable) to all expansion rules. The completeness of this refinement of free-variable tableaux immediately follows from Lemma 2.61, since the calculus can simulate the construction of any systematic sentence tableau.

As a final remark of this section, it should be emphasized that, from a search-theoretic perspective, tableau enumeration procedures are not optimally suited for confluent calculi (like the ones mentioned so far). This is because, for confluent tableau calculi, on *any* branch of the search tree there must be a closed tableau if the input set is unsatisfiable. A tableau enumeration procedure, however, does not take advantage of this proof-theoretic virtue of the calculus.

Chapter 3

Tableaux with Connections

3.1 Clausal Tableaux

The efforts in automated deduction for classical logic have been mainly devoted to the development of proof procedures for formulae in clausal form. This has two reasons. First, as discussed in Section 2, in classical logic, any first-order formula can be transformed into clausal form without affecting the satisfiability status and with only a polynomial increase of the formula size. Since, for formulae in clausal form, the tableau rules can be reduced and presented in a more condensed form, simpler and more efficient proof procedures can be achieved this way. Second and even more important, due to the uniform structure of formulae in clausal form, it is much easier to detect additional refinements and redundancy elimination techniques than for the full first-order format. This section will provide plenty of evidence for this fact.

Since in clause logic negations are only in front of atomic formulae, only atomic branch closures can occur. Accordingly, when a closed free-variable tableau for a set of clauses S is to be constructed and a clause in S has been selected for branch expansion, one can deterministically decompose it to the literal level. Such a macro step consists of a formula rule application, a sequence of γ'-steps and possibly a β-step. It is convenient to ignore the intermediate formulae and reformulate such a sequence of inference steps as a single condensed tableau expansion rule. In order to simplify the presentation, we consider only clauses without quantifier prefixes. This is no restriction, since every clause is logically equivalent to its universal closures.

Definition 3.1 (Renaming, variant) Let F be a formulae, S a set of formulae, and $\sigma = \{x_1/y_1, \ldots, x_n/y_n\}$ a substitution such that all $y_1, \ldots, y_n$ are distinct variables new to S. $F\sigma$ is called a *renaming* or *variant of* $x_1, \ldots, x_n$ *in* F *wrt* S.

Clausal tableaux are trees labelled with literals (and other control information) inductively defined as follows.

Definition 3.2 (Clausal tableau) Let S be any set of clauses. A tree consisting of just one unlabelled node is a *clausal tableau for S*. The single branch of this tree is considered as *open*. If B is a branch of a clausal tableau T for S, then the formula trees obtained from the following two inference rules are *clausal tableaux for S*:

(E) $B \oplus L_1 \mid \cdots \mid L_n$ where $L_1 \vee \cdots \vee L_n$ is a renaming of the free variables in a clause of S wrt the formulae in T; the rule (E) is called *clausal expansion rule* or just *expansion rule*. The new branches are considered as *open*. Its leaf nodes are called *subgoals*.

(C) the closure rule of free-variable tableaux, also called *reduction rule*. Now the respective branch is considered as *closed*.

Definition 3.3 (Tableau clause) For any non-leaf node N in a clausal tableau, the set of nodes $N_1, \ldots, N_m$ immediately below N is called the node *family below* N; if the nodes $N_1, \ldots, N_m$ are labelled with the literals $L_1, \ldots, L_m$, respectively, then the clause $L_1 \vee \cdots \vee L_m$ is named the *tableau clause* below N; The tableau clause below the root node is called the *start* or *top clause* of T.

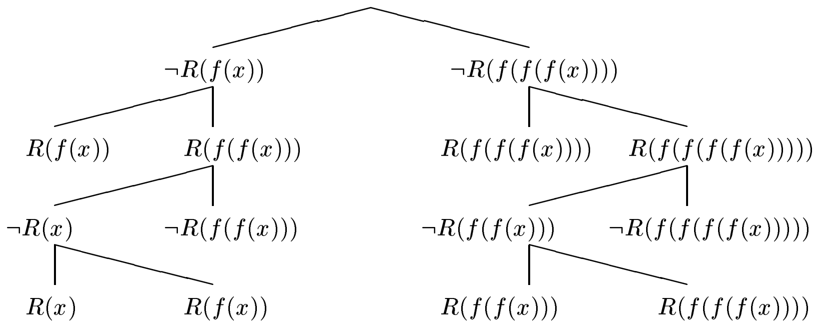


Figure 3.1: Closed clausal tableau.

In Figure 3.1, a closed clausal tableau is displayed. Here the unifiers of closure steps were already applied. The input is the set of clauses $\forall x(R(x) \vee R(f(x)))$ and $\forall x(\neg R(x) \vee \neg R(f(f(x))))$ corresponding to the negation of the formula F presented in Example 1.12. So we have demonstrated that F is valid. The format of clausal tableaux provides a relatively concise representation of tableau proofs containing all relevant information. Note that a full free-variable tableau proof including all intermediate inference steps would have more than twice the size.

In general, variables in clausal tableaux are considered as *rigid*, i.e., just as placeholders for arbitrary ground terms. In Section 6.3, it will be shown that this condition can be relaxed for certain variables. The example also shows the necessity of renaming variables. Without renaming it would be impossible to unify the second literal in the first clause with the complement of the second literal in the second clause, which is done, for example, in the second closure step on the left. Furthermore, multiple copies of the same input clauses are needed.

Let us make some remarks on the peculiarities of this definition as compared with the more familiar definition of tableaux used before. On the one hand, we carry the input set or formula S alongside the tableau and do not put its members at the beginning of the tableau. Instead we leave the root unlabelled. This facilitates the comparison of tableaux for different input sets. For example, one tableau may be an instance of another tableau, even if their input sets differ. Second, a branch is considered as closed only if the closure rule was explicitly applied to it, all other branches are considered as open, even when they are complementary. This precaution simplifies the presentation, in particular, the proof of the Lifting Lemma (Lemma 5.9), and does more adequately reflect the actual situation when implementing tableaux.

The completeness and confluence of clausal tableaux for clause formulae follow immediately from the fact that clausal tableaux can simulate free-variable tableaux, by simply omitting formula steps, β-steps, and γ'-steps, and by performing clausal expansion steps in place.

Clausal tableaux provide a large potential for refinements, i.e., for imposing additional restrictions on the tableau construction. For instance, one can integrate *ordering restrictions* (see [Klingenbeck and Hähnle, 1994]) as they are successfully used in resolution-based systems. The most important structural refinement of clausal tableau concerning automated deduction, however, is to use links or *connections* to guide the proof search.

3.2 Connection Tableaux

A closer look at the clausal tableau displayed in Figure 3.1 reveals an interesting structural property. In every node family below the start clause, at least one node has a complementary ancestor. This property can be formulated in two variants, a weaker and a stronger one.

Definition 3.4 (Path connectedness, connectedness)

1. A clausal tableau is said to be *path connected* or called a *path connection tableau* if, in every family of nodes except the start clause, there is one node with a complementary ancestor.

2. A clausal tableau is said to be *(tightly) connected* or called a *(tight) connection tableau* if, in every family of nodes except the start clause, there is one node with a complementary predecessor.

With the connection conditions, every clause has a certain relation with the start clause. This allows a goal-oriented form which may be used to guide the proof search. In Figure 3.2 the difference between the two notions is illustrated with a closed path connection tableau and a closed connection tableau for the set of propositional clauses p, $\neg p \vee q$, $r \vee \neg p$, and $\neg p \vee \neg q$. It is obvious that

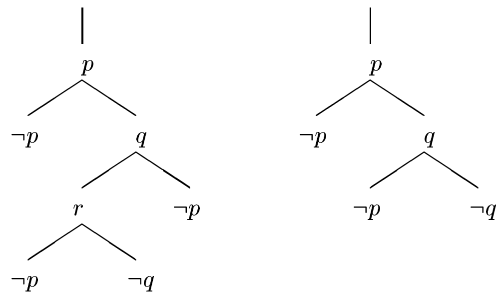


Figure 3.2: A path connection and a connection tableau.

the tight connection condition is properly more restrictive, since there exists no closed connection tableau which uses the redundant clause $r \lor \neg p$.

Let us make some brief historical remarks on the rôle of connections in tableaux. The notion of a connection is a central concept in automated deduction whereas tableau calculi, traditionally, have no reference to connections—as an illustration, note that the notion does not even occur in [Fitting, 1996]. On the other hand, it was widely not noticed in the area of automated deduction and logic programming that calculi like *model elimination* [Loveland, 1968, Loveland, 1978], *SLD-resolution* [Kowalski and Kuehner, 1970], or the *connection calculi* [Bibel, 1987] are proof-theoretically better viewed as tableau calculi. This permits, for instance, to view the calculi as *cut-free* proof systems. The relation of these calculi to tableaux has not been recognized, although, for example, the original presentation of model elimination [Loveland, 1968] is clearly in tableau style. The main reason for this may be that until recently both communities (tableaux and automated deduction) were almost separated. As a further illustration of this fact, note that unification was not really used in tableaux before the end ot the eighties [Reeves, 1987, Fitting, 1990]. In Chapter 4, we will expound the relation of connection tableaux with model elimination, SLD-resolution, and the connection method.

In order to satisfy the connection conditions, for every tableau expansion step except the first one, the closure rule has to be applied to one of the newly attached nodes. This motivates to amalgamate both inference rules into a new macro inference rule.

Definition 3.5 ((Path) extension rule) The *(path) extension rule* is defined as follows: perform a clausal expansion step immediately followed by a closure step unifying one of the newly attached literals, say L, with the complement of the literal at its predecessor node (at one of its ancestor nodes); the literal L and its node are called *entry* or *head literal* respectively *entry* or *head node*.

The building of such macro inference rules is a standard technique in automated deduction to increase efficiency. With these new rules the clausal tableau calculi may be reorganized.

Definition 3.6 ((Path) connection tableau calculus) The *(path) connection tableau calculus* consists of the following three inferences rules:

- the (path) extension rule,
- the closure or reduction rule,
- and the *start rule*, which is simply the expansion rule, but restricted to only one application, namely the attachment of the start clause.

A fundamental proof-theoretic property of the two connection tableau calculi is that they are not proof confluent, as opposed to the general clausal tableau calculus. This can easily be recognized, for instance, by considering the unsatisfiable set of unit clauses $S = \{p, q, \neg q\}$. If we select p as start clause, then the tableau cannot be completed to a closed tableau without violating the (path) connectedness condition. In other terms, using the (path) connectedness condition, one can run into dead ends. The important consequence to be drawn from this fact is that, for those tableau calculi, systematic branch saturation procedures of the type presented before do not exist. Since an open connection tableau branch that cannot be expanded does not guarantee the existence of a model, connection tableaux are therefore not suited for *model generation*. Weaker connection conditions that are compatible with model generation were developed in [Manthey and Bry, 1988, Billon, 1996, Baumgartner, 1998] and will be considered in Section 4.2.

3.3 Proof Search in Connection Tableaux

When using non-confluent deduction systems like the connection tableau calculi, in order to find a proof, in general, all possible deductions have to be enumerated in a fair manner until the first proof is found. It is important to emphasize that the search spaces of tableau calculi cannot be represented as familiar *and-or-trees* in which the and-nodes represent the tableau clauses and the or-nodes the alternatives for expansion. Such a more compact representation is not possible in the first-order case, because the branches in a free-variable tableau cannot be treated independently.

In Figure 3.3, the complete connection tableau search tree for a set of clauses is given. For this simple example, the search tree is finite. Note that the search space of the general clausal tableau calculus (without a connection condition) is infinite for S. This is but one illustration of the search pruning effect achieved by the connection conditions.

3.3.1 Completeness Bounds for Connection Tableaux

In contrast to the completeness bounds typically used for general tableaux, which are based on limitations of γ-rule applications, for connection tableaux, different completeness bounds are favourable.

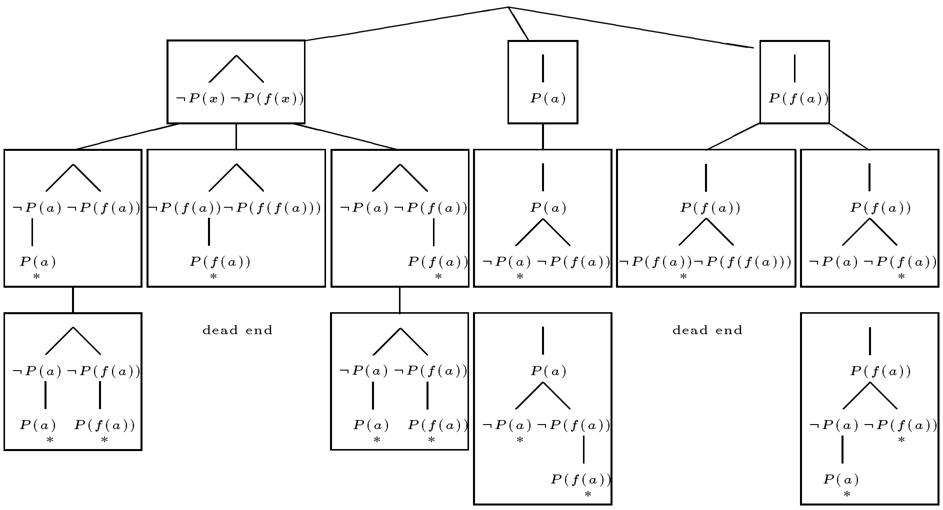


Figure 3.3: The connection tableau search tree for the set S consisting of the three clauses $\neg P(x) \vee \neg P(f(x))$, $P(a)$, and $P(f(a))$.

Inference bound

The most natural completeness bound is the so-called *inference bound* which counts the number of inference steps that are needed to construct a closed tableau. Using the inference bound, the search tree is explored level-wise; that is, for size n, the search tree is explored until depth $\leq n$. The search effort can be reduced by using *look-ahead* information as follows. As soon as a clause is selected for attachment, its length is taken into account for the current inference number, since obviously, for every subgoal of the clause at least one inference step is necessary to solve it. This enables us to detect as early as possible when the current size limit is exceeded. For example, considering the search tree given in Figure 3.3, with inference limit 2, one can avoid an expansion step with the first clause $\neg P(x) \vee \neg P(f(x)$, since any closed tableau with this clause as start clause will at least need 3 inference steps. This method was first used in [Stickel, 1988].

Depth bound

A further simple completeness bound is the *depth bound*, which limits the length of the branches of the tableaux considered in the current search level. In connection tableaux, one can relax this bound so that it is only checked when non-unit clauses are attached. This implements a certain unit preference strategy. An experimental comparison of the inference bound and the relaxed depth bound is contained in [Letz et al., 1992].

Both of the above bounds have certain deficiencies in practice. Briefly, the inference bound is too optimistic, since it implicitly assumes that subgoals which are not yet processed may be solved with just one inference step. The weakness of the depth bound, on the other hand, is that it is too coarse in the sense that the number of tableaux in a search tree with depth $\leq n+1$ is usually much larger than the number of tableaux with depth $\leq n$. In fact, in the worst case, the increase function is doubly exponential whereas, in the case of the inference bound, the increase function is exponential. Furthermore, both bounds favour tableaux of completely different structures. Using the inference bound, trees containing few long branches are preferred, whereas the depth bound prefers symmetrically structured trees.

A divide-and-conquer optimization of the inference bound

In [Harrison, 1996], the following method was applied for avoiding some of the deficiencies of the inference bound. In order to comprehend the essence of the method, assume N_1 and N_2 be two subgoals (among others) in a tableau and let the number of remaining inferences be k. Now it is clear that one of the two subgoals must have a proof of $\leq k/2$ inferences in order to respect the size limit. This suggests the following two-step algorithm. First, select the subgoal N_1 and attempt to solve it with inference limit $k/2$; if this succeeds, solve the rest of the tableau with whatever is left over from k. If this has been done for all solutions of the subgoal N_1, repeat the entire process for N_2. The advantage of this method is that the exploration of N_1 and N_2 to the full limit k is often avoided. Its disadvantage is that pairs of solutions of the subgoals with size $\leq k/2$ will be found twice, which increases the search space. In order to avoid a breaking down of this method in the recursive case, methods of failure caching as presented in Section 5.3.3 are needed. In practice, this method performs better if instead of $k/2$ smaller limits like $k/3$ or $k/4$ are used, although those do not guarantee that all proofs on the respective iterative-deepening level can be found. A possible explanation for this improved behavior is that the latter methods tend to prefer short or unit clauses which is a generally successful strategy in automated deduction (see also Section 3.4.2 where a similar effect may be achieved with a method based on a different idea).

Clause dependent depth bounds

Other approaches aim at improving the depth bound. The depth bound is typically implemented as follows. For a given tableau depth limit, say k, every node in the tableau is labelled with the value $k-d$ where d is the distance from the root node. If this value of a node is 0, then no tableau extension is permitted at this node. Accordingly, one may call this value of a node its *resource*. This approach permits a straightforward generalization of the depth bound. Instead of giving the open successors $N_1, \ldots, N_m$ of a tableau node N with resource i the resource $j = i-1$, the resource j of each of $N_1, \ldots, N_m$ is the value of a function r of two arguments, the resource i of N and the number m of new subgoals in the attached clause. We call such bounds *clause dependent depth bounds*. With clause dependent depth bounds a smoother increase of the iterative deepening levels can be achieved. Two such clause dependent depth bounds have been used in practice, one defined by $r(i, m) = i - m$ (this bound is available in the system SETHEO since version V.3 [Goller et al., 1994]) and the other by $r(i, m) = (i-1)/m$ (this bound was called *sym* in [Harrison, 1996]).

Weighted depth bounds

Although with clause dependent depth bounds, a higher flexibility can be obtained, these bounds are all in the spirit of the pure depth bound in the sense that the resource j of a node is determined at the time the node is attached to the tableau. In order to increase the flexibility and to permit an integration of features of the inference bound, the so-called *weighted depth bounds* have been developed. The main idea of the weighted depth bounds is to use a bound like the clause dependent depth bound as a basis, but to take the inferences into account when eventually allocating the resource to a subgoal. In detail, this is controlled by three parameterized functions w_1, w_2, w_3 as follows. When entering a clause with m subgoals from a node with resource i, first, the maximally available resource j for the new subgoals is computed according to a clause dependent depth bound, i.e., $j = w_1(i, m)$. Then, the value j is divided into two parts, a *guaranteed* part $j_g = w_2(j, m)$ and an *additive* part $j_a = j - j_g$. Whenever a subgoal is selected, the additive part is modified depending on the inferences Δi performed since the clause was attached to the tableau,[1] i.e., $j'_a = w_3(j_a, \Delta i)$. The eventually allocated resource for a selected subgoal then is $j_g + j'_a$.

Depending on the parameter choices for the functions w_1, w_2, w_3, the respective weighted depth bound can simulate the inference bound ($w_1(i, m) = i - m$, $w_2(j, m) = 0$, $w_3(j_a, \Delta i) = j_a - \Delta i$) or the (clause dependent) depth bound(s) or any combination of them.

A parameter selection which represents a simple new completeness bound combining inference and depth bound is, for example, $w_1(i, m) = 1$, $w_2(j, m) = j-(m-1)$, $w_3(j_a, \Delta i) = j_a/(1+\Delta i)$. On problems with equality, this bound turned out to be much more successful than each of the other bounds [Moser et al., 1997]. One reason for the success of this strategy that it also performs a unit preference strategy.

3.4 Subgoal Processing

There is a source of indeterminism in the clausal tableau calculi presented so far that can be removed without any harm. This indeterminism concerns the selection of the next subgoal at which an expansion, extension, or closure step is to be performed.

[1] We assume that the look-ahead optimization is used, according to which reduction steps and extension steps into unit clauses do not increase the current inference value. This implies that $\Delta i = 0$ if no extension steps in non-unit clauses have been performed on subgoals of the current clause.

Most complete refinements and extensions of clausal tableau calculi developed to date are *independent of the subgoal selection*, i.e., the completeness holds for any subgoal selection function (for an exception see Section 6.1.4). If a calculus has this property, then one can decide in advance for one subgoal selection function ϕ and ignore all tableaux in the search tree that are not constructed according to ϕ. This way the search effort can be reduced significantly. As an illustration of this method of *search pruning*, consider the search tree displayed in Figure 3.3. For this simple tree, one can only distinguish two subgoal selection functions ϕ_1 and ϕ_2. ϕ_1 selects the right subgoal and ϕ_2 the left subgoal in the start clause. When deciding for ϕ_1, the three leftmost lower boxes will vanish from the search tree. In case of ϕ_2, only two boxes will be pruned away.

For the clausal tableau calculi presented up to this point, even the following stronger independence property holds.

Proposition 3.7 (Strong independence of the subgoal selection) Given any closed (path) (connection) tableau T' for a set of clauses S constructed with n inference steps, then for any subgoal selection function ϕ, there exists a sequence $T_0, \ldots, T_n$ of (path) (connection) tableaux constructed according to ϕ such that T_n is closed and T' is an instance of T_n.

In case a calculus is strongly independent of the subgoal selection, not only completeness is preserved, but even minimal proof lengths. Furthermore, if a completeness bound of the sort described above is used, then the iterative-deepening level on which the first proof is found is always the same, independently of the subgoal selection. Note that the strong independence of the subgoal selection (and hence minimal proof lengths) will be lost for certain extensions of the clausal tableau calculus like *folding up* [Letz et al., 1994] and the *local* closure rule which is studied below.

One particularly useful form of choosing subgoals is *depth-first* selection, i.e., one always selects the subgoal of an open branch with maximal length in the tableau. *Depth-first left-most/right-most* selection always chooses the subgoal on the left-most/right-most open branch (which automatically has maximal depth). Depth-first left-most selection is the built-in subgoal selection strategy of Prolog. Depth-first selection has a number of advantages, the most important being that the search is kept relatively local. Furthermore, very efficient implementations are possible.

3.4.1 Subgoal Reordering

The order of subgoal selection has influences on the size of the search space, as illustrated with the search tree above. This is because subgoals normally share variables and thus the solution substitutions of one subgoal have an influence on the solution substitutions of the other subgoals.

A general *least commitment* paradigm is to prefer subgoals that produce fewer solutions. In order to identify a non-closable connection tableau as early as possible, the solutions of a subgoal should be exhausted as early as possible. Therefore, subgoals for which probably only few solutions exist should be selected earlier than subgoals for which many solutions exist. This results in the *fewest-solutions* principle for subgoal selection.[2]

Depth-first selection means that all subgoal alternatives stem from one clause of the input set. Therefore, the selection order of the literals in a clause can be determined *statically*, i.e., once and for all before starting the proof search, as in [Letz et al., 1992]. But subgoal selection can also be performed *dynamically*, whenever the literals of the clause are handled in a tableau. The static version is cheaper (in terms of performed comparisons), but often an optimal subgoal selection cannot be determined statically, as can be seen, for example, when considering the transitivity clause $P(x,z) \vee \neg P(x,y) \vee \neg P(y,z)$. Statically, none of the literals can be preferred. Dynamically, however, when performing an extension step entering the transitivity clause from a subgoal $\neg P(a,z)$, the first subgoal $\neg P(x,y)$ is instantiated to $\neg P(a,y)$. Since now it contains only one variable, is should be preferred according to the fewest-solutions principle. Entering the transitivity clause from a subgoal $\neg P(x,a)$ leads to preference of the second subgoal $\neg P(y,a)$.

3.4.2 Subgoal Alternation

When a subgoal in a tableau has been selected for solution, a number of complementary unification partners are available, viz. the connected path literals and the connected literals in the input clauses. Together they form the so-called *choice point* of the subgoal. One common principle of standard backtracking search procedures in connection tableaux (and in Prolog) is that, whenever a subgoal has been selected, its choice point must be completely finished, i.e., when retracting an alternative in the choice point of a subgoal, one has to stick to the subgoal and try another alternative in its choice point. This standard methodology has an interesting search-theoretic weakness.

This can be illustrated with the following generic example, variants of which often occur in practice. Given the subgoals $\neg P(x,y)$ and $\neg Q(x,y)$ in a tableau, assume the following clauses be in the input.

(1) $P(a,a)$,
(2) $P(x,y) \vee \neg P'(x,z) \vee \neg P'(y,z)$,
(3) $P'(a_i,a)$, $\quad 1 \leq i \leq n$,
(4) $Q(a_i,b)$, $\quad 1 \leq i \leq n$.

Suppose further we have decided to select the first subgoal and perform depth-first subgoal selection. The critical point, say at time t, is after unit clause (1) in the choice point was tried and no compatible solution instance for the other subgoal was found. Now we are forced to enter clause (2). Obviously, there are n^2 solution substitutions (unifications) for solving clause (2) (the product of the solutions of its subgoals). For each of those solutions, we have to perform n unifications with the Q-subgoal, which all fail. Including the unifications spent

[2] A special case of the fewest-solutions principle is the *first-fail* principle.

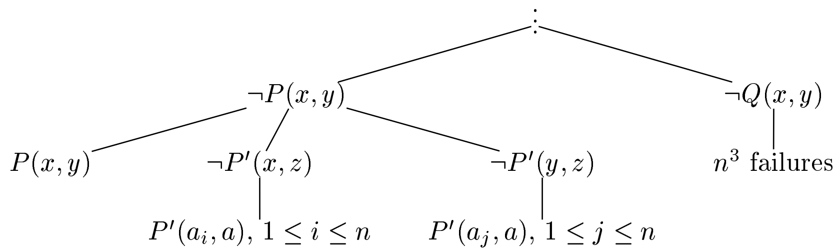


Figure 3.4: Effort in case of standard subgoal processing.

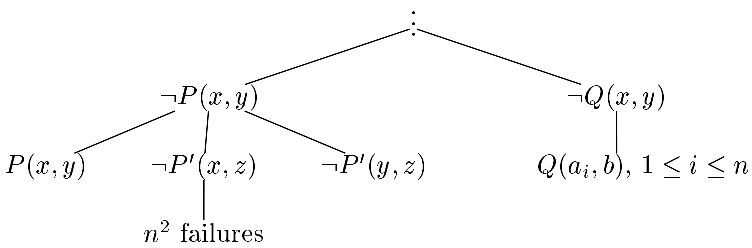


Figure 3.5: Effort when switching to another subgoal.

in clause (2), this amounts to a total effort of $1 + n + n^2 + n^3$ unifications (see Figure 1). Observe now what would happen when at time t we would not have entered clause (2), but would switch to the Q-subgoal instead. Then, for each of the n solution substitutions $Q(a_i, b)$, one would jump to the P-subgoal, enter clause (2) and perform just n failing unifications for its first subgoal. This sums up to a total of just $n + n(1 + n) = 2n + n^2$ unifications (see Figure 2).

It is apparent that this phenomenon has to do with the fewest-solutions principle. Clause (2) generates more solutions for the subgoal $\neg P(X,Y)$ than the clauses in the choice point of the subgoal $\neg Q(X,Y)$. This shows that taking the remaining alternatives of *all* subgoals into account provides a choice which can better satisfy the fewest-solution principle. The general principle of ***subgoal alternation*** is that one always switches to that subgoal with a next clause that probably produces the fewest solutions.

One might object that with a different subgoal selection, selecting the Q-subgoal first, one also could avoid the cubic effort. But it is apparent that the example could be extended such that the Q-subgoal would additionally have a longer clause as alternative, so that the total number of its solutions would be even larger than that of the P-subgoal. In this case, with subgoal alternation one could jump back to the P-subgoal and try clause (2) next, in contrast to standard subgoal selection. Another possibility of jumping to the Q-subgoal *after* having

standard backtracking	subgoal alternation
A1 B2	A1 B2
A1 B4	A1 B4
A1 B6	A1 B6
A3 B2	$\cap\!\cup$ B2 A3
A3 B4	B2 A5
A3 B6	$\cap\!\cup$ A3 B4
A5 B2	A3 B6
A5 B4	$\cap\!\cup$ B4 A5
A5 B6	$\cap\!\cup$ A5 B6

Table 3.1: Order of tried clauses for subgoals A and B with clauses of lengths 1,3,5 and 2,4,6 in their choice points, respectively. $\cap\!\cup$ indicates subgoal alternations.

entered clause (2) would be free subgoal selection. In fact, subgoal alternation under depth-first subgoal selection comes closer to standard free subgoal selection, but both methods are not identical.

The question is, when it is worthwhile to stop the processing of a choice point and switch to another subgoal? As a matter of fact, it cannot be determined in advance, how many solutions a clause in the choice point of a subgoal produces for that subgoal. A useful criterion, however, is the *shortest-clause* principle, since, in the worst case, the number of subgoal solutions coming from a clause is the product of the numbers of solutions of its subgoals.[3]

In summary, subgoal alternation works as follows. The standard subgoal selection and clause selection phases are combined and result in a single selection phase that is performed before each derivation step. The selection yields the subgoal for which the most suitable unification partner exists wrt. the number of solutions probably produced. For this, the unification partners of all subgoals are compared with each other using, for instance, the shortest-clause principle. If more than one unification partner is given the mark of best, their corresponding subgoals have to be compared due to the principles for standard subgoal selection, namely the first-fail principle and the fewest-solutions principle.

In order to compare the working of subgoal alternation (using the shortest-clause principle) with the standard non-alternating variant, consider two subgoals A and B with clauses of lengths 1,3,5 and 2,4,6 in their choice points, respectively. Table 1 illustrates the order in which clauses are tried.

Subgoal alternation has a number of interesting effects when combined with other methods in connection tableaux. First note that the method leads to the preference of short clauses. A particularly beneficial effect of preferring short clauses, especially the preference of unit clauses, is the early instantiation of variables. Unit clauses are usually more instantiated than longer clauses, because

[3] Also, the number of variables in the *calling* subgoal and in the *head* literal of a clause matter for the number of solutions produced.

they represent the "facts" of the input problem, whereas longer clauses in general represent the axioms of the underlying theory. Since normally variables are shared between several subgoals, the solution of a subgoal by a unit clause usually leads to instantiating variables in other subgoals. These instantiations reduce the number of solutions of the other subgoals and thus reduce the search space to be explored when selecting them. Advantage is also taken from subgoal alternation when combined with local failure caching considered in Section 5.3.3. Failure caching can only exploit information from *closed* sub-tableaux, thus a large number of small subproofs provides more information for caching than a small number of large sub-tableaux that cannot be closed. Since subgoal alternation prefers short clauses and hence small subproofs, the local failure caching mechanism is supported.

Subgoal alternation leads to simultaneously processing several choice points. This provides the possibility of computing *look-ahead information* concerning the minimal number of inferences still needed for closing a tableau. A simple estimation of this inference value is the number of subgoals plus the number of all subgoals in the shortest alternative of each subgoal. In general, when using standard subgoal selection, every choice point except the current one contains connected path literals and connected unit clauses, that is, the number of subgoals with the shortest alternative for each subgoal equals zero. Using subgoal alternation, at several choice points the reduction steps and the extension steps with unit clauses have already been tried, so that only the unification partners *with* subgoals are left in the choice points of several subgoals. Thus, one obtains more information about the needed inference resources than in the standard procedure. This *look-ahead information* can be used for search pruning, whenever the number of inferences has an influence on the search bound.

However, under certain circumstances alternating between subgoals may be disadvantageous. If a subgoal cannot be solved at all, switching to another subgoal may be worse than sticking to the current choice point, since this may earlier lead to the retraction of the whole clause. Obviously, this is important for ground subgoals, because they have maximally one solution substitution in the Horn case. Since ground subgoals do not contain free variables, they normally cannot profit from early instantiations achieved by subgoal alternation, i.e., switching to brother subgoals and instantiating their free variables cannot lead to instantiations within a ground subgoal. Therefore, when processing a ground subgoal, the fewest-solutions principle for subgoal selection becomes more important than the shortest-clause principle for subgoal alternation. For this reason, subgoal alternation should not be performed when the current subgoal is ground.

Chapter 4

Related Calculi and Connection Conditions

In this chapter, we illustrate that connection tableaux can be used to capture other calculi from automated deduction and we discuss calculi with weaker concepts of connectedness.

4.1 Connection Tableaux and Related Calculi

Due to the fact that tableau calculi work by building up tree structures whereas other calculi derive new formulae from old ones, the close relation of tableaux with other proof systems is not immediately evident. There exist similarities of tableau proofs to deductions in other calculi. In order to clarify the interdependencies, it is helpful to reformulate the process of tableau construction in terms of formula generation procedures. There are two natural formula interpretations of tableaux which we shall mention and which both have their merits.

Definition 4.1 The *branch formula* of a formula tree T is the disjunction of the conjunctions of the formulae on the branches of T.

Another finer view is preserving the underlying tree structure of the formula tree.

Definition 4.2 (Formula of a formula tree (inductive))

1. The *formula* of a one-node formula tree labelled with the formula F is simply F.
2. The *formula* of a complex formula tree with root N (with label F) and immediate formula subtrees $T_1, \ldots, T_n$, in this order, is $F \wedge (F_1 \vee \cdots \vee F_n)$ (or simply $F_1 \vee \cdots \vee F_n$ if N is unlabelled) where F_i is the formula of T_i, for every $1 \leq i \leq n$.

Evidently, the branch formula and the formula of a formula tree are equivalent. Futhermore, it is clear that the following proposition holds, from which, as a corollary, also follows the soundness of the method of clausal tableaux.

Proposition 4.3 If F is the (branch) formula of a clausal tableau for a set of clauses S, then F is a logical consequence of S.

With the formula notation of tableaux, one can identify a close correspondence of tableau deductions to calculi of the *generative* type. This way, the relation of tableaux with Gentzen's *sequent system* was elaborated in [Smullyan, 1968] using so-called *block tableaux*. We are interested in recognizing similarities to calculi from the area of automated deduction. For this purpose, it is helpful to only consider the *open parts* of tableaux, which we call *goal trees*.

Definition 4.4 (Goal tree) The *goal tree of* a tableau T is the formula tree obtained from T by cutting off all closed branches.

The goal tree of a tableau contains only the open branches of a tableau. Obviously, for the continuation of the refutation process, all other parts of the tableau may be disregarded without any harm.

Definition 4.5 (Goal formula)

1. The *goal formula* of any closed tableau is the falsum $\bot$.
2. The *goal formula* of any open tableau is the formula of the goal tree of the tableau.

Using the goal formula interpretation, one can view the tableau construction as a linear deduction process in which always a new goal formula is deduced from the previous one until eventually the falsum is derived. In Example 4.6, we give a goal formula deduction that corresponds to a construction of the tableau in Figure 3.1, under a branch selection function ϕ that always selects the right-most branch.

Example 4.6 (Goal formula deduction) The set of clauses $S = \{R(x) \vee R(f(x)),$ $\neg R(x) \vee \neg R(f(f(x)))\}$ has the following goal formula refutation.

$$\begin{array}{l}
\neg R(x) \vee \neg R(f(f(x))) \\
\neg R(x) \vee (\neg R(f(f(x))) \wedge R(f(f(f(x))))) \\
\neg R(x) \vee (\neg R(f(f(x))) \wedge R(f(f(f(x)))) \wedge \neg R(f(x))) \\
\neg R(x) \vee (\neg R(f(f(x))) \wedge R(f(f(f(x)))) \wedge \neg R(f(x)) \wedge R(f(f(x)))) \\
\neg R(x) \\
\neg R(f(x)) \wedge R(f(f(x))) \\
\neg R(f(x)) \wedge R(f(f(x))) \wedge \neg R(x) \\
\neg R(f(x)) \wedge R(f(f(x))) \wedge \neg R(x) \wedge R(f(x)) \\
\bot
\end{array}$$

Proposition 4.7 The goal formula of any clausal tableau T is logically equivalent to the formula of T.

4.1.1 Model Elimination Chains

Using the goal tree or goal formula notation, one can easily identify a close similarity of connection tableaux with the model elimination calculus as presented in [Loveland, 1978], which we will discuss in some more detail. Originally, model elimination was introduced as a tree-based procedure with the full generality of subgoal selection in [Loveland, 1968], although the deductive object of a tableau is not explicitly used in this paper. As Don Loveland has pointed out, the linearized version of model elimination presented in [Loveland, 1969, Loveland, 1978] was the result of an adaptation to the resolution form. Here, we treat a subsystem of model elimination without factoring and lemmata, called *weak model elimination* in [Loveland, 1978], which is still refutation-complete. The fact that weak model elimination is indeed a specialized subsystem of the connection tableau calculus becomes apparent when considering the goal formula deductions of connection tableaux. The weak model elimination calculus can be viewed as that refinement of the connection tableau calculus in which the selection of open branches is performed in a *depth-first right-most* or *left-most* manner, i.e., always the right-most (left-most) open branch has to be selected. We decide here for the right-most variant. Due to this restriction of the subgoal selection, a one-dimensional "chain" representation of goal formulae is possible in which no logical operators are necessary. The transformation from goal formulae with depth-first right-most selection function to model elimination chains works as follows. To any goal formula generated with a depth-first right-most selection function, apply the following operation: replace every conjunction $L_1 \wedge \cdots \wedge L_n \wedge F$ with $[L_1 \cdots L_n]F$ and delete all disjunction symbols.

In a model elimination chain, the occurrences of bracketed literals denote the non-leaf nodes and the occurrences of unbracketed literals denote the subgoals of the goal tree of the tableau. For every subgoal N corresponding to an occurrence of an unbracketed literal L, the bracketed literal occurrences to the left of L encode the ancestor nodes of N. The model elimination proof corresponding to the goal formula deduction given in Example 4.6 is depicted in Example 4.8.

Example 4.8 (Model elimination chain deduction) The set consisting of the two clauses $R(x) \vee R(f(x))$ and $\neg R(x) \vee \neg R(f(f(x)))$ has the following model elimination chain refutation.

$$
\begin{array}{l}
\neg R(x)\ \neg R(f(f(x))) \\
\neg R(x)\ [\ \neg R(f(f(x)))\ ]\ R(f(f(f(x)))) \\
\neg R(x)\ [\ \neg R(f(f(x)))\ R(f(f(f(x))))\ ]\ \neg R(f(x))) \\
\neg R(x)\ [\ \neg R(f(f(x)))\ R(f(f(f(x))))\ \neg R(f(x)))\ ]\ R(f(f(x))) \\
\neg R(x) \\
[\ \neg R(f(x))\ ]\ R(f(f(x))) \\
[\ \neg R(f(x))\ R(f(f(x)))\ ]\ \neg R(x) \\
[\ \neg R(f(x))\ R(f(f(x)))\ \neg R(x)\ ]\ R(f(x)) \\
\bot
\end{array}
$$

It is evident that weak model elimination is a refinement of the connection tableau calculus. Viewing chain model elimination as a tableau refinement has various proof-theoretic advantages concerning generality and the possibility of defining extensions and refinements of the basic calculus. Also the soundness and completeness proofs of chain model elimination are immediate consequences of the soundness and completeness proofs of connection tableaux, which are very short and simple if compared with the rather involved proofs in [Loveland, 1978]. Subsequently, we will adopt the original and more general view of model elimination as intended by Don Loveland [Loveland, 1968] and use the terms connection tableaux and model elimination synonymously.

It is straightforward to recognize that SLD-resolution, although traditionally introduced as a resolution refinement, can also be viewed as a restricted form of model elimination, in which simply reduction steps are omitted. If the underlying formula is a *Horn formula*, i.e., contains only Horn clauses, then it is obvious that this restriction on model elimination preserves completeness.

4.1.2 The Connection Method

Another framework in automated deduction which is related with tableaux is the *connection method.* Based on work by Prawitz [Prawitz, 1960, Prawitz, 1969], the connection method was introduced by Andrews [Andrews, 1981] and Bibel [Bibel, 1981, Bibel, 1987]—we shall use Bibel's terminology as reference point. In contrast to the tableau framework, the kernel of the connection method is not a deductive system, but a declarative syntactic characterization of logical validity or inconsistency. While, in the original papers, the connection method represents logical validity directly, we work with the dual variant representing inconsistency, which makes no difference concerning the employed notions and mechanisms. Furthermore, we work on the clausal case only. It is essential for the presentation of this method that different occurrences of a literal in a clause and a formula can be distinguished. Occurrences of literals in a clausal formula are denoted with triples $\langle L, i, j\rangle$ where L is the literal with unique identifier i in the clause c_j in F.

Definition 4.9 (Path, connection, mating, spanning) Given a set of clauses $S = \{c_1, \ldots, c_n\}$, a *path through* S is a set of n literal occurrences in S, exactly one from each clause in S. A *connection in* S is a two-element subset $\{\langle K, i, k\rangle, \langle L, j, l\rangle\}$ of a path through S such that K and L are literals with the same predicate symbol, one negated and one not. Any set of connections M in S is called a *mating in* S; the pair $\langle M, S\rangle$ is termed a *connected formula.* A mating M is said to be *spanning for* S if every path through S is a superset of a connection in M.

A set of propositional clauses S is unsatisfiable if and only if there is a spanning mating for S. In the first-order case, the notions of multiplicities and unification come into play.

Definition 4.10 (Multiplicity, unifiable connection, mating) First, a *multiplicity* is just a mapping $\mu : \mathbb{N} \longrightarrow \mathbb{N}_0$ which is then extended to clausal formulae, as follows. Given a multiplicity μ and two sets of clauses $S = \{c_1, \ldots, c_n\}$ and $S' = \{c_1^1, \ldots, c_1^{\mu(1)}, \ldots, c_n^1 \ldots, c_n^{\mu(n)}\}$ where every c_i^k is a variable-renamed variant of c_i, we call S' a (μ-)*multiplicity* of S. A connection $\{\langle K, i, k\rangle, \langle L, j, l\rangle\}$ is termed *unifiable* if the atoms of K and L are. A mating is *unifiable* if there is a simultaneous unifier for all its connections.

Theorem 4.11 A set of clauses S is unsatisfiable if and only if there is a unifiable spanning mating for a multiplicity of S. [Bibel, 1987]

Obviously, it is decidable whether a set of clauses has a unifiable and spanning mating. In Chapter 8 we will consider the complexity of this decision problem.

4.1.3 Matings-based Connection Procedures

The just mentioned decidability suggests a two-step methodology of *iterative-deepening* proof search, as performed with the connection tableau procedures. The outer loop is concerned with increasing the multiplicity whereas the inner procedure explores the finite search space determined by the given multiplicity. Although there are different methods for identifying a unifiable and spanning mating for some multiplicity of a formula, one of the most natural ways is exemplified with a procedure which is similar to the connection tableau calculi, but without a renaming of the clauses in the given multiplicity.

Definition 4.12 ((Path) connection tableau calculus without renaming) The two calculi are the same as the ones given in Definition 3.6 except that

1. they work on a multiplicity S' of the input set,
2. no renaming is permitted in a (path) extension step, and
3. the computed substitution σ is also applied to the clauses in S'.

How are the matings concepts related with tableaux? This is obvious for the calculi without renaming, since the clause copies to be used in a tableau are determined in advance. Whenever a (path) extension or a reduction step is performed in the construction of a tableau for a multiplicity S' which involves two tableau nodes N and N' with corresponding literal occurrences $\langle K, i, k\rangle$ and $\langle L, j, l\rangle$, respectively, then we say that the connection $\{\langle K, i, k\rangle, \langle L, j, l\rangle\}$ in S' is *used* in the tableau. With *the mating* of a tableau for S' we mean the set of connections in F' used in the construction of the tableau. For connection tableau calculi with renaming, one can also define the *mating of* a tableau, the only difference being that the set of clause copies may increase during the tableau construction.

The connection procedure C_1^1 on pages 108f. in [Bibel, 1987], for example, is based on the path connection tableau calculus without renaming. However, there is a fundamental difference between the two types of calculi, the ones with and the ones without renaming. For the calculi without renaming, it is guaranteed that there are only finitely many regular or strict (path) connection tableaux for each multiplicity of the input formula—in terms of the connection calculus, strictness means that no literal occurrence appears more than once on a branch in the tableau. As a consequence, no additional limit on the tableau complexity has to be given to assure termination of the tableau search procedure for a given multiplicity.

The crucial difference of this type of tableau calculi from the ones of the previous section is that with multiplicity-based bounds static complexity limits are put on the input multiplicity whereas the tableau complexity is not directly bounded. As a matter of fact, when using strictness or regularity, also the depth of the tableaux is bounded, viz. by the number of clauses in the input multiplicity. But one may safely conjecture that the use of the pure multiplicity bound is much too coarse in order to be successful in practice (think of a multiplicity with hundreds of clauses). Instead one may limit the *cardinality of the matings* to be considered or the *number* of clauses in a multiplicity, which should work better in practice. Another drawback of multiplicity-based bounds is that certain search pruning techniques and extensions of the calculus are not as effective as for completeness bounds based on tableau complexity. This will be discussed at the end of Section 5.3.3 and Section 6.1.5.

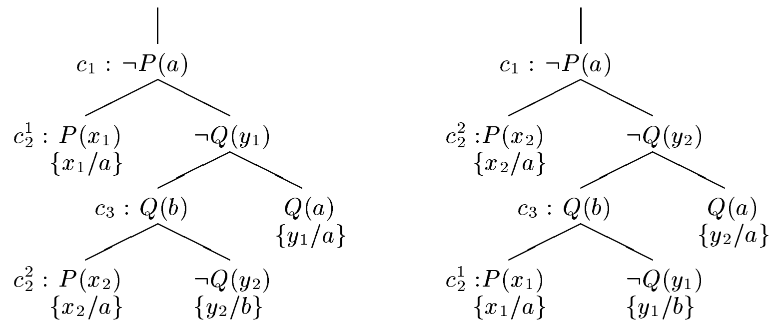


Figure 4.1: Multiplicity-based duplication of connection tableaux.

Furthermore, in the multiplicity-based case, there is a source of redundancy which has directly to do with the use of multiplicities. Consider the clausal formula

$$F = \neg P(a) \wedge (P(x) \vee \neg Q(y)) \wedge (Q(a) \vee Q(b)).$$

In Figure 4.1, two closed connection tableaux for the multiplicity $F' = c_1 \wedge c_2^1 \wedge c_2^2 \wedge c_3$ of F are displayed; also, the attached clause variants and the substitutions to be applied in the inference steps are given. It is evident that the two tableaux are variants of each other obtainable by exchanging the positions of the clauses c_2^1 and c_2^2. A naive procedure would simply generate both tableaux. Fortunately, this obvious redundancy can be avoided by using the proviso that any variant c_i^{k+1}, $k > 0$, of an initial clause c_i can be selected for extension only if the clause variant c_i^k in the multiplicity was already used in the tableau. This way the redundant tableau on the right cannot be constructed any more.

4.2 Other Clausal Tableau Calculi

In this section, we will review the most important other refinements of clausal tableaux developed so far. After a short exposition of the so-called restart variants of model elimination, we describe in some more detail two confluent and nondestructive restrictions of clausal tableaux. Both calculi permit non-enumerative proof search methods like in the systematic tableau procedures, but with the instantiation rule more or less guided by unification. Both calculi are nondestructive and hence permit the application of methods for branch saturation. There are also very recent initiatives of developing non-enumerative proof search methods for confluent but destructive clausal tableau calculi [Baumgartner et al., 1999], for which it is too early to give an assessment of their suitability for automated deduction.

4.2.1 Restart Model Elimination

In connection tableaux, a clause can be entered at any literal in an extension step, and at any node an extension step can be applied. The restart model elimination calculi [Loveland, 1991, Baumgartner and Furbach, 1998] restrict this possibility. In the basic version of restart model elimination, at positive literals no extension steps are permitted. In order to restore completeness, an alternative inference possibility is needed. This is the so-called restart step. A restart step is simply a tableau expansion step applied at subgoals with positive literals. This asymmetric treatment of literals is motivated by arguments from logic programming, where clauses also have a procedural reading. Disallowing extension steps at positive literals means disallowing the entering of a clause at a negative literal. This fits with the view that procedures with negated predicates seem not to make sense. Here only the suitability for automated proof search is of interest. The weak point of restart model elimination is that it is not compatible with regularity, only a *blockwise* regularity condition between two restart steps can be used.

Restart model elimination can even be sharpened to the extent that, for every clause, one can distinguish exactly one positive literal (if present) at which the clause may be entered. Further refinements of restart model elimination are described in [Baumgartner and Furbach, 1998]. The head selection function can even be generalized to arbitrary literals, but this requires a further weakening of the connection condition and that more clauses have to be employed for restart steps [Hähnle and Pape, 1997].

The competitiveness of all those approaches for automated deduction has not yet been demonstrated convincingly.

4.2.2 Hyper Tableaux

Beginning with the first paper [Manthey and Bry, 1988], in which the term "tableau" was not used, a number of hyper tableau calculi have been developed in the last years [Baumgartner, 1998]. A common characteric of all those systems is that they are based on a macro inference rule of the following form.

Definition 4.13 (Hyper extension rule) The *hyper extension rule* is just an expansion step immediately followed by reduction steps at all newly attached nodes that are labelled with negative literals.

It is important to note that such an inference step is possible only if really all negative literals in the attached clause can be closed by reduction steps. First, we formulate a very general version of a hyper tableau calculus, which will be specialized subsequently.

Definition 4.14 (General hyper tableaux) The *general hyper tableau calculus* consists just of the hyper extension rule.

This calculus is complete and compatible with the regularity restriction. In contrast to the connection tableau calculi, however, the calculus is also confluent and so will be all versions of hyper tableaux considered later on. Recall that confluence means that no proof enumeration is necessary, since any proof attempt can eventually be completed to a closed tableau. The problem of the general hyper tableau calculus, however, is to find a closed tableau without performing tableau enumeration, i.e., to find a strategy of applications of the general hyper extension rule which guarantees the generation of a closed tableau for any unsatisfiable clause set. As we will see, there are different solutions to this end.

Historically, the hyper tableau calculus was designed for sets of *range restricted* clauses only [Manthey and Bry, 1988].

Definition 4.15 (Range restrictedness) A clause is called *range restricted* if every variable occurring in a positive literal of the clause occurs in a negative literal of the clause.

Example 4.16

$$\begin{array}{rcl} & & P(a) \lor Q(b), \\ \neg P(x) & \lor & P(f(x)) \lor Q(f(x)), \\ \neg Q(x) & \lor & P(x) \lor R(x), \\ \neg P(b), & & \\ \neg R(a), & & \\ \neg P(f(x)), & & \\ \neg P(x) \lor \neg Q(f(x)). & & \end{array}$$

In Figure 4.2, a general hyper tableau for a set of range restricted clauses is given. Observe that the tableau is open, but no further hyper extension step can be applied without violating the regularity condition, i.e., the tableau is *saturated*. Consequently, by the completeness and the confluence of the calculus, the input set must be satisfiable. This shows the *reductive* power of the calculus, which renders it promising for proof search.

A further important property illustrated with the given example is that, in any hyper extension step, the attached clause is ground. This property is an obvious consequence of the range restrictedness of the input clauses, which has the followed further consequence.

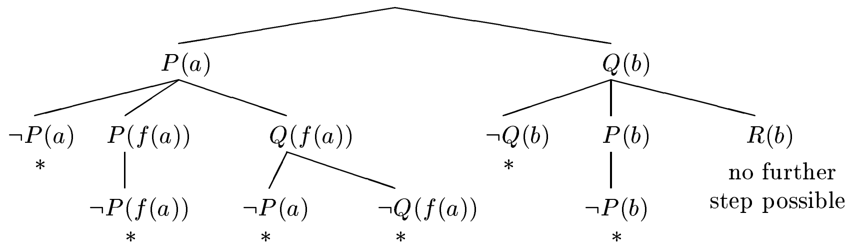


Figure 4.2: Saturated hyper tableau for Example 4.16.

Proposition 4.17 For sets of range restricted clauses, the general hyper tableau calculus is nondestructive.

The importance of the nondestructiveness of a tableau calculus has already been emphasized in Section 2.1. It permits the definition of a saturated systematic tableau. Evidently, the general hyper tableau clauses becomes destructive for clauses which are not range restricted. Consider, for example, the clause set $\{P(x,y) \vee P(y,x), \neg P(a,a)\}$. In order to generate a closed tableau, the variables in the top clause have to be instantiated. The problem is what to do with clauses like $\{P(x,y) \vee P(y,x)$ that are not range restricted. The original proposal was to just use the original γ-rule of sentence tableau on all variables remaining in a clause after a hyper extension step. As a matter of fact, only terms of the Herband universe of the input clause set need to be used [Bry and Yahya, 1996].[1]

Definition 4.18 (Hyper tableaux) The *hyper tableau calculus* consists of the hyper extension rule which is augmented in the following manner: immediately after each hyper extension step the variables in the new clause have to be instantiated to ground terms from the Herbrand universe of the input set.

It is clear that, with this modification, the hyper extension rule loses a lot of its attractiveness for automated proof search, since, in general, the rule permits infinitely many applications for any clause that is not range restricted. Yet, hyper tableaux have some interesting properties. For example, we have the following result.

Proposition 4.19 If S is a finite set of clauses without functions symbols of arity > 0, then every hyper tableau for S is finite.

This means that hyper tableaux provide a decison procedure for the class of datalogic formulae and for the Bernays-Schönfinkel class, i.e., the set of prenex formulae of the form $\exists^*\forall^*\Phi$ where Φ is quantifier-free and contains no function symbols.

[1] One can also transform any clause set containing clauses which are not range restricted into a range restricted clause set by introducing a new domain predicate [Manthey and Bry, 1988]. But this transformation just amounts to encoding the γ-rule in the input set in a data-oriented fashion.

For formulae containing proper function symbols, however, the instantiation problem is more or less similar to the one for sentence tableaux. This accounts for the fact that, for general first-order theorem proving, hyper tableaux are not generally successful. This weakness can be remedied to a certain extent by exploiting the technique of local variables which is developed in Section 6.3.

4.2.3 Disconnection Tableaux

The disconnection tableau calculus was developed in [Billon, 1996]. In order to comprehend the method, it is helpful to first describe the *clause linking* mechanism [Lee and Plaisted, 1992], a method which is in the spirit of the first theorem proving procedures developed in the sixties. Historically, the first theorem proving systems where direct applications of Herbrand's approach to proving the completeness of first-order logic (see [Davis and Putnam, 1960]). Such *Herbrand procedures* consist of two subprocedures, a generator for sets of ground instances and a propositional decision procedure. For some decades this methodology was not pursued in automated deduction, mainly because no efficient method of ground instance generation existed. The linking mechanism, and particularly its hyper linking variant [Lee and Plaisted, 1992], represents an ingenious method of integrating *unification* into the process of ground instantiation.

Definition 4.20 (Linking instance) Given a set of clauses S, let L be a literal in a clause $c \in S$ and K a renaming wrt. c of another literal in a clause of S. If there exists a unifier σ for L and $\sim K$, then the clause $c\sigma$ is called a *linking instance of c wrt. S*.

Instead of guessing arbitrary ground instances of clauses as in the theorem proving procedures of the sixties, one can iteratively form linking instances of the clause set.[2] Since this process will not automatically result in ground clauses, one has to slightly generalize the traditionally used propositional decision procedures.

Definition 4.21 (Ground satisfiability) Given a set S of clauses and a ground term t, let $S(t)$ denote the set of clauses obtained by replacing every variable in S uniformly with the term t. S is called *ground satisfiable wrt. t* if $S(t)$ is propositionally satisfiable.

The *clause linking method* simply consists in forming linking instances of the currently generated set of clauses and, from time to time, testing the current set S for ground satisfiability wrt. some arbitrary ground term t. If the satisfiability test fails, i.e., if $S(t)$ is propositionally unsatisfiable, then this obviously demonstrates the unsatisfiability of the original set of clauses. And when the term t is taken from the Herbrand universe of S, then every clause in the final propositional set $S(t)$ is a *Herbrand instance* of a clause in the original clause set. This method is complete, i.e., for any unsatisfiable clause set S_0 and for any ground term t

[2] The *hyper linking* variant requires that *each* literal in the clause c is unified with the renaming of some literal from the clause set.

(not necessarily occurring in S_0), there is a finite sequence of clause linking steps producing a clause set S such that $S(t)$ is propositionally unsatisfiable. In order to *find* such a clause set S it suffices to require that the linking instances be generated in a fair manner. Fairness simply means that, for any generated clause set S and every clause $c \in S$, every linking instance of c wrt. S will eventually be generated if no ground satisfiability tests are performed.

An important property of the linking method is that only one variant of a clause c needs to be kept, all other clauses which are renamings of c may be deleted. With this powerful deletion strategy, the method decides the class of datalogic formulae and hence the Bernays-Schönfinkel class.

One remaining weakness of the clause linking method is that the propositional decision procedure is completely separated from the generation of the linking instances. And the interfacing problem between the two subroutines may lead to tremendous inefficiencies. This has motivated the development of the disconnection method, which provides an intimate integration of the two subroutines [Billon, 1996]. The integration is achieved by embedding the linking process into a tableau guided control structure. As a further result of this embedding, the number of linking instances of clauses can be significantly reduced.

Definition 4.22 (Disconnection tableau calculus) Given an initial set S of clauses, the *disconnection tableau calculus* has two inference rules for tableau construction:

- the expansion rule applicable to clauses in S.
- the *path linking rule* which consists in the following operation. Let B be a tableau branch containing two nodes N and N' labelled with literals K and L, respectively, such that there is a unifier σ for K and a variant of $\sim L$ wrt. the tableau clause c of N, then attach the tableau clause $c\sigma$ at the leaf of B.

A tableau T is *ground closed for* some ground term t if the tableau $T\tau$ is closed where τ maps any variable appearing in T to the term t.

A disconnection tableau refutation of a clause set S is a pair $\langle T, t \rangle$ consisting of a disconnection tableau for S and a ground term t such that $T\tau$ is closed. This calculus is also compatible with the variant restriction, i.e., one can require that no two clauses which are variants of each other must occur on a branch. Let us illustrate the reductive power of this calculus with an example also used in [Billon, 1996]. Given the satisfiable set of the two clauses

$$P(x) \lor Q(x) \text{ and } \neg P(f(y)) \lor \neg Q(y),$$

one can apply the expansion rule twice resulting in the tableau shown on the left-hand side of Figure 4.3. Now, to the left-most branch only one path linking step can be applied, resulting in the tableau on the right-hand side of the figure in which the left-most branch is now ground closed. The new branch with the leaf literal $Q(f(z))$ can no more be expanded under the variant restriction and we can terminate. Since the tableau is not ground closed, the input formula must be satisfiable. The other tableau calculi considered so far do not terminate for this input set. Also, saturation based procedures like resolution do not terminate unless ordering restrictions are used.

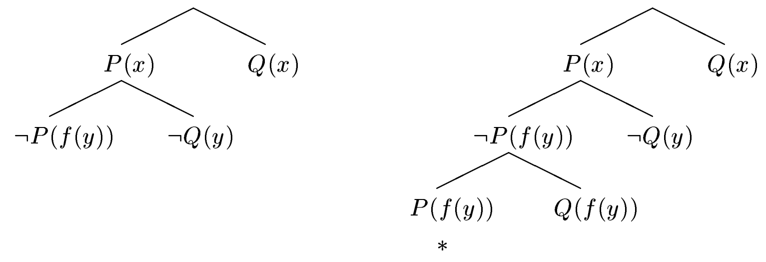


Figure 4.3: Saturated disconnection tableau for $\{P(x) \vee Q(x), \neg P(f(y)) \vee \neg Q(y)\}$.

Chapter 5

Search Pruning in Connection Tableaux

The pure calculus of connection tableaux is only moderately successful in automated deduction. This is because the corresponding search trees are still full of redundancies. In general, there are two different paradigms for reducing the effort of tableau search procedures. On the one hand, one may attempt to find further completeness-preserving restrictions on the individual proof objects, the tableaux. On the other hand, one may work on the level of the search space and try to demonstrate the redundancy of a certain tableau T, not because T has certain structural deficiencies, but because of the existence of another (better) tableau in the search space.

5.1 Structural Refinements of Connection Tableaux

First, we consider methods which attempt to restrict the tableau *calculus*, that is, disallow certain inference steps if they produce tableaux of a certain *structure*—note that the connection condition is such a structural restriction on general clausal tableaux. The effect on the tableau search tree is that the respective nodes together with the dominated subtrees can be ignored so that the branching rate of the tableau search tree decreases. These *structural* methods of redundancy elimination are *local* pruning techniques in the sense that they can be performed by looking at single tableaux only.

5.1.1 Regularity

A fundamental structural refinement of connection tableaux is the so-called regularity condition, which was already introduced in Section 2.1. Recall that a tableau is *regular* if no literal occurs more than once on a branch. The term "regular" has been introduced to emphasize the analogy to the definition of *regular* resolution [Tseitin, 1970]. Imposing the regularity restriction has some important unexpected consequences. As opposed to general clausal tableaux, where regularity preserves minimal proof lengths, minimal closed connection tableaux may not be regular. In Chapter 7 it will be shown that regular connection tableaux cannot even polynomially simulate connection tableaux. Nevertheless, a wealth of experimental results clearly shows that this theoretical disadvantage is more than compensated for by the strong search pruning effect of regularity [Letz et al., 1992], so that this refinement is indispensable for any practical proof procedure based on connection tableaux.

5.1.2 Tautology Elimination

Normally, it is a good strategy to eliminate certain clauses from the input set which can be shown to be redundant for finding a refutation. Tautological clauses are of such a sort.[1] In the ground case, tautologies may be identified once and for ever in a preprocessing phase and can be eliminated before starting the actual proof search. In the first-order case, however, it may happen that tautologies are generated dynamically. Let us illustrate this phenomenon with the example of the clause $\neg P(x, y) \vee \neg P(y, z) \vee P(x, z)$ which expresses the transitivity of a relation. Suppose that during the construction of a tableau this clause is used in an extension step (for simplicity renaming is neglected). Assume further that after some subsequent inference steps the variables y and z are instantiated to the same term t. Then a tautological instance $\neg P(x, t) \vee \neg P(t, t) \vee P(x, t)$ of the transitivity formula has been generated. Since no tautological clause is relevant in a set of formulae, connection tableaux with tautological tableau clauses need not be considered when searching for a refutation. Therefore the respective tableau and any extension of it can be disregarded.

Interestingly, the conditions of tautology-freeness and regularity are partially overlapping. Thus the non-tautology condition, on the one hand, does cover all occurrences of identical predecessor nodes, but not the more remote ancestors. The regularity condition, on the other hand, captures all occurrences of tautological clauses for backward reasoning with Horn clauses (i.e. with negative start clauses only), but not for non-Horn clauses.

5.1.3 Tableau Clause Subsumption

An essential pruning method in resolution theorem proving is *subsumption deletion*, which during the proof process deletes any clause that is subsumed by another clause, and this way eliminates a lot of redundancy. Although no new clauses are generated in the tableau approach, a restricted variant of clause sub-

[1] Although tautologies may facilitate the construction of smaller tableau proofs, since they can be used to simulate the cut rule. But one certainly wants to avoid an uncontrolled use of cuts.

sumption reduction can be exploited in the tableau framework, too. First, we shortly recall the definition of subsumption between clauses.

Definition 5.1 (Subsumption for clauses) Given two clauses c_1 and c_2, we say that c_1 ***subsumes*** c_2 if there is a variable substitution σ such that the set of literals contained in $c_1\sigma$ is a subset of the set of literals contained in c_2.

Similar to the dynamic generation of tautologies, it may happen, that a clause which has been attached in a tableau step during the tableau construction process is instantiated and then subsumed by another clause from the input set. To give an example, suppose the transitivity clause from above and a unit clause $P(a, b)$ be contained in the input set. If now the transitivity clause is used in a tableau and after some inference steps the variables x and z are instantiated to a and b, respectively, then the resulting tableau clause $\neg P(a, y) \vee \neg P(y, b) \vee P(a, b)$ is subsumed by $P(a, b)$. Obviously, for any closed tableau using the former tableau clause a closed tableau exists which uses the latter instead.

Again there is the possibility of a pruning overlap with the regularity and the non-tautology conditions. Note that, strictly speaking, the avoidance of tableau clause subsumption is not a pure ***tableau structure*** restriction, since a case of subsumption cannot be defined by merely looking at the tableau. Additionally, it is necessary to take the respective input set into account.

5.1.4 Strong Connectedness

When employing an efficient transformation from the general first-order format to clausal form, one has sometimes to introduce new predicates which are used to abbreviate certain formulae [Eder, 1985, Plaisted and Greenbaum, 1986, Boy de la Tour, 1990]. Assume, for instance, we have to abbreviate a conjunction of literals $a \wedge b$ with a new predicate d by introducing a biconditional $d \leftrightarrow a \wedge b$. This rewrites to the three clauses $\sim a \vee \sim b \vee d$, $a \vee \sim d$, and $b \vee \sim d$. Interestingly, every resolvent between the three clauses is a tautology. Applied to the tableau construction, this means that whenever one of these clauses is immediately below another one, then a hidden form of a tautology has been generated as shown in Figure 5.1. This example also illustrates that with definitions one can simulate the effect of the cut rule.

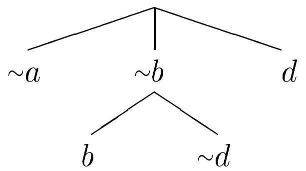


Figure 5.1: Hidden tautologies in tableaux.

Interestingly, certain cases of such hidden tautologies may be avoided. For this purpose, in [Letz, 1993a] the notion of connectedness was strengthened to ***strong*** *connectedness.*

Definition 5.2 (Strong connectedness) Two clauses c_1 and c_2 are *strongly connected* if there is a substitution σ such that the union of the sets of literals in $c_1\sigma$ and $c_2\sigma$ contains exactly two complementary literals, i.e., c_1 and c_2 have a non-tautological resolvent.

It is evident that strong connectedness is a sharpening of the method of tautology elimination from one to two tableau clauses. In Section 5.2, it will be proven that, for all pairs of adjacent tableau clauses, strong connectedness may be demanded without losing completeness. However, is is essential that the two clauses be adjacent, i.e., one must be located immediately below the other. For more distant pairs of tableau clauses one dominated by the other, the condition that they be strongly connected is not compatible with the condition of regularity. A straightforward counter-example is the set of the four clauses

$$\{\neg p \vee \neg q, \neg p \vee q, p \vee \neg q, p \vee q\}.$$

Regardless with which clause we start, we always need a "tautological" connection. This holds even when additional inference rules like factorization or folding up (see Section 6.1) are available.

5.1.5 Use of Relevance Information

By using *relevance information*, the set of possible start clauses can be minimized.

Definition 5.3 (Essentiality, relevance, minimal unsatisfiability) A formula F is called *essential in* a set S of formulae if S is unsatisfiable and $S \setminus \{F\}$ is satisfiable. A formula F is named *relevant in* S if F is essential in some subset of S. An unsatisfiable set of formulae S is said to be *minimally unsatisfiable* if each formula in S is essential in S.

As will be shown in Section 5.2, the connection tableau calculus is complete in the strong sense that, for every relevant clause in a set S, there exists a closed connection tableau for S with this clause as start clause. Since in any unsatisfiable set of clauses, some negative clause is relevant, by default one may start with negative clauses only. The application of this default pruning method achieves a significant reduction of the search space. In many cases, however, one has even more information concerning the relevance of certain clauses. Normally, a satisfiable subset of the input is well-known to the user, namely, the clauses specifying the theory axioms and the hypotheses. Such relevance information is also provided in the TPTP library [Sutcliffe et al., 1994]. A goal-directed system can enormously profit from the relevance information by considering only those clauses as start clauses that stem from the conjecture. As an example, consider an axiomatization of set theory containing the basic axiom that the empty set contains no set, which is normally expressed as a negative unit clause. Evidently, it is not very reasonable to start a refutation with this clause.

It is important to note, however, that when relevance information is being employed, then *all* conjecture clauses have to be tried as start clauses and not only the all-negative ones. Relevance information is normally more restrictive than the default method except when *all* negative clauses are stemming from the conjecture, in which case obviously the default mode is more restrictive.

5.2 Completeness of Connection Tableaux

Let us turn now to the completeness proof of connection tableaux incorporating the structural refinements of regularity, strong connectedness, and the use of relevance information. The possibility of eliminating tautologies and subsumed tableau clauses is evident and will not be considered explicitly. Since the path connectedness condition is properly less restrictive, it suffices to consider the full connection condition. Unfortunately, we cannot proceed as in the case of free-variable tableaux where a direct simulation of sentence tableaux was possible, just because of the lacking confluence. Instead, an entirely different approach for proving completeness will be necessary. The proof we give here consists of two parts. In the first part, we demonstrate the completeness for the case of ground formulae—this is the interesting part of the proof. In the second part, this result is lifted to the first-order case by a simulation technique similar to the one used in the proof of Lemma 2.61. Beforehand, we need some additional terminology.

Definition 5.4 (Strengthening) The *strengthening* of a set of clauses S *by* a set of literals $P = \{L_1, \ldots, L_n\}$, written $P \rhd S$, is the set of clauses obtained by first removing all clauses from S containing literals from P and afterwards adding the n unit clauses $L_1, \ldots, L_n$.

Example 5.5 For the set of propositional clauses $S = \{p \vee q, p \vee s, \neg p \vee q, \neg q\}$, the strengthening $\{p\} \rhd S$ is the set of clauses $\{p, \neg p \vee q, \neg q\}$.

Clearly, every strengthening of an unsatisfiable set of clauses is unsatisfiable, too. In the ground completeness proof, we will make use of the following further property.[2]

Lemma 5.6 (Strong Mate Lemma) Let S be an unsatisfiable set of ground clauses. For any literal L contained in any relevant clause c in S there exists a clause c' in S such that

(i) c' contains $\sim L$,

(ii) every literal in c' different from $\sim L$ does not occur complemented in c, and

(iii) c' is relevant in the strengthening $\{L\} \rhd S$.

[2]In terms of resolution, it expresses the fact that, for any literal L in a clause c that is relevant in a clause set S, there exists a non-tautological resolvent "over" L with another relevant clause in S.

Proof From the relevance of c follows that S has a minimally unsatisfiable subset S_0 containing c; every formula in S_0 is essential in S_0. Hence, there is an Herbrand interpretation $\mathcal{H}$ for S_0 with $\mathcal{H}(S_0 \setminus \{c\}) = \top$ and $\mathcal{H}(c) = \bot$, i.e., $\mathcal{H}$ assigns $\bot$ to every literal in c, hence $\mathcal{H}(L) = \bot$. Define another Herbrand interpretation

$$\mathcal{H}' = \begin{cases} \mathcal{H} \cup \{L\} & \text{if } L \text{ is an atomic formula} \\ \mathcal{H} \setminus \{\sim L\} & \text{otherwise} \end{cases}$$

using Notation 1.65 introduced on Page 26. By construction, $\mathcal{H}'(c) = \top$. The unsatisfiability of S_0 guarantees the existence of a clause c' in S_0 with $\mathcal{H}'(c') = \bot$. We prove that c' meets the conditions (i) – (iii). First, the clause c' must contain the literal $\sim L$ and not the literal L, since otherwise $\mathcal{H}(c') = \bot$, which contradicts the selection of $\mathcal{H}$, hence (i). Secondly, for any literal L' in c' different from $\sim L$: $\mathcal{H}(L') = \mathcal{H}'(L') = \bot$. As a consequence, L' cannot occur complemented in c, since otherwise $\mathcal{H}(c) = \top$; this proves (ii). Finally, the essentiality of c' in S_0 entails that there exists an interpretation $\mathcal{H}''$ with $\mathcal{H}''(S_0 \setminus \{c'\}) = \top$ and $\mathcal{H}''(c') = \bot$. Since $\sim L$ is in c', $\mathcal{H}''(L) = \top$. Therefore, c' is essential in $S_0 \cup \{L\}$ and also in its unsatisfiable subset $\{L\} \rhd S_0$. From this and the fact that $\{L\} \rhd S_0$ is a subset of $\{L\} \rhd S$ follows that c' is relevant in $\{L\} \rhd S$.

Proposition 5.7 (Completeness of regular strong connection tableaux) For any finite unsatisfiable set S of ground clauses and any clause c which is relevant in S, there exists a closed regular strong connection tableau for S with top clause c.

Proof Let S be a finite unsatisfiable set of ground clauses and c any relevant clause in S. A closed regular strong connection tableau T for S with top clause c can be constructed from the root to its leaves via a sequence of intermediate tableaux, as follows. Start with a tableau consisting simply of c as top clause. Then iterate the following non-deterministic procedure as long as the intermediate tableau has a branch whose leaf node has no complementary ancestor.

> Choose an arbitrary such leaf node N in the current tableau with literal L. Let c be the tableau clause of N and let $P = \{L_1, \ldots, L_m, L\}$, $m \geq 0$, be the set of literals on the path from the root up to the node N. Then, select any clause c' which is relevant in $P \rhd S$, contains $\sim L$, is strongly connected to c, and does not contain literals from the path $\{L_1, \ldots, L_m, L\}$; perform an expansion step with c' at the node N.

First, note that, evidently, the procedure admits solely the construction of regular strong connection tableaux, since in any expansion step the attached clause contains the literal $\sim L$, no literals from the path to its parent node (regularity), nor is a literal different from $\sim L$ in c' contained complemented in c. Due to regularity, there can be only branches of finite length. Consequently, the procedure must terminate, either because every leaf node has a complementary ancestor, or because no clause c' exists for expansion which meets the conditions stated in the procedure. We prove that the second alternative does never occur, since for any *open* leaf node N with literal L there exists such a clause c'. This will be demonstrated by induction on the node depth. The induction base, $n = 1$, is evident, by the Strong Mate Lemma (5.6). For the step from n to $n+1$, with $n \geq 1$, let N be an open leaf node of tableau depth $n+1$ with literal L, tableau clause c, and with a path set $P \cup \{L\}$ such that c is relevant in $P \rhd S$, the induction assumption. Let S_0 be any minimally unsatisfiable subset of $P \rhd S$ containing c, which exists by the induction assumption. Then, by the Strong Mate Lemma, S_0 contains a clause c' which is strongly connected to c and contains $\sim L$. Since no literal in $P' = P \cup \{L\}$ is contained in a non-unit clause of $P' \rhd S$ and because N was assumed to be open, no literal in P' is contained in c' (regularity). Finally, since S_0 is minimally unsatisfiable, c' is essential in S_0; therefore, c' is relevant in $P' \rhd S$. □

The second half of the completeness proof is a standard lifting argument.

Definition 5.8 (Ground (instance) set) Let S be a set of clauses and S' a set of ground clauses. If, for any clause $c' \in S'$, there exists a clause $c \in S$ such that c' is a substitution instance of c, then S' is called a *ground (instance) set* of S.

Lemma 5.9 Let T' be a closed regular strong connection tableau for a ground set S' of a set of clauses S. Then, for any branch selection function ϕ, there exists a closed regular strong connection tableau T for S constructed according to ϕ such that T is more general than T'.

Proof The proof is exactly as the proof of Lemma 2.61 except that here it is much simpler, since no δ-rule applications can occur. Whenever an expansion or extension step is performed in the construction of T' with a clause c', then a clause $c \in S$ is selected with c' being a ground instance of c and a respective expansion or extension step with c is performed in the construction of T. Furthermore, as in the proof of Lemma 2.61, it may be necessary to perform additional closure steps, which obviously are not needed in the ground proof. □

Theorem 5.10 (Completeness of regular connection tableaux) *For any unsatisfiable set of clauses, any clause c that is relevant in S, and any branch selection function ϕ, there exists a closed regular strong connection tableau constructed according to ϕ and with a top clause that is an instance of c.*

Proof Immediate from Proposition 5.7 and Lemma 5.9. □

5.3 Intertableaux Pruning

In this section, we consider the second main paradigm of improving proof search in enumerative tableau procedures. Here the basic idea is to work on the level of the entire search space. As will be demonstrated with a number of examples, it is often possible to identify a certain tableau T as redundant, because there exists another, better tableau T' in the search space. A natural definition of T' being better than T could be that the tableau T can be closed only if the tableau T' can be closed.

5.3.1 Using Matings for Pruning Tableaux

As already mentioned in Section 4.1.2, one can associate a mating, i.e., a set of connections, with any clausal tableau. Interestingly, this mapping is not injective in general. So one and the same mating may be associated with different tableaux. This means that the matings concept provides a more abstract view on the search space and enables us to group tableaux into equivalence classes. Under certain circumstances, it is not necessary to construct *all tableaux* in such a class but only *one representative*. In order to illustrate this, let us consider the set of propositional clauses

$$\{\neg P_1 \lor \neg P_2, \neg P_1 \lor P_2, P_1 \lor \neg P_2, P_1 \lor P_2\}.$$

As shown in Figure 5.2, the set has 4 closed regular connection tableaux with all-negative start clause $\neg P_1 \lor \neg P_2$. If, however, the involved sets of connections are inspected, it turns out that the tableaux all have the same mating consisting of 6 connections. The redundancy contained in the tableau framework is that certain tableaux are *permutations* of each other corresponding to different possible ways of *traversing* a set of connections. Obviously, only one of the tableaux in such an equivalence class has to be considered.

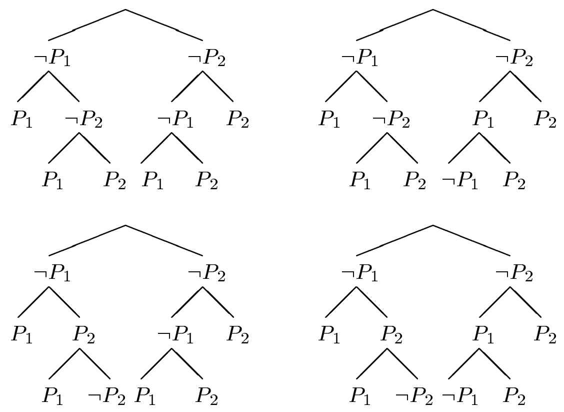


Figure 5.2: Four closed connection tableaux for the same spanning mating.

The question is, how exactly this redundancy can be avoided. A general line of development would be to store all matings that have been considered during the tableau search procedure and to ignore all tableaux which encode a mating which was already generated before. This approach would require an enormous amount of space. Based on preliminary work in [Letz, 1993a], in [Letz, 1998b] a method was developed which can do with very little space and avoid the form of duplication shown in Figure 5.2. To comprehend the method, note that, in the example above, the source of the redundancy is that a certain connection can be used both in an extension step and in a reduction step. This causes the combinatorial explosion. The idea is now to block certain reduction steps by using an ordering $\prec$ on the occurrences of literals in the input set which has to be respected during the tableau construction, as follows. Assume, we want to perform a reduction step from a node N to an ancestor node N'. Let $N_1, \ldots, N_n$ be the node family below N'. The nodes $N_1, \ldots, N_n$ were attached by an extension step "into" a node complementary to N', say N_i. Now we simply do not permit the reduction step from N to N' if $N_i \prec N$ where the ordering $\prec$ is inherited from the literal occurrences in the input set to the tableau nodes. As can easily be verified, for any total ordering, in the example above only one closed tableau can be constructed with this proviso. As shown in detail in [Letz, 1998b], with this method one may achieve a superexponential reduction of the number of closed tableaux in the search space with almost no overhead.

On the other hand, there may be problems when combining this method with other search pruning techniques.

Matings Pruning and Strong Connectedness

For instance, the method is not compatible with the condition of strong connectedness presented in Section 5.1.4. As a counterexample, consider the set of the four clauses given in Example 5.11.

Example 5.11 $\{P \vee Q(a),\, P \vee \neg Q(a),\, \neg P \vee Q(a),\, \neg P \vee \neg Q(x)\}$.

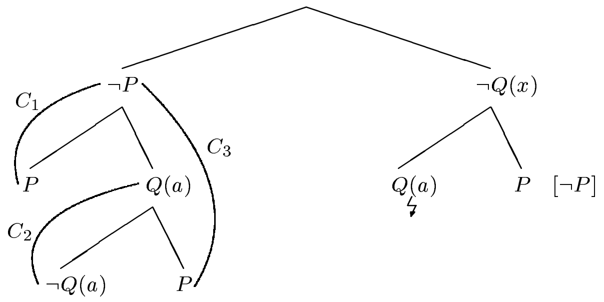


Figure 5.3: Deduction process for Example 5.11.

If we take the fourth clause, which is relevant in the set, as top clause, enter the first clause, then the second one by extension, and finally perform a reduction step, then the closed subtableau on the left-hand side encodes the mating $\{C_1, C_2, C_3\}$. Now, any extension step at the subgoal labelled with $\neg Q(x)$ on the right-hand side immediately violates the strong connection condition. Therefore, backtracking has to occur, up to the state in which only the top clause remains. Afterwards, the second clause must be entered, followed by an extension step into the first one. But now the mating pruning forbids a reduction step at the subgoal labelled with P, since it would produce a closed subtableau encoding the same mating $\{C_3, C_2, C_1\}$ as before. Since extension steps are impossible because of the regularity condition, the deduction process would fail and incorrectly report that there exists no closed tableau with the fourth clause as top clause.

This is but one example of an incompatibility between different pruning methods, here a structural one (strong connectedness) with a global one (avoiding the repetition of matings). And in this case there is no reasonable reconciliation of both pruning methods. It depends on the particular input formula which one of the techniques is more effective for search pruning.

Minimal Matings

A further potential of using matings for pruning tableaux is by exploiting the minimality of matings.

Definition 5.12 (Minimal mating) A mating M is called *minimal for* a set of clauses S if, for each connection C in M, there is a path through S containing C and no other connection from M.

Proposition 5.13 A set of clauses S is unsatisfiable if and only if there is a unifiable minimal spanning mating for a multiplicity of S.

Proof If a set of clauses S is unsatisfiable, then, by Theorem 4.11, there exists a unifiable spanning mating M for a multiplicity S' of S. Assume M be not minimal for S'. Then, some connection C in M can be removed without affecting the spanning property. This way, after finitely many steps, a minimal spanning mating for S' is obtained. □

The general motivation for developing complete refinements of matings is to achieve redundancy elimination in the first-order case. In contrast to the propositional case, where always the full set of connections in a formula can be taken, in the first-order case, with every connection that is added to a mating, an additional unification problem has to be solved. The restriction to minimal matings keeps the simultaneous unification problem as easy as possible.

In [Letz, 1999b], it was proven that, for any minimal mating for a set of clauses S, there exists a closed strict connection tableau for S. Unfortunately, this does no more hold when strictness is replaced by regularity. In fact, the restriction to minimal matings is not compatible with the regularity condition.

Proposition 5.14 There is an unsatisfiable set of clauses for which there exists no closed regular (path) connection tableau with a minimal mating.

Proof Let S be the set consisting of the following 7 propositional clauses

$$(1)\,\neg p \vee \neg s, (2)\, p \vee \neg q, (3)\, p \vee r \vee q, (4)\, \neg r \vee \neg s, (5)\, s \vee \neg r, (6)\, s \vee \neg q, (7)\, \neg p \vee q.$$

We prove that, when taking the first clause $\neg p \vee \neg s$ as start clause, then there is no closed regular connection tableau with a minimal mating. For illustration, consider Figure 5.4. First, it is straightforward to recognize that any regular

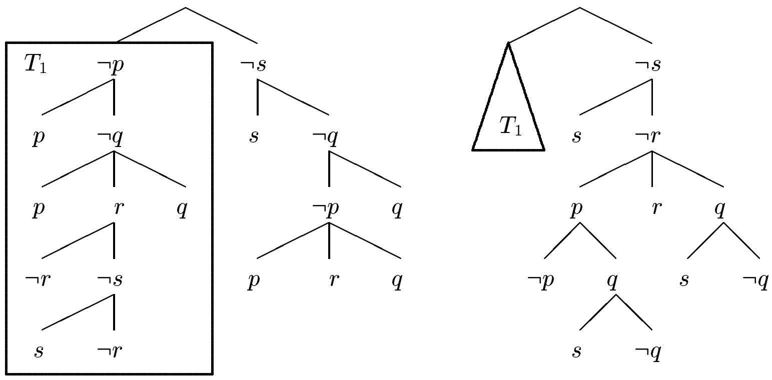


Figure 5.4: The incompatibility of minimal matings and regularity.

solution of the $\neg p$-subgoal in the top clause has the following mating consisting of the six connections (with upper indices giving the clause numbers):

$$\{\neg p^1, p^2\}, \{\neg p^1, p^3\}, \{\neg q^2, q^3\}, \{r^3, \neg r^4\}, \{r^3, \neg r^5\}, \{\neg s^2, q^3\}.$$

Now, the top $\neg s$-subgoal can either be extended with the fifth or the sixth clause. Let us start with the case of the latter clause $s \vee \neg q$, which is shown on the left-hand side of the figure. The remaining $\neg q$-subgoal can only be extended by attaching the clause $\neg p \vee q$ and afterwards the clause $p \vee r \vee q$, or the other way round. When the respective q-subgoal is solved by a reduction step, a subgoal labelled with r remains. At this stage, the tableau has the additional four connections:

$$\{\neg s^1, s^6\}, \{\neg q^6, q^7\}, \{\neg p^7, p^3\}, \{\neg q^6, q^3\}.$$

Although the tableau is still open, the ten connections of the tableau are minimal and spanning for S. In order to extend the r-subgoal in a regular way, only the clause $s \vee \neg r$ can be used. This does not increase the set of connections. But now the remaining s-subgoal must be solved by a reduction step using the top $\neg s$-subgoal. This requires the additional connection $\{\neg s^1, s^5\}$ and renders the mating of the tableau nonminimal. The other main case is elaborated on the right-hand side of the figure. Here the redundant connection $\{\neg s^1, s^5\}$ is used first. It is straightforward to recognize that the other four connections are still essential for closing this tableau in a regular manner, so that the minimality of the mating cannot be achieved. It is easy to verify that the more general case of path connection tableaux is also captured by this example.

In order to have an example where the incompatibility of minimal matings and regularity holds for any start clause, we may use the following *duplication trick*. We duplicate the clause set by using consistently renamed predicate symbols and afterwards replace the top clause $c = \neg p \vee \neg s$ and its renamed version $c' = \neg p' \vee \neg s'$ by the new clause $c \vee c'$. For the resulting clause set, it does not matter with which clause we start. Whenever the clause $c \vee c'$ is entered from a subgoal for the first time on a branch, then the (path) connection condition guarantees that either no literals from the original or no literals from the renamed part appear on the branch up to this extension step. This means that, for the respective new part, say, c', we can proceed as if c' would be a top clause. $\square$

In summary, this means that either we have to give up regularity or we cannot use the minimality restriction on matings. This is a second severe incompatibility between a structural and a global pruning technique, and again there seems to be no reasonable remedy.

5.3.2 Tableau Subsumption

A much more powerful application of the idea of subsumption between input clauses and tableau clauses consists in generalizing subsumption between clauses to subsumption between entire tableaux. For a powerful concept of subsumption between formula trees, the following notion of formula tree *contractions* proves helpful.

Definition 5.15 ((Formula) tree contraction) A (formula) tree T is called a *contraction* of a (formula) tree T' if T' can be obtained from T by attaching n (formula) trees to n non-leaf nodes of T, for some $n \geq 0$.

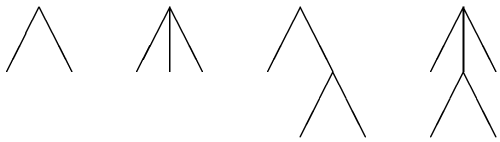


Figure 5.5: Illustration of the notion of tree contractions.

In Figure 5.5, the tree on the left is a contraction of itself, of the second and the fourth tree but not a contraction of the third one. Furthermore, the third tree is a contraction of the fourth one, which exhausts all contraction relations among these four trees. Now subsumption can be defined easily by building on the instance relation between formula trees.

Definition 5.16 (Formula tree subsumption) A formula tree T *subsumes* a formula tree T' if some formula tree contraction of T' is an instance of T.

Since the exploitation of subsumption between entire tableaux has not enough potential for reducing the search space, we favour the following form of subsumption deletion.

Definition 5.17 (Subsumption deletion) For any pair of different nodes $\mathcal{N}$ and $\mathcal{N}'$ in a tableau search tree $\mathcal{T}$, if the goal tree of the tableau at $\mathcal{N}$ subsumes the goal tree of the tableau at $\mathcal{N}'$, then the whole subtree of the search tree with root $\mathcal{N}'$ is deleted from $\mathcal{T}$.

With subsumption deletion, a further form of global redundancy elimination is achieved which is complementary to the tableau structural pruning methods like regularity. Note also that the case of tableau clause subsumption is trivially subsumed by this method. In [Letz et al., 1992] it is shown that, for many formulae, cases of goal tree subsumption inevitably occur during proof search. Since this type of redundancy cannot be identified with tableau structure refinements like connectedness, regularity, or allies, methods for avoiding tableau subsumption seem to be essential for achieving a well-performing tableau search procedure.

Tableau subsumption vs. regularity

Similar to the case of resolution where certain refinements of the *calculus*, i.e., restrictions of the resolution *inference rule*, become incomplete when combined with subsumption deletion, such cases also occur for refinements of tableau calculi. Formally, the compatibility with subsumption deletion can be expressed as follows.

Definition 5.18 (Compatibility with subsumption) A tableau calculus is said to be *compatible with subsumption* if any of its search trees $\mathcal{T}$ has the following property. For arbitrary pairs of nodes $\mathcal{N}$, $\mathcal{N}'$ in $\mathcal{T}$, if the goal tree S of the tableau T at $\mathcal{N}$ subsumes the goal tree S' of the tableau T' at $\mathcal{N}'$ and if $\mathcal{N}'$ dominates a success node, then $\mathcal{N}$ dominates a success node.

The (connection) tableau calculus is compatible with subsumption, but the integration of the regularity condition, for example, poses problems.

Proposition 5.19 The regular connection tableau calculus is incompatible with subsumption.

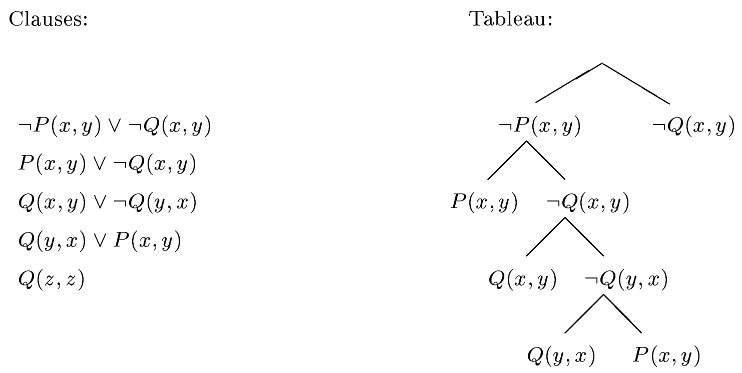


Figure 5.6: The incompatibility of subsumption and regularity.

Proof We use the unsatisfiable set of clauses displayed on the left of Figure 5.6. Taking the first clause as top clause and employing the depth-first left-most selection function, the first subgoal N labelled with $\neg P(x, y)$ can be solved by deducing the tableau T depicted on the right of the figure. Since N has been solved optimally, i.e., without instantiating its variables, the goal tree of T subsumes the goal trees of all other tableaux working on the solution of N. Hence, all tableaux competing with T can be removed by subsumption deletion. But T cannot be extended to a solved tableau, due to the regularity condition, the crucial impediment being that an extension step into $Q(z, z)$ is not permitted, since it would render the already solved subtableau on the left irregular. To obtain a formula in which subsumption is fatal for *any* top clause, one can employ the *duplication trick* used in the proof of Proposition 5.14. □

With the same example, one can also show the incompatibility of tableau subsumption and tautology deletion. Exactly in the situation when the tableau becomes irregular (when x is unified with y), the tableau clause $Q(x, y) \vee \neg Q(y, x)$ becomes tautological. The obvious problem with regularity and tautology deletion is that it considers the entire tableau whereas tableau subsumption considers only the goal trees of the tableau. A straightforward solution therefore is to restrict the structural conditions to goal trees, too. The respective weakening of regularity is called *goal tree regularity*.

5.3.3 Failure Caching

The observation that cases of subsumption inevitably will occur in practice suggests to organize the enumeration of tableaux in such a manner that cases of subsumption can really be detected. This could be achieved with a proof procedure which explicitly constructs competitive tableaux and thus investigates the search tree in a breadth-first manner. However, as already mentioned, the explicit enumeration of tableaux or goal trees is practically impossible. But when performing an implicit enumeration of tableaux by using iterative-deepening search procedures, at each time only one tableau is in memory. This renders it very difficult to implement subsumption techniques in an adequate way. A restricted concept of subsumption deletion, however, can be achieved using so-called "failure caching" methods. The idea underlying this approach is to avoid the repetition of subgoal solutions which apply the same or a more special substitution to the respective branch. There are two approaches, one is using a permanent cache [Astrachan and Loveland, 1991], the other a temporary one [Letz et al., 1992]. We describe the latter method, which might be called "local failure caching" in more detail, because it turned out to be more successful in practice. Subsequently, we assume that only depth-first branch selection functions are used.

Definition 5.20 (Solution -, failure substitution) Given a tableau search tree $\mathcal{T}$ for a tableau calculus and a depth-first branch selection function, let $\mathcal{N}$ be a node in $\mathcal{T}$, T the tableau at $\mathcal{N}$ and N the selected subgoal in T.

1. If $\mathcal{N}'$ with tableau T' is a node in the search tree $\mathcal{T}$ dominated by $\mathcal{N}$ such that all branches through N in T' are closed, let $\sigma' = \sigma_1 \cdots \sigma_n$ be the composition of substitutions applied to the tableau T on the way from $\mathcal{N}$

to $\mathcal{N}'$. Then the substitution $\sigma = \{x/x\sigma' \in \sigma' : x \text{ occurs in } T\}$, i.e., the set of bindings in σ' with domain variables occurring in the tableau T, is called a *solution (substitution) of N at $\mathcal{N}$ via $\mathcal{N}'$.*

2. If $\mathcal{T}'$ is an initial segment of the search tree $\mathcal{T}$ containing no proof at $\mathcal{N}'$ or below it, then the solution σ is named a *failure substitution for N at $\mathcal{N}$ via $\mathcal{N}'$ in $\mathcal{T}'$.*

Briefly, when a solution of a subgoal N with a substitution σ does not permit to solve the rest of the tableau under a given size bound, then this solution substitution is a failure substitution. We describe how failure substitutions can be applied in a search procedure which explores tableau search trees in a depth-first manner employing structure sharing and backtracking.

Definition 5.21 (Generation, application, and deletion of a failure substitution) Let $\mathcal{T}$ be a finite initial segment of a tableau search tree.

1. Whenever a subgoal N selected in a tableau T at a search node $\mathcal{N}$ in $\mathcal{T}$ has been closed via (a sub-refutation to) a node $\mathcal{N}'$ in the search tree, then the computed solution σ is stored at the tableau node N. If the tableau at $\mathcal{N}'$ cannot be completed to a closed tableau in $\mathcal{T}'$ and the proof procedure backtracks over $\mathcal{N}'$, then σ is turned into a failure substitution.

2. In any alternative solution process of the tableau T below the search node $\mathcal{N}$, if a substitution $\tau = \tau_1 \cdots \tau_m$ is computed with one of the failure substitutions stored at N being more general than τ, then the proof procedure immediately backtracks.

3. When the search node $\mathcal{N}$ (at which the tableau node N was selected for solution) is backtracked, then all failure substitutions at N are deleted.

In order to comprehend the mechanism, we show the method at work on an example, Example 5.22. The search process is documented in Figure 5.7. Assume, we start with the first clause in S and explore the corresponding tableau search tree using a depth-first left-to-right branch selection function, just like in Prolog. Accordingly, in inference step 1, the subgoal $\neg P(x)$ is solved using the clause $P(a)$. With this substitution, the remaining subgoals cannot be solved. Therefore, when backtracking step 1, the failure substitution $\{x/a\}$ is stored at the subgoal $\neg P(x)$. The search pruning effect of this failure substitution shows up in inference step 5 when the failure substitution is more general than the computed tableau substitution, i.e., the tableau T_5 is subsumed by the tableau T_1. Without this pruning method, the steps 2 to 4 would have to be repeated.

Note also that one has to be careful to delete failure substitutions under certain circumstances, as expressed in item 3 of the procedure. This provision applies, for example, to the failure substitution $\{y/a\}$ generated at the subgoal $\neg Q(y)$ after the retraction of inference step 2. When the choice point of this subgoal is completely exhausted, then $\{y/a\}$ has to be deleted. Otherwise, it would prevent the solution process of the tableau when, in step 7, this subgoal is again solved using the clause $Q(a)$.

Example 5.22 Let S be the set of the five clauses

$$\neg P(x) \vee \neg Q(y) \vee \neg R(x), \quad P(a), \quad P(z) \vee \neg Q(z), \quad Q(a), \quad Q(b), \quad R(b).$$

	action	subgoals	substitution	fail.subs.
T_0	start step	$\neg P(x), \neg Q(y), \neg R(x)$	$\emptyset$	$\emptyset$
T_1	$P(a)$ entered	$\neg Q(y), \neg R(a)$	$\{x/a\}$	$\emptyset$
T_2	$Q(a)$ entered	$\neg R(a)$	$\{x/a, y/a\}$	$\emptyset$
	unification failure	$\neg R(a)$	$\{x/a, y/a\}$	$\emptyset$
	retract step 2	$\neg Q(y), \neg R(a)$	$\{x/a\}$	$\{y/a\}$
T_3	$Q(b)$ entered	$\neg R(a)$	$\{x/a, y/b\}$	$\{y/a\}$
	unification failure	$\neg R(a)$	$\{x/a, y/a\}$	$\{y/a\}$
	retract step 3	$\neg Q(y), \neg R(a)$	$\{x/a\}$	$\{y/a\}$
	retract step 1	$\neg P(x), \neg Q(y), \neg R(x)$	$\emptyset$	$\{x/a\}$
T_4	$P(x) \vee \neg Q(x)$ entered	$\neg Q(x), \neg Q(y), \neg R(x)$	$\{z/x\}$	$\{x/a\}$
T_5	$Q(a)$ entered	$\neg Q(y), \neg R(a)$	$\{z/a, x/a\}$	$\{x/a\}$
T_5	T_5 subsumed by T_1	$\neg Q(y), \neg R(a)$	$\{z/a, x/a\}$	$\{x/a\}$
	retract step 5	$\neg Q(x), \neg Q(y), \neg R(x)$	$\{z/x\}$	$\{x/a\}$
T_6	$Q(b)$ entered	$\neg Q(y), \neg R(b)$	$\{z/b, x/b\}$	$\{x/a\}$
T_7	$Q(a)$ entered	$\neg R(b)$	$\{z/b, x/b, y/a\}$	$\{x/a\}$
T_8	$R(b)$ entered		$\{z/b, x/b, y/a\}$	$\{x/a\}$

Figure 5.7: Proof search using failure substitutions.

As already noted in the example, when the failure substitution $\{x/a\}$ at $\neg P(x)$ is more general than an alternative solution substitution of the subgoal, then the goal tree of the former tableau subsumes the one of the tableau generated later. The described method preserves completeness, for certain completeness bounds.

Proposition 5.23 Let $\mathcal{T}$ be the initial segment of a (connection) tableau search tree defined by some branch selection function and the depth bound (Section 3.3.1) or some clause-dependent depth bound (Section 3.3.1) with size limitation k. Assume a failure substitution σ has been generated at a node N selected in a (connection) tableau T at a search node $\mathcal{N}$ in $\mathcal{T}$ via a search node $\mathcal{N}'$ according to the procedure in Definition 5.21. If T_c is a closed (connection) tableau in the search tree $\mathcal{T}$ below the search node $\mathcal{N}$ and τ is the composition of substitutions applied when generating T_c from T, then the failure substitution σ is not more general than τ.

Proof Assume indirectly, that σ is more general than τ, i.e., $\tau = \sigma\theta$. Let S_c and S be the subtableaux with root N in T_c respectively in the tableau at $\mathcal{N}'$. Then, replacing S_c in T_c with $S\theta$ results in a closed (connection) tableau T_c'. Furthermore, it is clear that T_c' satisfies size limitation k of the respective completeness bound. Since the (connection) tableau calculus is strongly independent of the selection function, a variant of T_c' must be contained in the tree $\mathcal{T}$ below the search node $\mathcal{N}'$. But this contradicts the assumption of σ being a failure substitution. $\square$

The failure caching method described above has to be adapted when combined with other completeness bounds. While, for the (clause-dependent) depth bounds, exactly the described method can be used, one has to be careful not to lose completeness when using the inference bound. It may happen that a subgoal solution with solution substitution σ exhausts almost all available inferences so that there are not enough left for the remaining subgoals, and there might exist another, smaller solution tree of the subgoal with the same substitution which would permit the solution of the remaining subgoals. Then the failure substitution σ would prevent this. Accordingly, in order to guarantee completeness, the number of inferences needed for a subgoal solution has to be attached to a failure substitution, and only if the solution tree computed later is greater or equal to the one associated with σ, σ may be used for pruning.

When performing iterative-deepening according to the multiplicity bound, however, the situation is more difficult. In order to preserve completeness, not only the solution substitution of a subgoal N (and its branch) has to be considered, but also the substitutions applied to all clauses used in the subrefutation of N. This renders failure caching practically useless for the multiplicity-based calculi.

Furthermore, when using failure caching together with structural pruning methods like regularity, tautology deletion, or tableau clause subsumption, phenomena like the one discussed in Section 5.3.2 (Proposition 5.19) may lead to incompleteness. A remedy is to restrict the structural conditions to the goal tree of the current tableau. But even if this condition is complied with, completeness may be lost, as demonstrated with the following example.

Example 5.24 Let S be the set of the seven clauses

$$\neg P(x,b) \vee \neg Q(x), P(x,b) \vee \neg R(x) \vee \neg P(y,b), P(a,z), P(x,b), R(a), R(x), Q(a).$$

Using the first clause as start clause and performing a Prolog-like search strategy, the clause $P(x,b) \vee \neg R(x) \vee \neg P(y,b)$ is entered from the subgoal $\neg P(x,b)$. Solving the subgoal $\neg R(x)$ with the clause $R(a)$ leads to a tableau structure violation (irregularity or tautology) when $\neg P(y,b)\{x/a\}$ is solved with $P(a,z)$. This triggers the creation of a failure substitution $\{x/a\}$ at the subgoal $\neg R(x)$. The alternative solution of $\neg R(x)$ (with $R(x)$) and of $\neg P(y,b)$ (with $P(a,z)$) succeeds, so that the subgoal $\neg P(x,b)$ in the top clause is solved with the empty substitution $\emptyset$. The last subgoal $\neg Q(x)$ in the top clause, however, cannot be solved using the clause $Q(a)$ due to the failure substitution $\{x/a\}$ at $\neg R(x)$. This initiates backtracking, and the solution substitution $\emptyset$ at the subgoal $\neg P(x,b)$ is turned into a failure substitution. As a consequence, any alternative solution of this subgoal will be pruned, so that the procedure does not find a closed tableau, although the set is unsatisfiable. The problem is that the first encountered tableau structure violation has mutated to a failure substitution $\{x/a\}$. One possible solution is to simply ignore the fatal failure substitution $\{x/a\}$ when the respective node $\neg P(x,b)$ is solved. In general, this suggests the following modification of Definition 5.21.

Definition 5.25 (Failure caching with structural conditions) Items 1 and 3 are as in Definition 5.21, item 2 has to replaced with the following.

2'. In any alternative solution process *of the subgoal* N (instead of the entire tableau T) below the search node $\mathcal{N}$, if a substitution $\tau = \tau_1 \cdots \tau_m$ is computed with one of the failure substitutions stored at N being more general than τ, then the proof procedure immediately backtracks.

In other terms, the failure substitutions at a subgoal have to be deactivated when the subgoal has been solved. This restricted usage of failure substitutions for search pruning preserves completeness. It would be interesting to investigate which weaker restrictions on failure caching and the structural tableau conditions would guarantee completeness. With the failure caching procedure described in Definition 5.25 a significant search pruning effect can be achieved, as confirmed by a wealth of experimental results [Letz et al., 1994, Moser et al., 1997].

Comparison with other methods

The *caching* technique proposed in [Astrachan and Stickel, 1992] stores the solutions of subgoals independently of the path contexts in which the subgoals appear. Then, cached solutions can be used for solving subgoals by *lookup* instead of search. In the special case in which no solutions for a cached subgoal exist, the cache acts in the same manner as the local failure caching mechanism. For propositional Horn sets, this method results in a polynomial decision procedure [Plaisted, 1994, Plaisted and Zhu, 1997]. One difference is that failure substitutions take the path context into account and hence are compatible with goal tree regularity whereas the mentioned caching technique is not. On the other hand, permanently cached subgoals without context have more cases of application than the temporary and context-dependent failure substitutions. The main disadvantage of the context-ignoring caching technique, however, is that its applicability is restricted to the Horn case. Note that the first aspect of the mentioned caching technique, namely, replacing search by lookup, cannot be captured with a temporary mechanism as described above, since lookup is mainly effective for *different* subgoals whereas failure substitutions are merely used on different solutions of *one and the same* subgoal.

In [Loveland, 1978] a different concept of subsumption was suggested for model elimination chains. Roughly speaking, this concept is based on a proof transformation which permits to ignore certain subgoals if the set of literals at the current subgoals is subsumed by an input clause. Such a replacement is possible, for example, if the remaining subgoals can be solved without reduction steps into their predecessors. In terms of tableaux, Loveland's subsumption *reduces* the current goal tree while our approach tries to *prune* it.

Chapter 6

Methods of Shortening Proofs

The analytic tableau approach has proven successful, both proof-theoretically and in the practice of automated deduction. It is well-known, however, since the work of Gentzen [Gentzen, 1935] that the purely analytic paradigm suffers from a fundamental weakness, namely, the poor *deductive power*. That is, for very simple examples, the smallest tableau proof may be extremely large if compared with proofs in other calculi. In this section, we shall review methods which can remedy this weakness and lead to significantly shorter proofs.

The methods we mention are of three completely different types. First, we present mechanisms that amount to adding additional inference rules to tableau systems. The mechanisms are all centered around the (backward) cut rule, which, in its full form, may lead to nonelementarily smaller tableau proofs. Those mechanisms have the widest application, since they already improve the behaviour of tableaux for propositional logic. Second, we consider so-called liberalizations of the δ-rule which may also lead to nonelementarily smaller tableau proofs. Their application, however, is restricted to formulae that are not in Skolem form. Since in automated deduction normally a transformation into Skolem form is performed, the techniques seem mainly interesting as an improvement of this transformation. Finally, we consider in some more detail a line of improvement which is first-order by its very nature, since it can only be effective for free-variable tableaux. It is motivated by the fact that free variables in tableaux need not necessarily be treated as rigid by the closure rule. The generalization of the rule results in a calculus in which the complexity of proofs can be significantly smaller than the Herbrand complexity of the input formula, which normally is a lower bound to the length of any analytic tableau proof.

6.1 Controlled Integration of the Cut Rule

Gentzen's sequent calculus [Gentzen, 1935] contains the *cut rule* which in the tableau format can be formulated as follows.

Definition 6.1 (Tableau) cut rule) The *(tableau) cut rule* is the following tableau expansion rule

(Cut) $\dfrac{}{F \quad \mid \quad \neg F}$ where F is any first-order formula.

The formula F is called the *cut formula* of the cut step. If F is an atomic formula, we speak of an *atomic cut* step.

The cut rule is logically redundant, i.e., whenever there exists a closed tableau with cuts for an input set S, then there exists a closed cut-free tableau for S. Even the following stronger redundancy property holds. For this, note that the effect of the cut rule can be simulated by adding, for every applied cut with cut formula F, the special tautological formula $F \vee \neg F$ to the input set, since then the cuts can be performed by using the β-rule on those tautologies. So in a sense the power of the cut can already be contained in an input set if the right tautologies are contained. That tautologies need not be used as expansion formulae in a tableau is evident from the fact that, for every interpretation and variable assignment, one of the tableau subformulae of a tautology will become true.

Although tautologies and therefore the cut rule are redundant, they can lead to nonelementary reductions of the proof length [Orevkov, 1979, Statman, 1979]. While this qualifies the cut rule as one of the fundamental methods for representing proofs in a condensed format, obviously, the rule has the disadvantage that it violates the tableau subformula property. Consequently, from the perspective of proof *search*, an unrestricted use of the cut rule is highly detrimental, since it blows up the search space.

6.1.1 Factorization and Complement Splitting

The problem therefore is to perform cuts in a controlled manner. A controlled application of the cut rule can be achieved, for instance, by performing a cut in combination with the β-rule only.

Definition 6.2 (β-rule with cut) Whenever a β-step is to be applied, first, perform a cut step with one of the formulae β_1 or β_2, afterwards perform the β-step on the new right branch. The entire operation is displayed in Figure 6.1.

Since one of the new branches is closed, only two open branches have been added, like in the standard β-rule, but one of the branches has one more formula on it which can additionally be used for closure steps. The β-cut rule fulfils the weaker tableau subformula property that any formula in a tableau is either a tableau subformula or the negation of a tableau subformula in the

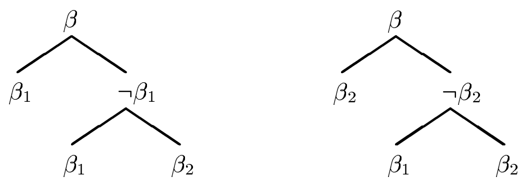


Figure 6.1: β-rule with cut.

input set. This property suffices for guaranteeing that there exist no infinite decomposition sequences. In first-order logic, even a nonelementary proof length reduction can be achieved with this method, as demonstrated in [Egly, 1997]. In the clausal case, which we consider here, this mechanism may lead to a reduction from exponential to linear proof length even in the propositional case. This will be considered in detail in Section 7.2.2. A number of different names have been used for this technique like "(Prawitz) reduction" [Prawitz, 1960], "folding down" [Letz et al., 1994], "lemmas" [d'Agostino, 1999] or "complement splitting" [Bry and Yahya, 1996]. We prefer the folding down format introduced later, since this will permit a closer relation with other techniques.

This method is also closely related with factorization, which we consider next. The *factorization* rule was introduced to the model elimination format in [Kowalski and Kuehner, 1971] (see also [Loveland, 1972]) and used in the connection calculus [Bibel, 1987], Chapter III.6, but, due to format restrictions, for depth-first selection functions only. On the general level of the tableau calculus, which permits arbitrary branch selection functions, the rule can be motivated as follows. Consider a closed tableau containing two nodes N_1 and N_2 labelled with the same literal. Furthermore, suppose that all ancestor nodes of N_2 are also ancestors of N_1. Then, the closed tableau part T below N_2 could have been reused as a solution and attached to N_1, because all expansion and reduction steps performed in T under N_2 are possible in T under N_1, too. This observation leads to the introduction of *factorization* as an additional inference rule. Factorization permits to mark a subgoal N_1 as solved if its literal can be unified with the literal of another node N_2, provided that the set of ancestors of N_2 is a subset of the set of ancestors of N_1; additionally, the respective substitution has to be applied to the tableaux. Reasonable candidates for N_2 are all brothers and sisters of N_1, i.e., all nodes with the same predecessor as N_1, and the brothers and sisters of its ancestors. In Figure 6.2, with an arrow such a factorization step is displayed. Obviously, in order to preserve soundness the rule must be constrained to prohibit solution cycles. Thus, in Figure 6.2 factorization of the subgoal N_4 on the right-hand side with the node N_3 with the same literal on the left-hand side is not permitted after the first factorization (node N_1 with node N_2) has been performed, because this would involve a reciprocal, and hence unsound, employment of one solution within the other. To avoid the cyclic application of factorization, tableaux have to be supplied with an additional factorization dependency relation.

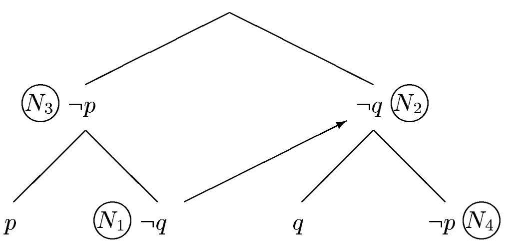


Figure 6.2: Factorization step in a connection tableau.

Definition 6.3 (Factorization dependency relation) A *factorization dependency relation on a tableau* T is a strict partial ordering $\prec$ on the tableau nodes ($N_1 \prec N_2$ means that the solution of N_2 depends on the solution of N_1).

Definition 6.4 (Tableau factorization) Given a tableau T and a factorization dependency relation $\prec$ on its nodes. First, select a subgoal N_1 with literal L and another node N_2 labelled with a literal K such that

1. there is a minimal unifier σ: $L\sigma = K\sigma$,
2. N_1 is dominated by a node N which has the node N_2 among its immediate successors, and
3. $N_3 \not\prec N_2$, where N_3 is the brother node of N_2 on the branch from the root down to and including N_1.[1]

A *factorization* step consists in the following operation. Modify $\prec$ by first adding the pair of nodes $\langle N_2, N_3 \rangle$ and then forming the transitive closure of the relation; then, apply the substitution σ to the tableau; finally, consider the branch with leaf N_1 as closed. We say that the subgoal N_1 has been *factorized with* the node N_2. The tableau construction is started with an empty factorization dependency relation, and all other tableau inference rules leave the factorization dependency relation unchanged.

Applied to the example shown in Figure 6.2, when the subgoal N_1 is factorized with the node N_2, the pair $\langle N_2, N_3 \rangle$ is added to the previously empty relation $\prec$, thus denoting that the solution of the node N_3 depends on the solution of the node N_2. After that, factorization of the subgoal N_4 with the node N_3 is not possible any more.

It is clear that the factorization dependency relation only relates brother nodes, i.e., nodes which have the same immediate predecessor. Furthermore, the applications of factorization at a subgoal N_1 with a node N_2 can be subdivided into two cases. Either, the node N_2 has been solved or one of the branches through N_2 is open, In the second case we shall speak of an *optimistic* application of factorization, since the node N_1 is marked as solved *before* it is known whether a

[1] Note that N_3 may be N_1 itself.

solution exists. Conversely, the first case will be called a *pessimistic* application of factorization. It is obvious that in the pessimistic case no cyclic factorizations may occur, therefore a factorization dependency relation is not needed.

Similar to the case of ordinary (connection) tableaux, if the factorization rule is added, the order in which the tableau rules are applied does not influence the structure of the tableau.

Proposition 6.5 (Strong selection independency of factorization) Any closed (connection) tableau with factorization for a set of clauses constructed with one selection function can be constructed with any other selection function.

Switching from one selection function to another may mean that certain optimistic factorization steps become pessimistic factorization steps and vice versa. If we are working with goal trees, i.e., completely remove solved parts of a tableau, as done in the chain format of model elimination, then for all *depth-first* selection functions solely optimistic applications of factorization can occur. Also, the factorization dependency relation may be safely ignored, because the depth-first procedure and the removal of solved nodes render cyclic factorization attempts impossible. It is for this reason, that the integration approaches of factorization into model elimination or into the connection calculus have not mentioned the need for a factorization dependency relation. Note also that if factorization is integrated into the chain format of model elimination, then the mentioned strong node selection independency does not hold, since pessimistic factorization steps cannot be performed.

The addition of the factorization rule increases the deductive power of (connection) tableaux significantly. In fact, the factorization rule is equivalent to the method of complement splitting, as considered in Section 7.2.2.

6.1.2 The Folding Up Rule

An inference rule which, for connection tableaux, is stronger than factorization concerning deductive power, is the so-called *folding up rule* (in German: "Hochklappen"). Folding up generalizes the *c-reduction* rule introduced to the model elimination format in [Shostak, 1976]. In contrast to factorization, for which pessimistic and optimistic application do not differ concerning deductive power, the shortening of proofs achievable with folding up results from its pessimistic nature. The theoretical basis of the rule is the possibility of extracting *bottom-up lemmata* from solved parts of a tableau, which can be used on other parts of the tableau (as described in [Loveland, 1968] and [Letz et al., 1992], or [Astrachan and Loveland, 1991]). Folding up represents a particularly efficient realization of this idea.

We explain the rule with an example. Given the tableau displayed on the left of Figure 6.3, where the arrow points to the node at which the last inference step (a reduction step with the node 3 levels above) has been performed. With this step we have solved the dominating nodes labelled with the literals r and q. In the solutions of those nodes the predecessor labelled with p has been used

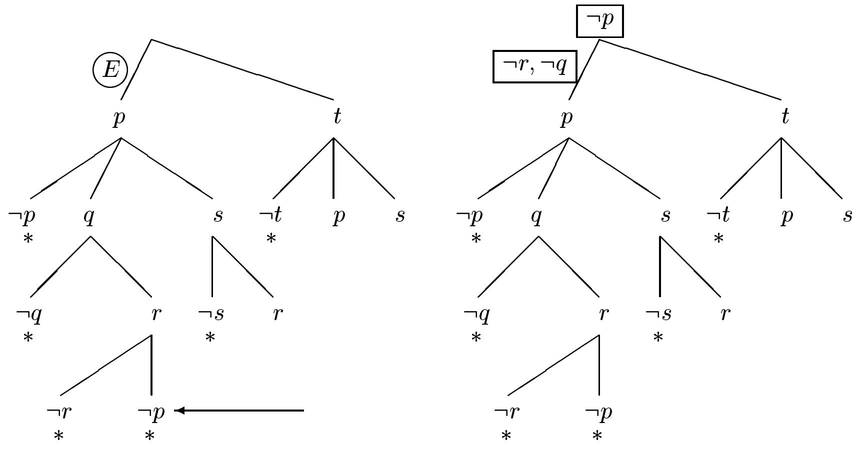


Figure 6.3: Connection tableau before and after three times folding up.

for a reduction step. Obviously, this amounts to the derivation of two lemmata $\neg r \vee \neg p$ and $\neg q \vee \neg p$ from the underlying formula. The new lemma $\neg q \vee \neg p$ could be added to the underlying set and subsequently used for extension steps (this has already been described in [Letz et al., 1992]). The disadvantage of such an approach is that the new lemmata may be *non-unit* clauses, as in the example, so that extension steps into them would produce new subgoals, together with an unknown additional search space. The redundancy brought in this way can hardly be controlled.

With the folding up rule a different approach is pursued. Instead of adding lemmata of arbitrary lengths, so-called *context unit lemmata* are stored. In the discussed example, we may obtain two context unit lemmata:

$\neg r$, valid in the *(path)* context p, and
$\neg q$, valid in the context p.

Also, the memorization of the lemmata is not done by augmenting the input formula but *within* the tableau itself, namely, by "folding up" a solved node to the edge which dominates its solution context. More precisely, the folding up of a solved node N to an edge E means labelling E with the negation of the literal at N. Thus, in the example above the edge E above the p-node on the left-hand side of the tableau is successively labelled with the literals $\neg r$ and $\neg q$, as displayed on the right-hand side of Figure 6.3; lists of context-unit lemmata are depicted as framed boxes. Subsequently, the literals in the boxes at the edges can be used for ordinary reduction steps. So, at the subgoal labelled with r a reduction step can be performed with the edge E, which was not possible before the folding up. After that, the subgoal s could also be folded up to the edge E, which we have not done in the figure, since after solving that subgoal the part below E is completely solved. But now the p-subgoal on the left is solved, and we can fold it up above the root of the tableau; since there is no edge above the root, we simply fold up *into* the root. This folding up step facilitates that the p-subgoal on the right can be solved by a reduction step.

The gist of the folding up rule is that only *unit* lemmata are added, so that the additionally imported indeterminism is not too large. Over and above that, the technique gives rise to a new form of pruning mechanism called *strong regularity*, which is discussed below. Lastly, the folding up operation can be implemented very efficiently, since no renaming of variables is performed, as in a full lemma mechanism.

In order to be able to formally introduce the inference rule, we have to slightly generalize the notion of tableaux.

Definition 6.6 (Edge-labelled tableau, path set) An *edge-labelled tableau (E-tableau)* is just a clausal tableau as introduced in Definitions 3.2 with the only modifications that also the edges and the root node are labelled, namely, with lists of literals. Additionally, in every extension and reduction step, the closed branch is *marked* with the respectively used ancestor literal. The *path set* of a non-root node N in an E-tableau is the union of the sets of literals at the nodes dominating N and in the lists at the root and at the edges dominating the immediate predecessor of N.

Definition 6.7 (E-tableau folding up) Let T be an E-tableau, N a non-leaf node with literal L which dominates a closed subtree. The insertion position of the literal $\sim L$ is computed as follows. From the markings of all leaf nodes dominated by N, select the set M of nodes which dominate N (M contains exactly the predecessor nodes on which the solution of N depends).

> If M is empty or contains the root node only, then add the literal $\sim L$ to the list of literals at the root.
>
> Otherwise, let N' be the deepest path node in M. Add the literal $\sim L$ to the list of literals at the edge immediately above N'.[2]

As an illustration, consider Figure 6.3, and recall the situation when the 'q'-node N on the left has been solved completely. The markings of the branches dominated by N are the 'r'-node below N and the 'p'-node above N. Consequently, $\neg q$ is added to the list at the edge E.

Additionally, the reduction rule has to be extended, as follows.

Definition 6.8 (E-tableau reduction) Given a marked E-tableau T, select a subgoal N with literal L, then

1. either select a dominating node N' with literal K and a minimal unifier σ for L and $\sim K$, and mark the branch with N',

[2] The position of the inserted literal exactly corresponds to the *C-point* in the terminology used in [Shostak, 1976].

2. or select a literal K contained in the list at some dominating edge or at the root with a minimal unifier σ for L and $\sim K$; then mark the branch with the node immediately below the edge or with the root, respectively.

Finally, apply the substitution σ to the literals in the tableau and close the branch.

The *tableau* and the *(path) connection tableau calculus with folding up* result from the ordinary versions by working with edge-labelled tableaux, adding the folding up rule, substituting the old reduction rule by the new one, starting with a root labelled with the empty list, and additionally labelling all newly generated edges with the empty list. Subsequently, we will drop the prefix 'E-' and simply speak of 'tableaux', the context will clear up possibly ambiguities.

The soundness of the folding up operation is expressed in the following proposition.

Proposition 6.9 (Soundness of folding up) Let N be any subgoal with literal L in a tableau T, P the path set of N, and S a set of clauses. Suppose T' is any tableau deduced from T using folding up steps and employing only clauses from S in the intermediate extension steps. Then, for the new path set P' of N in T': $P \cup S$ logically implies P'.

Proof The proof is by induction on the number n of folding up steps between T and T'. The base case for $n = 0$ is trivial, since $P' = P$. For the induction step, let $P' = P_n$ be the path set of N after the n-th folding up step inserting a literal, say L', into the path above N. This step was the consequence of solving a literal $\sim L'$ with clauses from S and path assumptions from P_{n-1}, i.e., the path set of N before the n-th folding up step. This means that $P_{n-1} \cup S \cup \{\sim L'\}$ is unsatisfiable. Now, by the induction assumption, $P \cup S \models P_{n-1}$. Consequently, $P \cup S \models P_{n-1} \cup \{L'\} = P'$. □

In Section 7.2.2, it is proven that, for connection tableaux, the folding up rule is properly stronger concerning deductive power than complement splitting or the factorization rule.

6.1.3 The Folding Down Rule

The simulation of factorization by folding up also shows how a restriction of the folding up rule could be defined which permits an *optimistic* labelling of edges. If a strict linear (dependency) ordering $\prec$ is defined on the successor nodes $N_1, \ldots, N_m$ of any node, then it is permitted to label the edge leading to any node N_i, $1 \leq i \leq m$, with the set of the negations of the literals at all nodes which are smaller than N_i in the ordering. We call this operation the *folding down* rule (in German: "Umklappen"). The folding down operation can also be applied incrementally, as the ordering is completed to a linear one.

It is obvious that folding down is just Prawitz reduction or complement splitting in a slightly different format. The folding down rule can also be viewed as a very simple and efficient way of *implementing* factorization. Over and above that, if also the literals on the edges are considered as path literals in the regularity test, an additional search space reduction called *strong regularity* can be obtained this way, which is difficult to identify in the factorization framework.

6.1.4 Enforced Folding and Strong Regularity

The folding up operation has been introduced as an ordinary inference rule which, according to its indeterministic nature, may be applied or not. Alternatively, we could have defined versions of the (connection) tableau calculi with folding up in which any solved node *must* be folded up immediately after it has been solved. It is clear that whether folding up is performed freely, as an ordinary inference rule, or in an enforced manner, the resulting calculi are not different concerning deductive power, since the folding up operation is a monotonic operation which does *not decrease* the inference possibilities. But the calculi differ with respect to their search spaces, since by treating the folding up rule just as an ordinary inference rule, which may be applied or not, an additional and absolutely useless form of indeterminism is imported. Consequently, the folding up rule should not be introduced as an additional inference rule, but as a tableau operation to be performed immediately after the solution of a subgoal. The resulting calculi will be called the *(connection) tableau calculi with enforced folding up*.

The superiority of the enforced folding up versions over the unforced ones also holds if the regularity restriction is added, according to which no two *nodes* on a branch can have the same literal as label. But the manner in which the folding up and the folding down rules have been introduced raises the question whether the regularity condition might be sharpened and extended to the consideration of the literals in the labels of the edges, too. It is clear that such an extension of regularity does not go together with folding up, since any folding up operation makes the respective closed branch immediately violate the extended regularity condition. A straightforward remedy is to apply the extended condition to the *goal trees* of tableaux only.

Definition 6.10 (Strong regularity) A tableau T is called *strongly regular* if it is regular and no literal at a subgoal N of T is contained in the path set of N.

When the strong regularity condition is imposed on the connection tableau calculus with enforced folding up, then a completely new calculus is generated which is no extension of the regular connection tableau calculus, that is, not every proof in the regular connection tableau calculus can be directly simulated by the new calculus. This is because after the performance of a folding up operation certain inference steps previously possible for other subgoals may become impossible then. A folding up step may even lead to an immediate failure of the extended regularity test, as demonstrated below. Since the new calculus is no extension of the regular connection tableau calculus, we do not even know whether it is complete, since the completeness result for regular connection tableaux cannot be applied. In fact, the new calculus is *not complete* for every selection function.

Proposition 6.11 *There is an unsatisfiable set S of ground clauses and a selection function ϕ such that there is no refutation for S in the strongly regular connection tableau calculus with enforced folding up.*

Example 6.12 The set S consisting of the clauses

$$\begin{array}{llll} \neg p \vee \neg s \vee \neg r, & p \vee s \vee r, & \neg q \vee r, & q \vee \neg r, \\ \neg p \vee t \vee u, & p \vee \neg t \vee \neg u, & \neg q \vee s, & q \vee \neg s, \\ & & \neg q \vee t, & q \vee \neg t, \\ & & \neg q \vee u, & q \vee \neg u. \end{array}$$

Proof Let S be the set of clauses given in Example 6.12, which is minimally unsatisfiable. The non-existence of a refutation with the top clause $p \vee s \vee r$ for a certain unfortunate selection function ϕ is illustrated in Figure 6.4. If ϕ selects the s-node, then two alternatives exist for extension, separated by a $\vee$. For the one on the left-hand side, if ϕ shifts to the p-subgoal above and completely solves it in a depth-first manner, then the enforced folding up of the p-subgoal immediately violates the strong regularity, indicated with a '$\lightning$' below the responsible $\neg p$-subgoal on the left. Therefore, only the second alternative on the right-hand side may lead to a successful refutation. Following the figure, it can easily be verified that for any refutation attempt there is a selection possibility which either leads to extension steps which immediately violate the old regularity condition or produce subgoals labelled with $\neg p$ or $\neg r$. In those cases, the selection function always shifts to the respective p- or r-subgoal in the top clause, solves it completely and folds it up afterwards, this way violating the strong regularity. Consequently, for such a selection function, there is no refutation with the given top clause. The same situation holds for any other top clause selected from the set. This can be verified in a straightforward though tedious manner. Alternatively, in order to shorten the proof, we may use the duplication trick as used in the proof of Proposition 5.13. We duplicate the clause set by using consistently renamed predicate symbols and afterwards replace the top clause c and its renamed version c' by the new clause $c \vee c'$. For the new clause set, the incompleteness result holds for *any* top clause. □

This result demonstrates that there is a trade-off between optimal selection functions and structural restrictions on tableaux. It would be interesting to investigate under which weakenings of the strong regularity the completeness for arbitrary selection functions might be obtained. If we restrict ourselves to depth-first selection functions, however, the calculus is complete, as shown next.

We are now going to present completeness proofs of two calculi, namely, of strongly regular connection tableaux with enforced folding up, for depth-first selection functions, and of strongly regular connection tableaux with enforced folding down, for arbitrary selection functions. The completeness proofs are based on the following non-deterministic procedure for generating connection tableaux which is similar to the one used in the proof of Proposition 5.7 However, in the following procedure as an additional control structure a mapping α is carried

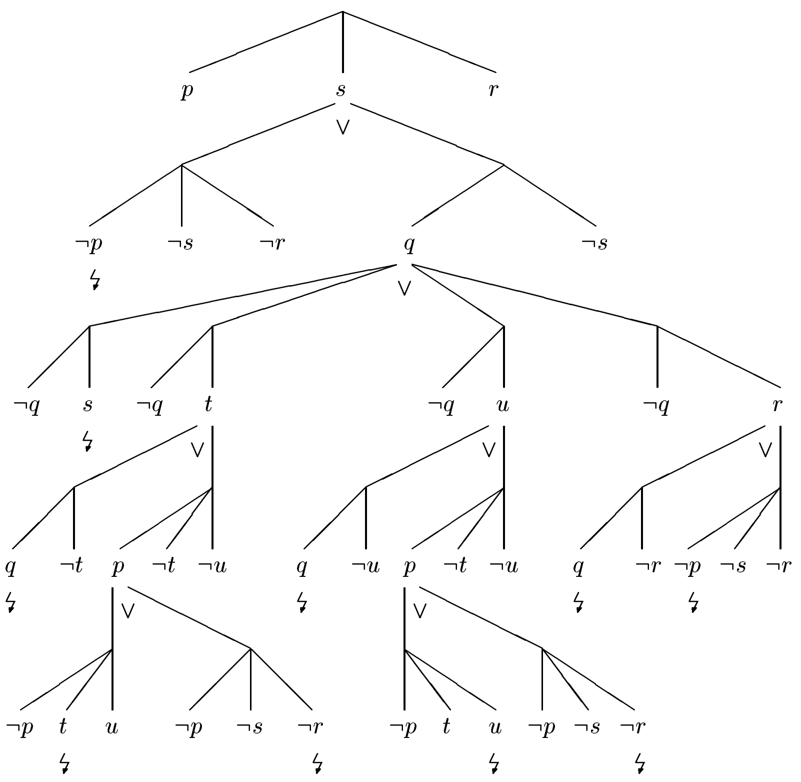


Figure 6.4: Incompleteness for free selection functions of the strongly regular connection tableau calculus with enforced folding up.

along which, upon selection of a subgoal N, associates with N a specific subset $\alpha(N)$ of the input clauses.

Definition 6.13 Let S_0 be a finite unsatisfiable set of ground clauses, c_0 any clause which is relevant in S_0, and ϕ any subgoal selection function. First, perform a start step with the clause c_0 at the root N_0 of a one-node tableau, select a subset S of S_0 with c_0 being essential in S, and set $\alpha(N_0) = S$. Then, as long as applicable, iterate the following procedure.

> Let N be the subgoal selected by ϕ, P the path set of N, L the literal and c the tableau clause at N, and $S = \alpha(N')$ where N' is the immediate predecessor node of N.
>
> – If $\sim L \in P$, perform a reduction step at N.
>
> – Otherwise, perform an extension step at N with a clause c' in S such that c' is relevant in $(P \cup \{L\}) \rhd S$, select a subset S' of S with c' being essential in the set $(P \cup \{L\}) \rhd S'$, and set $\alpha(N) = S'$.

Additionally, depending on the chosen extension of the calculus, enforced folding up or folding down operations need to be applied.

It suffices to perform the completeness proofs for the ground case, since the lifting to the first-order case is straightforward, using the Lifting Lemma (5.9) from Page 93.

Theorem 6.14 (Completeness for enforced folding up) For any finite unsatisfiable set S_0 of ground clauses, any depth-first branch selection function, and any clause c_0 which is relevant in S_0, there exists a refutation of S_0 with top clause c_0 in the strongly regular connection tableau calculus with enforced folding up.

Proof Let S_0 be a finite unsatisfiable set of ground clauses, c_0 any relevant clause in S_0, and ϕ any depth-first branch selection function. We demonstrate that *any* deterministic execution of Procedure 6.13 including enforced folding up operations leads to a refutation in which only strongly regular connection tableaux are constructed. We start with a tableau consisting simply of c_0 as top clause, and let α map the root to any subset S of S_0 in which c_0 is essential. Then we prove by induction on the number of inference steps needed for deriving a tableau that

(i) any generated tableau T is strongly regular, and

(ii) an inference step can be performed at the subgoal $\phi(T)$ according to Procedure 6.13.

The induction base, $n=0$, is evident. For the induction step, let T be a tableau generated with $n > 0$ inference steps, $N = \phi(T)$ with literal L and path set P, c the tableau clause at N, N' the immediate predecessor of N, and $\alpha(N') = S$. Two cases may be distinguished.

Case 1. Either, the last node selected before N was N'. In this case, by the induction assumption and the fact that Procedure 6.13 only permits extension (or start) steps with clauses not containing literals from the path set P, it is guaranteed that T is strongly regular, hence (i). By the induction assumption, c is essential in $P \rhd S$. Consequently, due to the Strong Mate Lemma, at N an inference step according to the procedure can be performed, therefore (ii).

Case 2. Or, the last inference step before the selection of N completely solved a brother node of N. In this case, after having entered the clause c, additional literals may have been inserted by intermediate folding up operations. We show that the resulting tableau is still strongly regular. For this, let N_i be an arbitrary subgoal in T, L_i the literal and c_i the clause at N_i, P_i its (extended) path set in T, and N_i' the immediate predecessor of N_i. With S_i we denote the clause set $\alpha(N_i')$. Let, furthermore, $T^\star$ be the former tableau resulting from the extension step at N' into the clause c, and $P_i^\star$ the path set of N_i in $T^\star$. By the induction assumption, L_i is not contained in $P_i^\star$. According to Procedure 6.13, in the solutions of brother nodes of N_i only clauses from the set $S_i \setminus \{c_i\}$ are permitted for extension steps. Due to the depth-first selection function, the solution process of brother nodes of N is a subprocess of the solution process of brother nodes of N_i. Therefore,

by the soundness of the folding up rule (Proposition 6.9), the set of literals K_i inserted into $P_i^\star$ during the derivation of T from $T^\star$ is logically implied by the satisfiable set $A_i = (P_i^\star \cup S_i) \setminus \{c_i\}$. Since, by the induction assumption, $A_i \cup \{c_i\}$ is unsatisfiable, $A_i \cup \{L_i\}$ is unsatisfiable, too. Consequently, $L_i \notin K_i$, and hence $L_i \notin P_i$. Since this holds for all subgoals of T, T must be strongly regular, which proves (i). Furthermore, all c_i remain essential in the sets $P_i \cup S_i$. Therefore, by the Strong Mate Lemma, at the subgoal N in T an inference step according to Procedure 6.13 can be performed, hence (ii).
Now we have proven that the procedure produces only strongly regular connection tableaux and whenever the procedure terminates, it must terminate with a closed tableau. Finally, the termination of the procedure follows from the fact that, for any finite set of ground clauses, only strongly regular tableaux of finite depth exist. □

Theorem 6.15 (Completeness for enforced folding down) For any finite unsatisfiable set S_0 of ground clauses, any subgoal selection function, and any clause c_0 which is relevant in S_0, there exists a refutation of S_0 with top clause c_0 in the strongly regular connection tableau calculus with enforced folding down.

Proof The structure of the proof is the same as the one for folding up, viz., by induction on the number of inference steps, the two properties given above have to be shown. Therefore, only the induction step is carried out. Suppose a subgoal N is selected with literal L, tableau clause c, path set P, and $\alpha(N') = S$ for the immediate predecessor N' of N. The enforced folding down operation inserts the negations of the literals at the unsolved brother nodes of N into the edge leading to N *before* the subgoal N is solved. First, we prove that such steps always preserve the strong regularity condition. Clearly, folding down operations can only violate this condition for tautological tableau clauses. Since no tautological clause can be relevant in a set and Procedure 6.13 only permits the use of relevant clauses, no tautological clause can occur in a generated tableau. It remains to be shown that any selected subgoal can be extended in concordance with Procedure 6.13. By the induction assumption, c is essential in $P \rhd S$. Hence, there is an interpretation $\mathcal{I}$ with $\mathcal{I}(c) = \bot$ and $\mathcal{I}((P \rhd S) \setminus \{c\}) = \top$. We prove that any folding down operation preserves the essentiality of the clause c. Let $P' = \{\sim K_1, \ldots, \sim K_n\}$ be the set of literals inserted above N in a folding down operation on the literals $K_1, \ldots, K_n$ at the other subgoals in c. Clearly, $\mathcal{I}(\sim K_i) = \top$, for all literals in P'. Therefore, c is essential in $P' \cup (P \rhd S)$ and hence also in its unsatisfiable subset $(P' \cup P) \rhd S$. □

6.1.5 The Benefit of Controlled Cuts in Proof Search

Except for the case of strong regularity, all tableau calculi with controlled variants of the cut rule increase the inferential possibilities. This means that also the search space *increases* in general. So how can one profit from these methods in proof search? The crucial point is that the search for closed connection tableaux is performed by employing iterative-deepening procedures. Since the size of the search space is normally exponential or even doubly exponential wrt. the applied tableau completeness bound (e.g., the inference or the depth bound), the real benefit of using variants of the cut rule is when this way the first proof can be found on an earlier level of the iterative-deepening search. This is possible and frequently happens in practice for tableau size bounds like the inference or the depth bound. Interestingly, such a gain is not possible for the multiplicity-based bounds. This is one explanation for the observation that multiplicity-based iterative-deepening procedures are relatively unsuccessful in practice.

6.2 Liberalizations of the δ-Rule

Our completeness proof of free-variable tableaux has revealed that, for any atomic sentence tableau proof, there is a free-variable tableau proof of the same tree size. Interestingly, the converse, does not hold. The reason lies in the use of the δ^+-rule in free-variable tableaux taken from [Hähnle and Schmitt, 1994, Fitting, 1996], which can lead to significantly shorter tableau proofs [Baaz and Fermüller, 1995].

Example 6.16 Any closed sentence tableau for the formula $\forall x(P(x) \wedge \exists y \neg P(y))$ requires two applications of the γ-rule whereas there is a closed free-variable tableau with only one application of the γ'-rule.[3]

Interestingly, in the first edition of Fitting's book [Fitting, 1990], a more restrictive variant of the δ^+-rule was given, in which the new Skolem term had to contain all variables on the branch and not only the ones contained in the respective δ-formula. Liberalization of the δ-rule means mainly reducing the number of variables to be considered in the respective Skolem term. So, the δ^+-rule introduced in Section 2.2 is already a liberalization of the original δ-rule in free-variable tableaux. In a sense, however, this older version of free-variable tableaux is conceptually cleaner with respect to the "rigid" treatment of free-variables. The original idea of a rigid interpretation of the free variables in a tableau is that they may stand for arbitrary ground terms. Accordingly, the notion of ground satisfiability was introduced [Fitting, 1990].

Definition 6.17 (Ground satisfiability) A collection $\mathcal{C}$ of sets of formulae is *ground satisfiable* if every ground instance of $\mathcal{C}$ has a satisfiable element.

Evidently, $\forall$-satisfiability (as defined on Page 51) entails ground satisfiability, but the converse does not hold. As an example consider the ground satisfiable collection $\{\{P(x), \exists y \neg P(y)\}\}$ which is not $\forall$-satisfiable. The difference between the old and the new δ^+-rule is that the old one preserves ground satisfiability, but the new one does not. The closure rule (C), however, preserves ground satisfiability and hence subscribes to a rigid interpretation of the free variables. So,

[3] On should mention, that this weakness has already been recognized in [Smullyan, 1968], who identified a condition under which a Skolem term in a δ-rule application need not be new. With this liberalization, shorter sentence tableau proofs can be constructed.

the system of free-variable tableaux (Definition 2.57) introduced in Section 2.2 is somewhat undecided in its treatment of free variables. One may argue, however, that with the new system, smaller tableau proofs can be formulated, and this is what counts. In the next subsection, we will therefore draw the consequence and also liberalize the closure rule in such a manner that ground satisfiability is no more preserved. The gain is that with the new rule an additional size reduction of tableau proofs may be achieved.

The δ^+-rule is by far not the "best" Skolemization rule, in the sense that it includes a minimal number of variables in the Skolem term. Consider, for example, the δ-formula $\exists x(P(y) \wedge P(x))$. With the δ^+-rule a unary Skolem term $f(y)$ has to be used. But it is evident that y is irrelevant and a Skolem constant a could be used instead. There are a number of improvements of the δ^+-rule which shall briefly be mentioned here. In [Beckert et al., 1993], it is shown that for each δ-formula stemming from the same formula occurrence in the input set and each number n of free variables in δ, always the same Skolem function symbol f_δ^n may be used without affecting the soundness of the rule. The thus improved δ-rule is named δ^{+^+}. In [Baaz and Fermüller, 1995], this method is further liberalized by identifying a *relevant* subset of the free variables in δ and excluding the other variables from the Skolem term. This method can identify the irrelevancy of the variable y in the aforementioned example. Furthermore, the notion of relevancy defined there can be decided in linear time. The corresponding rule is called δ^*. In the paper, a pairwise comparison between the four mentioned Skolemization methods (the one in [Fitting, 1990], δ^+, δ^{+^+}, and δ^*) is made wrt. the proof shortening effect that may be obtained. Interestingly, for each of the improvements, there are examples for which a nonelementary proof length reduction can be achieved wrt. the previous rule in the sequence.

Note that, in general, there is no notion of relevant free variables in a δ-formula which is both *minimal* and can be computed efficiently. This can be illustrated with the following simple consideration. Consider a δ-formula of the form $\exists y(F \wedge G)$ containing a free variable x in G but not in F. Suppose further that $\exists y(F \wedge G)$ be strongly equivalent to $F \wedge \exists y G$. Then, obviously, the variable x is not relevant, but this might only be identifiable by proving the strong equivalence of two formulae, which is undecidable in general. So the constraint for any notion of relevant variables is that it be efficiently computable.

6.3 Liberalization of the Closure Rule

The final improvement of the tableau rules that we investigate is again of a quantificational nature. It deals with the problem that the rigid interpretation of free variables often leads to an unnecessary lengthening of tableau proofs.

Definition 6.18 (Local variable) A variable x occurring free on an open tableau branch is called *local (to the branch)* if x does not occur free on other open branches of the tableau.

If a variable is local to a branch, then any formula containing x can be treated as universally quantified[4] in x, i.e., the universal closure of the formula wrt. the variable could be added to the branch. Let us formulate this as a tableau rule.

Definition 6.19 (Generalization rule) The *generalization rule* is the following expansion rule which can be applied to any open branch of a tableau

(G) $\quad \dfrac{F}{\forall x F}$ where x is a local variable.

Proposition 6.20 The generalization rule preserves $\forall$-satisfiability.

Proof Given a tableau T with a local variable x, assume F is any formula on an open branch B of T and T' is the tableau obtained by adding the formula $\forall xF$ to B. We work with the coincidence between $\forall$-satisfiability and the satisfiability of the open branch formula of a tableau. Let $\mathcal{B} = B_1 \vee \cdots \vee B \vee \cdots \vee B_n$ be the open branch formula of T with $B = F_1 \wedge \cdots \wedge F \wedge \cdots \wedge F_m$. Then the formula $B_1 \vee \cdots \vee (B \wedge \forall xF) \vee \cdots \vee B_n$ is the open branch formula $\mathcal{B}'$ of T'. Now $\mathcal{B}$ is equivalent to $\forall x\mathcal{B}$. Since x does occur free in B only, $\forall x\mathcal{B}$ is strongly equivalent to $\mathcal{B}'$. Consequently, the satisfiability of $\mathcal{B}$ entails the satisfiability of $\mathcal{B}'$. □

It is apparent that the generalization rule does not preserve ground satisfiability. As a matter of fact, the generalization rule is just of a theoretical interest, since it violates the tableau subformula property. Since we are mainly interested in calculi performing atomic branch closure, it is clear that the new universal formula will be decomposed by the γ'-rule, thus producing a renaming of x in F. And, as instantiations are only performed in closure steps, we would perform the generalization implicitly, exactly at that moment. This naturally leads to a local version of the closure rule.

Definition 6.21 (Local closure rule) Let T be a tableau and S the set of formulae in T. Suppose K and L are two literals in the path set of a branch of T. Let $K\tau$ be a renaming of all local variables in K wrt. S, and $L\theta$ a renaming of all local variables in L wrt. $S \cup \{K\tau\}$. Then, the *local closure rule* is the following rule.

(C_L) Modify T to $T\sigma$ if σ is a minimal unifier for $\{K\tau, {\sim}L\theta\}$ and consider the branch as closed.

The soundness of the local closure rule follows from the fact that its effect can be simulated by a number of applications of the generalization rule, the γ'-rule, and the ordinary closure rule.

Using the local closure rule instead of the standard closure rule, one can achieve a significant shortening of proofs, as illustrated with the following tableau which is smaller than the one given in Figure 3.1. Assume the tableau construction is performed using a right-most branch selection function. The crucial difference then occurs when the right part of the tableau is closed and a tableau clause of

[4] In [Beckert and Hähnle, 1998], the term *universal* variable was used for a similar notion.

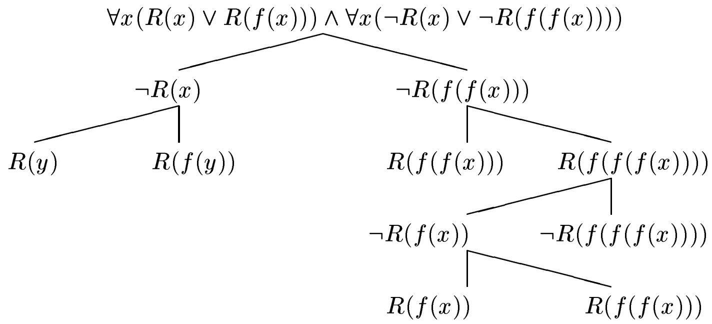


Figure 6.5: Closed clausal tableau with local reduction rule.

the form $R(y) \lor R(f(y))$ is attached on the left. Since the variable x is now local, it can be renamed and the two remanining branches can be closed using the local closure rule.

Although the displayed tableau has no unsatisfiable ground instance, the soundness of the local reduction rule assures that we have indeed refuted the input set. Note also that tableau calculi containing the generalization rule or the local reduction rule are not independent of the branch selection function. As long as the right part of the tableau is not closed, the variable x is not local on the left branch and a renaming of x is not permitted. Consequently, the order in which branches are selected can strongly influence the size of the final tableau. The gain, however, is that the local reduction rule permits to build refutations that are significantly smaller than the Herbrand complexity of the input, as shown in Section 7.4.

Another interesting side-effect of having local variables in a tableau is that the regularity condition can be sharpened.

Definition 6.22 (Regularity wrt. local variables) A tableau is called *regular wrt. local variables* if, for no two formulae Φ and Ψ on a branch of the tableau, there is a substitution σ on the local variables of Φ such that $\Phi\sigma = \Psi$.

So with the use of local variables, which permit more inferential possibilities and hence broaden the search space at first sight, one also obtains a significantly more powerful notion of regularity. In total, this may even lead to a reduction of the size of the search space.

6.3.1 Hyper Tableaux with Local Variables

The use of local variables is particularly beneficial on hyper tableaux. Recall that, in hyper tableaux, every newly attached clause has to be instantiated to a ground instance. It is straightforward to see that local variables need not be instantiated.

Definition 6.23 (Hyper tableaux with local variables) The calculus is the same as the hyper tableau calculus except that the local variables are not instantiated after a hyper extension step.

The hyper tableau calculus with local variables is complete and compatible with the structural condition of regularity wrt. local variables [Baumgartner, 1998]. A particular interesting example is the reflexivity clause $P(x, x)$ contained, for example, in the axioms of equality. The reflexivity clause is fatal for the standard hyper tableau calculus, since all Herbrand instances of the clause have to be used. In the local variant and in the presence of the sharpened regularity condition, no other instance of $P(x, x)$ need to be considered in the entire tableau. When using local variables, it is obvious that Horn clauses never have to be instantiated after a Hyper extension step, since all remaining variables occur in the single positive literal and hence are automatically local. As a result, Hyper tableaux with local variables performs on Horn clauses just as positive hyper resolution [Chang and Lee., 1973], which is known as a particular successful strategy in Horn clause logic.

Chapter 7

Complexities of Minimal Proofs

When assessing the general usefulness of a proof procedure for automated deduction, the main criterion is the ability to find proofs for as many problems as possible in some given time. A proof procedure consists of a proof system, i.e., a calculus with certain refinements, and a search strategy that controls in which order the inferences have to be applied. Because of the complex structure of proof procedures, it is very difficult to compare such procedures mathematically.

A wealth of analytical results, however, exist concerning the relative complexities of minimal proofs in different proof systems. Since those results completely ignore issues of *proof search*, they must not be taken as the only quality criterion of proof systems. However, if, for some interesting class of problems, one proof system permits only very long proofs whereas minimal proofs in another proof system are short, then this is useful information, since the complexities of minimal proofs are lower bounds to the complexities of the respective search spaces.

7.1 Proof Complexity Measures

In this chapter we will give a comprehensive list of relative complexity results of this type for refinements of tableau calculi and some other important proof systems for clause logic, namely, truth tables, semantic trees, and some refinements of resolution. Such results have a long tradition in logic beginning with the results of Gentzen [Gentzen, 1935] on the effect of cut elimination. The first comprehensive systematic such analysis of many different propositional calculi was by Cook and Reckhow [Cook, 1971, Cook and Reckhow, 1974]. Cook introduced the highly influential notion of *polynomial simulation* which compares calculi by abstracting from polynomial differences.

Definition 7.1 (Polynomial simulation) A calculus C_1 *polynomially simulates* a calculus C_2 if there is a polynomial p and, for every formula F and every proof

P_2 of (the inconsistency of) F in C_2 with complexity k, there exists a proof P_1 of (the inconsistency of) F in C_1 with a complexity smaller than $p(k)$.

There are a number of possibilities of measuring the complexities of proofs. In [Letz, 1993a, Letz, 1993b], three natural paradigms of decreasing precision are introduced, which we will shortly review. The finest and therefore most reliable measure charges the *computation cost* needed in a basic machine model to carry out a proof. This could be, for example, the number of configurations in a non-deterministic Turing machine or some other realistic machine model. Since such a measure is normally too detailed to be useful on a more abstract level, two other coarser measures were developed. The second measure abstracts from the actual cost of constructing the proof and simply considers the *size* of the proof object. In order for such a measure to be reliable, it is necessary that the chosen size measure is polynomially related to the actual computation cost. This was called *polynomial size-transparency* in [Letz, 1993a, Letz, 1993b]). On the highest abstraction level, one even disregards the sizes of proofs and only considers the number of inference steps needed to carry out a proof. Again, in order to be useful, this measure must be polynomially related to the actual computation cost. This property was called *polynomial transparency* in [Letz, 1993a, Letz, 1993b]). As shown there, some of the most important proof systems like resolution lack this property, even if only the minimal proofs of a formula are considered. Below we will also sketch the inadequacy of this measure for tableau systems with local unification.

As a consequence of these results, we will *not* use the number of inferences as a proof complexity measure, but the proof size. A naïve size measure of a tableau would be the sum of the symbol sizes of all occurrences of formulae in the tableau. However, the investigations of the complexity of unification have shown that even in one unification step the symbol size may increase exponentially. So the symbol size is not interesting, since one can do much better by using *dags* (directed acyclic graphs) for representing terms. In Figure 7.1, the symbol tree and the minimal symbol dag of a term are shown. We also permit that a set of formulae can be represented by a single symbol dag, so that, for example, the size of a clause set is properly defined.

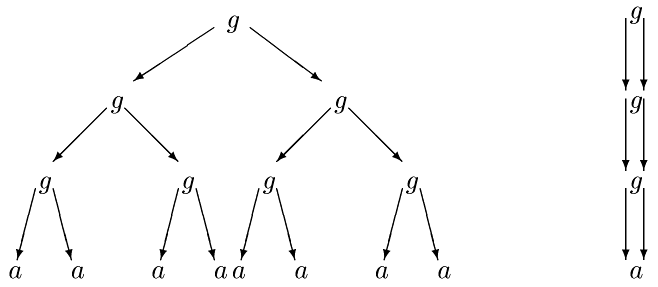


Figure 7.1: Symbol dags of a term $g(g(g(a,a),g(a,a)),g(g(a,a),g(a,a)))$.

Definition 7.2 The *size of a symbol dag* is the number of its edges plus the number of its leaf nodes.

It is clear that this size measure is *realistic*, i.e., polynomially related with the length of an appropriate string representation of the dag.

Definition 7.3 (Tableau size) The *size of a tableau or a general graph labelled with formulae* T is the number of edges and leaf nodes of T plus the size of the minimal symbol dag for the set of formulae appearing in T.

It is straightforward to recognize that this complexity measure is polynomially size-transparent for the considered tableau calculi provided techniques for polynomial unification are used. Furthermore, this measure easily applies to tableaux with folding up and folding down where the edges are labelled with lists of formulae.

7.2 Minimal Proof Lengths in Propositional Logic

7.2.1 Results for Cut-free Clausal Tableaux

We start with results for propositional clause logic. The most primitive approach to determining the satisfiability status of a propositional formula is the well-known *truth table* method. If n is the number of atoms occurring in a formula, an evaluated truth table always contains $n+1$ columns and 2^n lines. The first n columns in each line encode the truth assignments for the atoms in the formula, and the last column contains the truth value of the formula under this assignment.

Proposition 7.4 (Tableaux vs. truth tables) The method of truth tables cannot polynomially simulate tableaux and vice versa.

Proof For the nonsimulation of tableaux by truth tables, consider the class of clause sets

$$\{p_0, \neg p_0 \vee p_1, \ldots, \neg p_{n-1} \vee p_n, \neg p_n\}$$

which have exponential truth tables but obviously closed tableaux of linear size. For the other direction, we may use the class of clause sets given in Example 7.5. A truth table for an S_n has 2^n lines of length $n+1$ and hence a refutation of linear size. In the tableau calculus, however, in any depth k at most $k-1$ branches of a tableau clause can be closed. Therefore, any minimal closed tableau has the tree structure as shown in Figure 7.2, for $n = 3$. Consequently, taking the number of nodes with maximal depth in such a tableau T_n as a lower bound of its size, we obtain that the number of nodes of T_n is greater than $n \times n!$ while the size of S_n is of the order $\mathrm{O}(n \times 2^n)$. So the complexity of T_n is exponential in the complexity of S_n. □

Example 7.5 For any set $\{p_1, \ldots, p_n\}$ of distinct propositional atoms, let S_n denote the set of all 2^n clauses of the shape $L_1 \vee \ldots \vee L_n$ where $L_i = p_i$ or $L_i = \neg p_i$, $1 \leq i \leq n$.

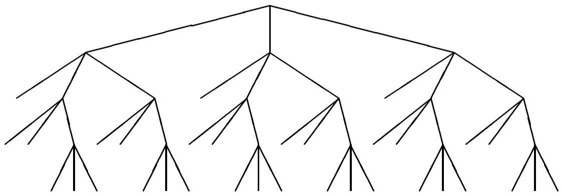


Figure 7.2: Tree structure of a minimal closed tableau for Example 7.5, $n = 3$.

The reason why tableaux can be inferior to truth tables for certain formulae is that the branching in the standard tableau systems does not partition the set of interpretations, i.e., the same interpretation may appear on different branches. When using forms of the cut like factorization or folding up, such a partitioning is guaranteed and hence these formulae become harmless for tableaux.

Next we consider refinements of tableaux like connectedness, regularity, and their combination. Nonsimulation results when adding such refinements may be obtained by using an obvious property of the cut rule. The cut rule may be termed a *data-oriented* inference rules in the sense that it can be simulated by adding tautologies to the input formulae. This is heavily exploited in the subsequent proofs. First, we consider the weak connection condition of path connectedness, which requires that every tableau clause below the top clause be connected to some ancestor literal.

Proposition 7.6 Path connection tableaux cannot polynomially simulate clausal tableaux.

Proof The following simple modification of Example 7.5 will do, namely, the class presented in Example 7.7. The additional tautologies[1] can be used to polynomially simulate the atomic cut rule in the clausal tableau calculus, hence permitting short proofs for this example. But in path connection tableaux, except for the start step, the tautologies do not help, since any path extension step at a node N with literal L using the tautology $L \vee \sim L$ just lengthens the respective path by a node labelled with the same literal L. Therefore, the size of any closed path connection tableau for an input set S_n is greater than $2n \times (n-1)!$ while the size of S_n is of the order $\mathrm{O}(n \times 2^n)$. □

Example 7.7 For any set $\{p_1, \ldots, p_n\}$ of distinct propositional atoms, let S_n denote the set of clauses given in Example 7.5, augmented with n tautologies of the shape $p_i \vee \neg p_i$, $1 \leq i \leq n$.

Next, we consider the relation of the three connection conditions. In contrast to path connectedness, the tight connectedness requires that every tableau clause below the top clause be connected to its *predecessor* node.

[1] That these formula are actually tautologies is not essential for the argument. We could equally well replace every tautology $p_i \vee \neg p_i$ with two clauses $p_i \vee \neg q_i$ and $q_i \vee \neg p_i$ with the q_i being n new distinct propositional atoms.

Proposition 7.8 Connection tableaux cannot polynomially simulate path connection tableaux.

Proof For this result we use another modification of Example 7.5, which is given in Example 7.9. The elements of this class have linear closed path connection tableaux, since the additional clauses permit the linear simulation of the atomic cut rule by starting with p_0 as top clause and successively attaching the clauses $p_i \vee \neg p_i \vee \neg p_0$, before nontautological clauses are used. It is clear that the tautologies do help only if entered at the literal $\sim p_0$. In a connection tableau, however, except for the start step (or for the second inference if we start with the clause p_0) this is not possible. Therefore, the size of any closed connection tableau for an input set S_n is greater than $2n \times (n-1)!$ while the size of S_n is of the order $\mathrm{O}(n \times 2^n)$. □

Example 7.9 For any set $\{p_1, \ldots, p_n\}$ of distinct propositional atoms, let S_n denote the set of clauses given in Example 7.5, augmented with the new atom p_0 and n tautologies of the shape $p_i \vee \neg p_i \vee \neg p_0$, $1 \leq i \leq n$.

In order for the strong connection condition to be satisfied, recall that it is necessary that adjacent tableau clauses have a non-tautological resolvent.

Proposition 7.10 Strong connection tableaux cannot polynomially simulate connection tableaux.

Proof Again a modification of Example 7.5 will do, which is given in Example 7.11. The new tautologies can be used successively in the connection tableau calculus, and hence permit a linear closed connection tableaux. The strong connection condition, however, completely excludes the use of tautological clauses, since any resolvent using a tautological clause will be tautological, too. □

Example 7.11 For any set $\{p_1, \ldots, p_n\}$ of distinct propositional atoms, let S_n denote the set of clauses given in Example 7.5, augmented with $2n-1$ tautologies, viz., the clause $p_1 \vee \neg p_1$ and, for $1 \leq i \leq n-1$, the two clauses $\neg p_i \vee p_{i+1} \vee \neg p_{i+1}$ and $p_i \vee p_{i+1} \vee \neg p_{i+1}$.

The next refinement to be considered is regularity, which requires that no literal occurs more than once on a branch. First, we know already from Proposition 2.28 that regularity is a must for general clausal tableaux, since minimal proofs are always regular. This also holds for the path connectedness condition.

Proposition 7.12 Every minimal closed path connection tableau is regular.

Proof We show the contraposition, i.e., that every closed irregular path connection tableau T is not minimal in size. Let T be such a tableau for a set S. Obtain a formula tree T' by performing Procedure 2.27 (Page40) on T. Clearly, T' is smaller than T and it is closed. Finally, the preservation of path connectedness can be realized by a simple induction on the number of iterations performed in

Procedure 2.27. For the induction step, consider the situation before the n-th iteration. Let N be the respective node with an ancestor N' both labelled with the same literal. Since, by the induction assumption, the current tableau is path connected, every tableau clause in the subtableau below N must be connected to some ancestor node. The critical case is the one in which the respective ancestor is N, which is missing after the n-th step. But since N' is labelled with the same literal, the respective tableau clauses are connected to N', too, which guarantees that the property of path connectedness is preserved. □

Corollary 7.13 Connection tableaux cannot polynomially simulate regular path connection tableaux.

Proof Immediate from the Propositions 7.8 and 7.12. □

Now we come to the interesting result that the regularity condition may be harmful for the deductive power of tableaux when using the tight connection condition. The problem is that the removal of irregularities is not guaranteed to preserve the connectedness. In order to restore connectedness, a global reorganization of the tableau may be necessary which can lead to a significant size increase of the tableau.

Proposition 7.14 Regular connection tableaux cannot polynomially simulate connection tableaux.

Proof For this result we use another modification of Example 7.5, which is given in Example 7.15. The elements of this class have linear closed connection tableaux, since the additional clauses permit the linear simulation of the atomic cut rule, as illustrated in Figure 7.3, for the case of $n = 3$; to gain readability, p_i is abbreviated with i and $\neg p_i$ with $\bar{i}$ in the figure. These connection proofs are highly irregular. In order to obtain short proofs, it is necessary to attach the *mediating* two-literal clauses of the shape $p_i \vee p_0$ and $\neg p_i \vee p_0$ again and again. In regular proofs, however, on each branch such mediating clauses can be used at most once. Consequently, on each branch tautological clauses can be attached at most twice. Therefore, the size of any closed regular connection tableau for an S_n is greater than $4n \times (n-2)!$ while the size of S_n is of the order $\mathrm{O}(n \times 2^n)$. □

Example 7.15 For any set $\{p_1, \ldots, p_n\}$ of distinct propositional atoms, let S_n denote the set of clauses given in Example 7.5, augmented with the new atom p_0 and

1. n tautologies of the shape $p_i \vee \neg p_i \vee \neg p_0$, $1 \leq i \leq n$,
2. n clauses of the structure $p_i \vee p_0$, $1 \leq i \leq n$, and
3. n clauses of the structure $\neg p_i \vee p_0$, $1 \leq i \leq n$.

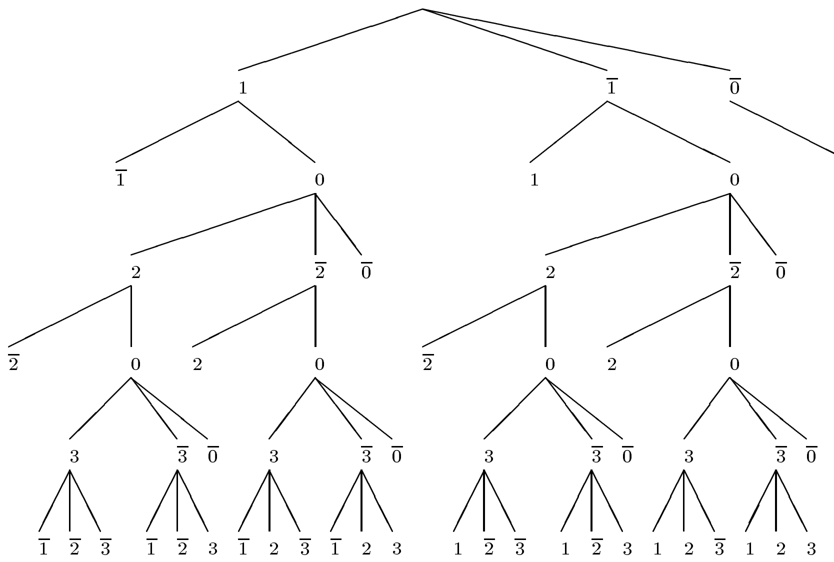


Figure 7.3: Polynomial closed connection tableau for Example 7.15, $n = 3$.

Furthermore, it is evident from all these results, that the various variants of cut-free clausal tableau calculi cannot be simulated by the respective calculi with tautology elimination. Additionally, for connection tableaux, even the deletion of subsumed tableau clauses may be fatal, as can be proven using the same Example 7.15. Note that all mediating clauses of the form $L \vee p_0$ are subsumed by the unit clause p_0 and must be deleted. This blocks the construction of a short proof.

In summary, the presented results illustrate the complementarity of improving *deductive* and *reductive* power for cut-free tableau calculi.

7.2.2 Results for Clausal Tableaux with Controlled Cuts

Next, we will come to the classification of clausal tableau systems which incorporate the atomic cut rule in some form. As a matter of fact, the more powerful full cut rule cannot be used, since it contradicts the clausal format. We will see that some of those calculi are much more robust than the cut-free ones concerning the addition of structural restrictions. But first, we will consider the unrestricted atomic cut rule and show that, with respect to minimal proof lengths, the atomic cut rule has an egalitarian effect on the clausal tableau calculi. For this purpose, we will introduce a certain pathological form of tableaux with cut.

Definition 7.16 (Cut normal form) A clausal tableau with cut is in *cut normal form* if on each branch only the last tableau clause is not attached by a cut step.

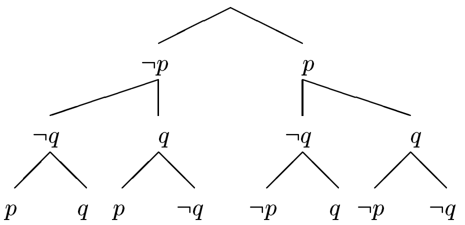


Figure 7.4: Tableau in cut normal form for $\{p \vee q, p \vee \neg q, \neg p \vee q, \neg p \vee \neg q\}$.

Proposition 7.17 For any closed clausal tableau with cut for a set S there is a closed regular connection tableau with cut for S in cut normal form such that the sizes of the tableaux are linearly related.

Proof We show how any tableau inference step can be cast into cut normal form. Cut and reduction steps are trivial. Any tableau expansion step using a clause of length n can be linearly simulated by n atomic cuts, an extension step, and n reduction steps, as shown in Figure 7.5. Finally, regularity may be achieved by pruning the resulting tree using Procedure 2.27 from Page 40. □

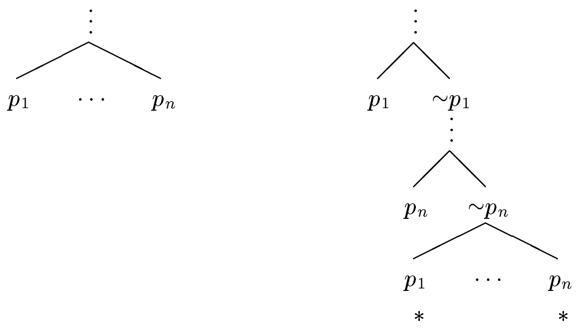


Figure 7.5: Casting tableau expansion into cut normal form.

Proposition 7.18 Every minimal closed clausal tableau T in cut normal form is a regular connection tableau with atomic cut.

Proof By Proposition 2.28, T must be regular. Now consider any tableau clause c not resulting from a cut inference. Assume, indirectly, that c be not connected to its predecessor node. Since, by the cut normal form, all literals in c occur complemented on the branch from the root up to N, one could prune T by deleting the cut step above c and shifting c one level up and the tableau would still be closed. This contradicts the minimal size assumption for T. □

So, with uncontrolled atomic cut, the tableau calculi are equally powerful. More interesting is the investigation of controlled versions of the cut rule like factorization, folding up, and folding down.

Proposition 7.19 Tableaux with atomic cut can linearly simulate tableaux with factorization.

Proof Given a closed tableau with factorization, each factorization step of a node N_1 with a node N_2, both labelled with a literal L, can be simulated as follows. First, perform a cut step with $\sim L$ and L at the ancestor N of N_2, producing new nodes N_4 and N_5; thereupon, move the tableau part formerly dominated by N below N_4; then, remove the tableau part underneath N_2 and attach it to N_5; finally, perform reduction steps at N_1 and N_2. The simulation is graphically shown in Figure 7.6. □

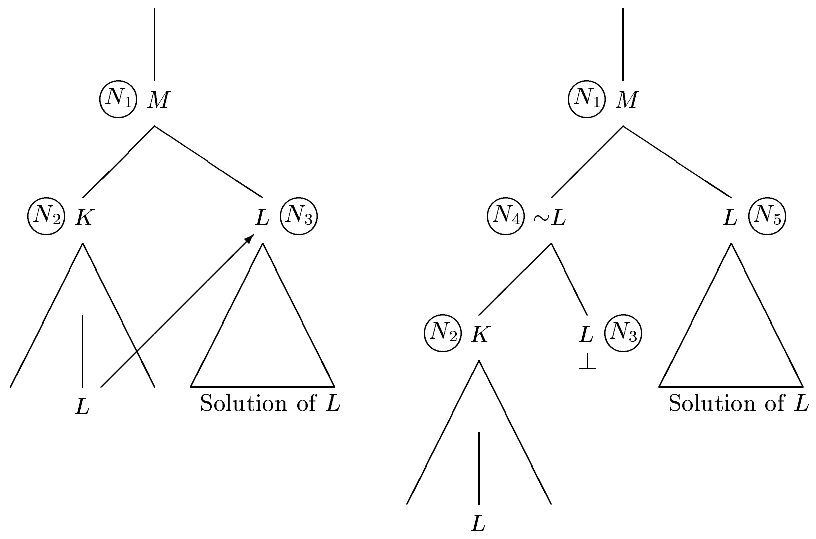


Figure 7.6: Simulation of factorization by cut.

Proposition 7.20 (Regular) (path) (connection) tableaux with folding down (or Prawitz reduction or complement splitting) and (regular) (path) (connection) tableaux with factorization linearly simulate each other.

Proof For the part of the linear simulation of factorization by folding down, consider any closed tableau T with factorization and either of the given structural conditions. Let $\prec$ be the factorization dependency relation of T. Now we construct a tableau T' by simply using $\prec$ as dependency ordering for folding down, as follows. Whenever a clause is attached, for every new node N_i, we label the corresponding edge with the literals at all brother nodes N_j which are smaller than N_i in $\prec$. This permits that any factorization step of a node N with a node N' in T can be simulated by a reduction step at N in T', since the literal at N' was before folded down to the branch above N. The other direction is similar, by just taking the dependency ordering on the node families used for folding down as factorization dependency relation. □

Proposition 7.21 Clausal tableaux cannot polynomially simulate regular strong connection tableaux with factorization.

Proof We may again use Example 7.5, which has only exponential closed clausal tableaux. For this class a closed regular strong connection tableaux of linear size can be constructed using a Prolog style branch selection function. Whenever a subgoal with literal L in a tableau clause c is extended, then one uses the clause which is identical to c except for the sign of the literal L. This guarantees strong connectedness. On the other hand, this also permits that the subgoals in c to the right of L can be used for factorization of the subgoals in the new clause, as illustrated in Figure 7.7. It is straightforward to recognize that this way every input clause is needed at most once as a tableau clause. By the strong independence of the selection function, the same proof can be constructed by any other selection function. □

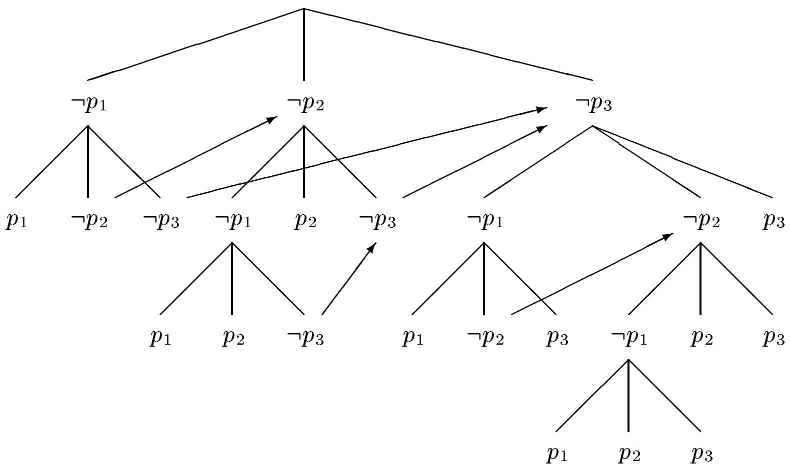


Figure 7.7: Closed regular strong connection tableau with factorization of linear size for Example 7.5, for the case of $n = 3$.

Next, we come to the folding up rule. While it is an open problem whether clausal tableaux with factorization can polynomially simulate clausal tableaux with folding up or atomic cut, for connection tableaux, the folding up rule is properly stronger concerning deductive power than the factorization rule.

Proposition 7.22 Path connection tableaux with factorization cannot polynomially simulate connection tableaux with folding up.

Proof We use the formula class specified in Example 7.23. The size of any such clause set is of the order $\mathrm{O}((m+n)^3)$. It can easily be recognized that any closed path connection tableau for an instance of this class has $\sum_{i=1}^{n} m^i$ branches when we start with the top clause $\neg p_1^1 \vee \cdots \vee \neg p_m^1$. Also, factorization is not possible under this assumption, since no two subgoals N_1 and N_2 with identical literals

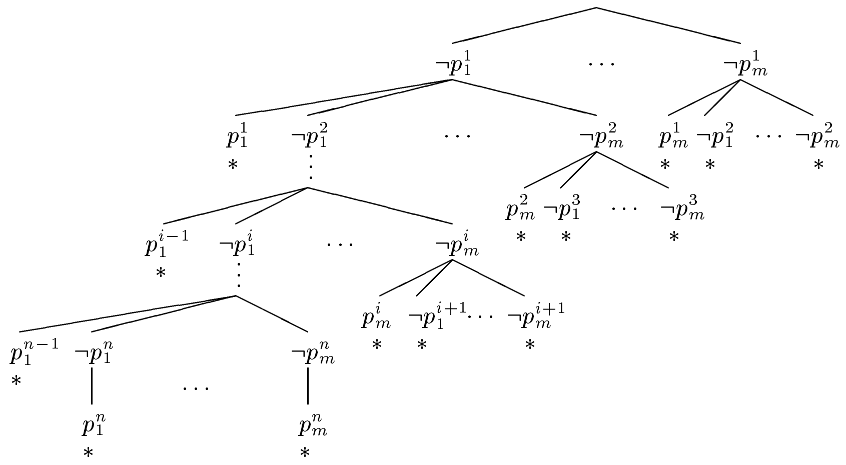


Figure 7.8: Linear connection tableau with folding up for Example 7.23.

can occur where N_2 is a brother node of (an ancestor of) N_1. Therefore, the example class is intractable for path connection tableaux with factorization, for this specific top clause. However, there exist closed connection tableaux with factorization of linear size if, e.g., one of the clauses

$$p_i^{n-1} \vee \neg p_1^n \vee \cdots \vee \neg p_m^n, \quad 1 \leq i \leq m$$

is taken as top clause. In order to obtain an unsatisfiable class which is intractable for *any* selection of the top clause, we can apply the same *duplication trick* as used in the proof of Proposition 5.14. We modify the class given in Example 7.23 by adding m literals $\neg q_1^1, \ldots, \neg q_m^1$ to the top clause, and by adding consistently renamed copies (replace p with q) of the other clauses. For the resulting clause set, it does not matter with which clause we start, since now in *any* minimal closed path connection tableau a subtableau must occurs which is isomorphic to the entire closed tableau for the initial clause set. Consequently, the new example class is intractable for path connection tableau with factorization.

On the other hand, any formula from the class has a linear closed connection tableau with folding up with the first clause as top clause, as illustrated in Figure 7.8. Since never reduction steps are needed (the clause set is Horn), the literal of any solved node can be folded up to the root and used for E-reduction steps afterwards. This involves that the resulting proof tree contains every input clause exactly once as a tableau clause. □

Example 7.23 For any two natural numbers m, n, let S_m^n denote the following set of clauses:

$$\begin{array}{ll}
\neg p_1^1 \vee \cdots \vee \neg p_m^1, & \\
p_i^1 \vee \neg p_1^2 \vee \cdots \vee \neg p_m^2, & \text{for } 1 \leq i \leq m \\
p_i^{n-1} \vee \neg p_1^n \vee \cdots \vee \neg p_m^n, & \text{for } 1 \leq i \leq m \\
p_i^n, & \text{for } 1 \leq i \leq m.
\end{array}$$

In fact, path connection tableaux with factorization cannot even polynomially simulate pure clausal tableaux (without atomic cut), which at first sight is a very surprising result.

Proposition 7.24 Path connection tableaux with factorization cannot polynomially simulate clausal tableaux.

Proof Any clause set S_m^n specified in Example 7.23 is a Horn clause set. It is well-known that, for any unsatisfiable Horn set, there is a unit hyper resolution refutation [Chang and Lee., 1973] of linear complexity [Dowling and Gallier, 1984]. By Proposition 7.47, in propositional logic clausal tableaux can linearly simulate unit hyper resolution. Consequently, the class S_m^n has closed clausal tableaux of linear size. □

Next, we come to the consideration of the folding up rule. Although more powerful than factorization, folding up is still a hidden form of the cut rule.

Proposition 7.25 Clausal tableaux with atomic cut linearly simulate clausal tableaux with folding up.

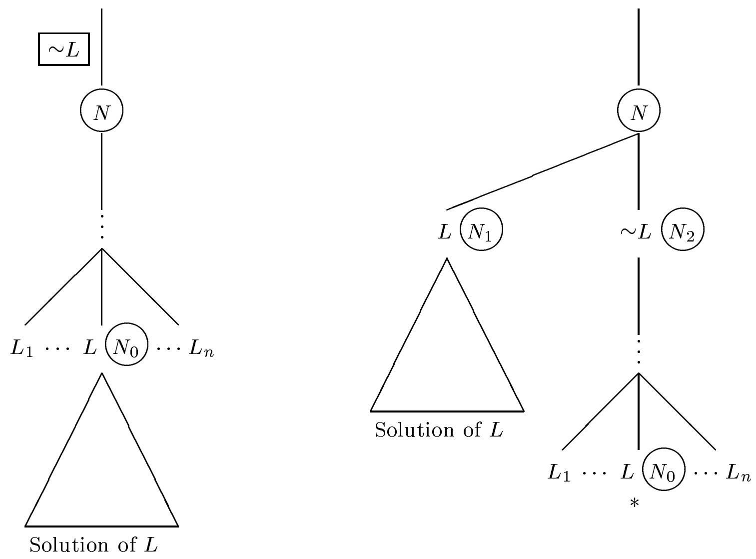


Figure 7.9: Simulation of folding up by cut.

Proof Given a tableau derivation with folding up, each folding up operation at a node N_0 adding the complement $\sim L$ of the literal L at a solved node to the label of an edge above a node N (or to the root), can be simulated as follows. Perform a cut step at the node N with the atom of L as cut formula, producing two new nodes N_1 and N_2 labelled with L and $\sim L$, respectively; shift the solution of L

from N_0 below the node N_1 and the part of the tableau previously dominated by N below its new successor node N_2; finally, perform a reduction step at the node N_0. It is apparent that the open branches of both tableaux can be injectively mapped to each other such that all pairs of corresponding branches contain the same leaf literals and the same sets of path literals, respectively. The simulation is graphically shown in Figure 7.9. □

Corollary 7.26 Regular connection tableaux with atomic cut can linearly simulate clausal tableaux with folding up.

Proof Immediate from the Propositions 7.25, 7.18, and 7.17. □

The possibility of a linear simulation in the other direction, atomic cut by folding up, is also quite straightforward for the case of clausal tableaux without connection conditions. When the connection condition is added, then the simulation in the other direction is very hard to prove. There is a proof sketch of this result in [Mayr, 1993]. It uses involved intermediate calculi and is very difficult to follow. Here we give a simpler proof which employs no additional concepts. The gist of this method is that we strongly exploit the fact that the proof size is preserved for any selection function, as long as it is a depth-first one.

Proposition 7.27 If T is a closed clausal, path connection, or connection tableau with folding up for a set of clauses S and σ is any depth-first subgoal selection function, then there exists a closed clausal, path connection, or connection tableau, respectively, with folding up for S with the same top clause and constructed according to σ such that T and T' have the same size.

Proof The key notion used in the proof is the following generalization of the concept introduced in Definition 2.60. Given any selection function σ, we say that a tableau T *is constructed according to σ up to inference n* if, for any inference $i \leq n$, the node selected in T at inference i is identical to the one selected by σ. We proof the existence of the desired tableau T' by induction on the number n. The induction base is trivial. For the induction step, assume that there exists a tableau T_n with the desired properties which is constructed according to σ up to inference n.

1. If the subgoal selected in inference $n+1$ is the same as the one selected by σ, then $T_{n+1} = T_n$.

2. Otherwise, let N with literal L be the subgoal selected by σ. In case, in T_n, at the subgoal N later an expansion, path extension, extension, or an ordinary reduction step is performed, then we simply give priority to this inference and let the rest unchanged. The resulting tableau T_{n+1} obviously satisfies the desired properties.

3. The remaining critical case is the one in which, in T_n, N is solved later by an E-reduction step with a literal folded up *after* inference $n+1$, since this

E-reduction step is not yet possible at inference $n + 1$. According to the definition of folding up, to this folding up step there corresponds a closed subtableau in T_n with root, say, N' and literal L. This involves that the subtableau below N' must be completely closed before N is selected. Now σ is a depth-first selection function and, by the induction assumption, T_n is constructed according to σ up to inference n. Therefore, N' must be selected in T_n after inference n. This enables that we can modify the tableau T_n by detaching the tree below N' and attaching it to N while preserving the size of the tableau, as depicted in Figure 7.10. As the next inference step, we perform now the respective extension step at N. Furthermore, we delay the selection of N' until the complete closure of N, and the folding up of its literal L to the same path position as in T_n. This enables that N' can be solved by an E-reduction step later. The resulting tableau T_{n+1} satisfies the desired properties.

In all three cases, T_{n+1} has the same size and T_{n+1} is constructed according to σ up to inference $n+1$. □

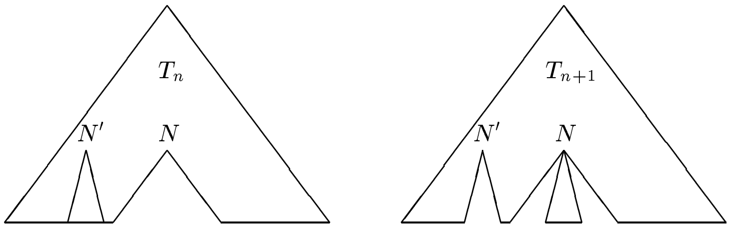


Figure 7.10: Possible modification when changing the selection function.

The other ingredient for proving the linear simulation of atomic cut by folding up is the following important property of connection tableaux.

Lemma 7.28 (1-level clause lifting) If T is a closed connection tableau with folding up containing a tableau clause c at depth $n > 0$, then there exists a closed connection tableau with folding up T' in which c occurs at depth $n-1$, and T and T' have the same size.

Proof The proof is by a transformation of the initial tableau into one with the desired properties. In order to structure the proof, we perform the transformation in two steps. In the first step, we modify T by using a depth-first selection function which satisfies a certain condition. In the second step, we perform the final modification on this tableau.

Step 1. Let c occur at depth n on a branch B in the given tableau T. Let further $p_1, \ldots, p_n$ be the literals at the nodes of B and $c_1, \ldots, c_n$ the sequence of tableau clauses along B, i.e., c_1 is the top clause in T and $c_n = c$. By Proposition 7.27, one may modify T to a tableau T'' of the same size by any depth-first selection function. We use the same transformation procedure as in the proof

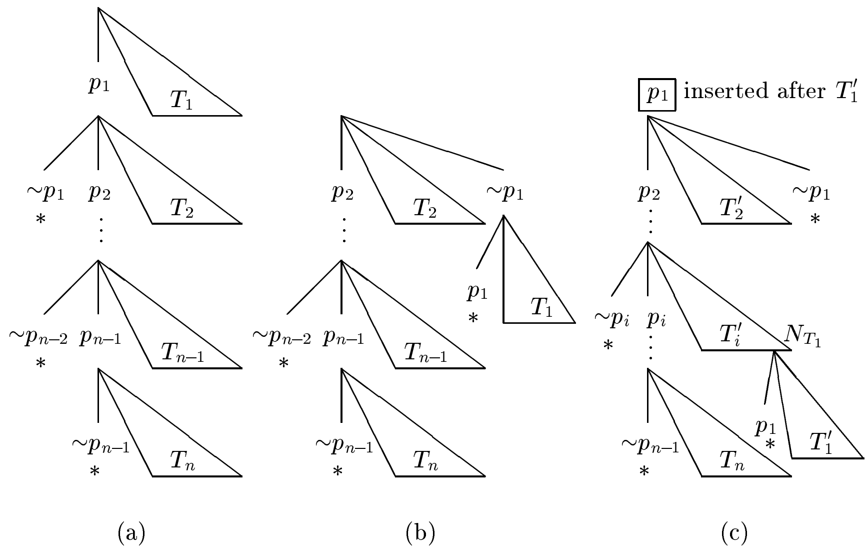


Figure 7.11: 1-level clause lifting in connection tableaux with folding up.

of this proposition and a depth-first selection function which, in every tableau clause c_i $(1 \leq i < n)$, selects the p_i-node first. Accordingly, after the start step, we successively perform $n-1$ extension steps at the p_i-nodes using the clauses c_{i+1}, respectively. Afterwards, the subtableaux $T_n, \ldots, T_1$ are solved, in this order. The structure of the tableau T'' is illustrated in Figure 7.11 (a).

Step 2. From T'' a tableau T' with c_n at depth $n-1$ may be constructed as follows. First, instead of c_1 we attach c_2 as top clause. Then we perform the same $n-2$ extension steps as in T'. Afterwards we extend the $\sim p_1$-subgoal in the new top clause c_2 using c_1. Finally, we attach the subtableaux $T_1, \ldots, T_n$ at the respective nodes, as shown in Figure 7.11 (b), and assume that the subtableaux are solved in the order $T_n, \ldots, T_1$, as in T''. It is clear that this leads to a closed tableau only if the p_1-path node is not used for a reduction step in one of the $T_2, \ldots, T_n$.

Otherwise, we have to repair the current tableau. Let T_i be the subtableau containing the first encountered subgoal N_{T_1} to be solved by a reduction step using the p_1-path node. Now we simply shift the subtableau below the $\sim p_1$-node below N_{T_1}. Afterwards, we attempt to replay the construction of T_1 as in T''. This is easily possible except for subgoals which were solved by E-reduction steps using literals folded up to the root *before* the solution of T_1 in T'', since those literals may not yet be available. Whenever such a subgoal N with literal L is encountered, we restructure the current tableau. To L there corresponds a closed subtableau T_L with root N' outside T_1. Since, by assumption, after solving N' in T'', its literal L was folded up to the root, the solution of T_L did not use any path literals between the root and N' for reduction steps. Now, in the solution of T_i in T'', the p_1-node was used. So N' cannot be one of the nodes between the root and the p_{i-1}-node at the root of T_i. Therefore, N' must be in one of the subtableaux $T_{i-1}, \ldots, T_2$, and we can modify the current subtableau below N_{T_1} by shifting T_L from N' to N without increasing the tableau size. Then we continue with the next subgoal in T_L. After we have completely solved N, we fold it up to the root. This permits that N' can afterwards be solved by an E-reduction step. The replay of T_L may require further such modification steps, since the order in which folded up literals are needed may be different from the order in which their solutions were generated in T''. But this will cause no problems, since in the replay of T_L no literals between the the root and the root of T_L are needed. If k is the number of literals folded up to the root of T'' before T_1 was tackled, then it is clear that after at most k such modifications we have completely solved the node N_{T_1}. Afterwards, we fold up its literal $\sim p_1$ to the root. This permits that all subsequently encountered subgoals needing p_1 as a path literal can be solved by an E-reduction step. The process is exemplarily illustrated in Figure 7.11 (c). Since all modifications preserve the tableau size and the connectedness condition, the resulting tableau T' is as desired. □

Proposition 7.29 (Clause lifting) If c occurs as a tableau clause in a closed connection tableau T with folding up, then there exists a closed connection tableau T' with folding up with top clause c such that T and T' have the same size.

Proof If n is the minimal depth of c in T, then $n-1$ applications of Lemma 7.28 will do. □

It is obvious that this lifting possibility also holds for any subtableau T of a tableau T', i.e., any clause c in T may be moved on top of T while preserving the size of T and the connection conditions *inside* T, and without affecting the rest of the tableau T' (of course, the connectedness to the root of T may be lost). This robustness concerning the reversion of tableaux does *not* hold without folding up.

Theorem 7.30 (Cut elimination for connection tableaux) If T is a closed connection tableau with folding up and atomic cut for a set of clauses S, then there exists a closed connection tableau with folding up T' for S such that the size of T' is less or equal to the size of T.

Proof The proof is by induction on the number of cuts performed in T, which we call the *cut degree* of T. The induction base, for cut degree 0, is trivial. For the induction step, assume the result to hold for any tableau of cut degree n. Now consider any tableau $\mathcal{T}$ with cut degree $n+1$. First, by Proposition 7.27, we may transform the tableau into a tableau of the same size and constructed with a depth-first selection function (the case of cuts is also captured by this proposition, since one may simulate atomic cuts by simply adding the respective tautological clauses). We show how the last cut performed in this tableau may be eliminated. Let N with literal K be the node at which this cut step is performed, L the cut literal, and T_1 and T_2 the subtableaux dominated by the two cut nodes, respectively. For an illustration, consider Figure 7.12. Our aim is to find a tableau

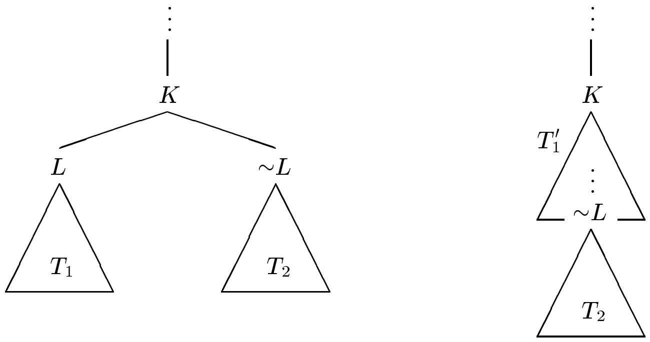


Figure 7.12: Cut elimination in connection tableaux with folding up.

clause c below N containing the literal $\sim K$, in order to permit an extension step in place of the cut. Obviously, the interesting situation is the one in which T_1 and/or T_2 are complex subtableaux, since otherwise cut elimination is trivial.

Pruning. First, we consider the case in which $\sim K$ does neither occur in T_1 nor in T_2. Then, we prune the tableau by canceling out the tableau clause of the node N and attach the entire subtableau dominated by N to the predecessor of N. The problem occurs that in the unpruned tableau one may have folded up K or some literals occuring during the solution of brother nodes of N, and one may have used the resulting path literals for E-reduction steps on later selected subgoals N_i. Obviously, after the pruning these folded up literals are no more available. This can be captured as follows. Whenever such a subgoal N_i is selected and the respective folded up literal is not yet available, then we simply replay the solution of this literal in the pruned part, in the same manner as in Step 2 of the proof of the 1-level clause lifting lemma (Lemma 7.28). It is obvious that this way the size of the resulting tableau decreases just by the size of the node family of N. This pruning step is repeated until it is no more applicable.

Cut elimination. Afterwards, either the cut has moved up to the top of the tableau or the literal $\sim K'$ at the node N' above the cut does occur in one of the subtableaux T_1 or T_2.

1. In the former case, instead of performing the cut at the root, we attach the tableau $T_1' = T_1$, which corresponds to a start step.

2. Otherwise, we may assume w.r.g. that $\sim K'$ occurs in some tableau clause c of T_1 and that the L-node is selected before the $\sim L$-node (otherwise we would simply switch to an appropriate selection function including the respective modifications). Now, by Proposition 7.29 (the subtableau variant), we may revert the subtableau T_1 and bring c on top. Next, instead of performing the cut at the K'-node we attach the reverted tableau T_1', which corresponds to an extension step.

Obviously, some branches in T_1' may be open now, namely, the branches whose nodes were closed by a reduction step with the cut literal L, which is now missing in the path. Let N_L be the first such encountered node. We attach the subtableau T_2 at N_L, which corresponds to an extension step, since, by assumption, there is no cut in T_2. Afterwards we replay the construction of T_2 as usual, i.e., possibly including further modifications as in Step 2 of the proof of Lemma 7.28. After the solution of T_2, the literal L may be folded up at least to the edge above the node N' or to the root in case T_1' is immediately below the root. This enables that any later selected subgoal in T_1' with literal L may be solved by an E-reduction step using the respective path literal. The result of the entire operation is a connection tableau with folding up and cut for S with cut degree n and a size that is properly smaller than the size of $\mathcal{T}$.

Consequently there exists a closed cut-free connection tableau with folding up T' for S which is not larger than T. □

Corollary 7.31 Connection tableaux with folding up can linearly simulate clausal tableaux with cut.

Proof Immediate from Theorem 7.30 and Proposition 7.17. □

7.3 Semantic Trees and Resolution

Beside the relation of tableau refinements with each other, it is also important how tableaux are related with other proof systems. We consider two families of calculi, resolution systems and semantic tree procedures. Both families operate with a single inference rule, namely, a condensed variant of the atomic cut rule. The difference between both families is that resolution systems work by a *forward* application of the cut rule whereas semantic tree procedures use the cut rule in a *backward* manner.

First, we consider the method of semantic trees, which is a natural improvement of truth tables. Semantic trees [Robinson, 1968, Kowalski and Hayes, 1969] are typically used as a representation tool for analyzing first-order proof procedures of the resolution type. A binary version of semantic trees turns out to be an excellent basis for propositional proof procedures of the Davis, Putnam, Loveland, Logemann (*DPLL*) type [Davis et al., 1962]. The simple motivation for the method is that a formula can often be given a definite truth value on the basis of merely a *partial* interpretation. In such a case, the truth value of the partial interpretation V of the formula is the same as the truth value of all total interpretations which are extensions of V. This way, in one inference step, instead of checking single interpretations, entire *sets* of interpretations can be examined. This potential for shortening truth tables was also noticed by Kleene in [Kleene, 1967]. Semantic trees generalize his method.

Definition 7.32 (Semantic tree) A *semantic tree for* a set of clauses S is a binary rooted tree with a total labelling of its edges and a (possibly partial) labelling of its leaf vertices, meeting the following conditions.

1. Each pair of edges originating in the same vertex is labelled with a ground literal L respectively its complement where L is an instance of a literal in some clause of S.

2. Any leaf node N may be labelled with a ground instance c of a clause in S, provided that all literals in c occur complemented on the branch leading from the root up to N.

A semantic tree is called *closed (for S)* if every leaf node is labelled with a ground instance of a clause (in S).

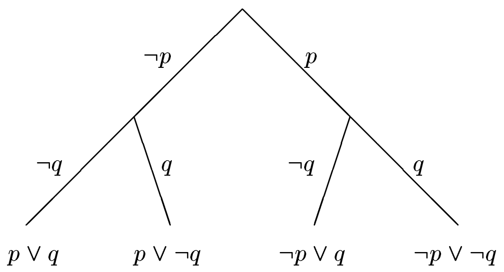


Figure 7.13: Closed semantic tree for $\{p \vee q, p \vee \neg q, \neg p \vee q, \neg p \vee \neg q\}$.

An example of a closed semantic tree is depicted in Figure 7.13. There is the following relationship between semantic trees and tableaux with cut.

Proposition 7.33 Clausal tableaux with atomic cut and semantic trees can linearly simulate each other.

Proof Any semantic tree is basically a tableau in cut normal form, and vice versa. For illustration compare Figure 7.13 with Figure 7.4. □

Next, we consider resolution, which consists of a single inference rule. We start with the fragment of resolution for ground clauses, which is the dual of Quine's *consensus* rule [van Orman Quine, 1955].

Definition 7.34 (Propositional resolution rule) Given two clauses c and c', with $c - c'$ we denote the clause obtained from c when removing all literals contained in c'. Let c_1 and c_2 be two clauses with c_1 containing a literal L and c_2 containing $\sim L$. The clause $c_1 - L \vee ((c_2 - \sim L) - (c_1 - L))$ is called a *propositional resolvent* of c_1 and c_2 *wrt.* L. c_1 and c_2 are termed *parent clauses* of the resolvent.

Example 7.35 The clause $p \vee q \vee \neg r$ is a propositional resolvent of $p \vee \neg r \vee s$ and $p \vee \neg s \vee q$.

In the first-order case, the resolution rule is more complicated, since it incorporates renaming, *factoring*, and unification.

Definition 7.36 (Resolution rule) Given two clauses c_1 and c_2, let c_2' be a renaming of the variables in c_2 wrt. c_1. Let further S_1 be a set of literals contained in c_1 and S_2 a set of literals contained in c_2' such that there is a minimal unifier σ for the set $S_1 \cup \{\sim L : L \in S_2\}$. Then the propositional resolvent of the clauses $c_1\sigma$ and $c_2'\sigma$ is called a *resolvent* of c_1 and c_2 *wrt.* $L\sigma$.

Example 7.37 The clause $P(a, a)$ is a resolvent of the clauses $P(a, x) \vee Q(x, a)$ and $P(x, z) \vee \neg Q(x, y) \vee \neg Q(y, z)$ wrt. $Q(a, a)$.

Definition 7.38 (Resolution proof) A *resolution deduction* or *proof of* a a clause c_n *from* a set of clauses S is a finite sequence $D = (c_1, \ldots, c_n)$ of clauses such that, for each clause c_i $(1 \leq i \leq n)$, either $c_i \in S$ or c_i is a resolvent of two parent clauses which both occur in the sequence D at positions $< i$. A resolution proof of the empty clause $\perp$ from a set S is called a *resolution refutation of* S.

In order to exhibit the close relation of resolution with semantic trees, it will also be convenient to have a graph notation of a resolution proof.

Definition 7.39 (Resolution dag) A *resolution dag* of a clause c *from* a set of clauses S is a finite directed acyclic graph T which is rooted and binary branching, and whose nodes are labelled with clauses in such a way that

1. the root of T is labelled with c,
2. the leaf nodes of T are labelled with clauses from S, and
3. the clause at any non-leaf node N in T is a resolvent of the clauses at the successor nodes of N.

If the dag t of a resolution dag T is a tree, T is named a *resolution tree.*

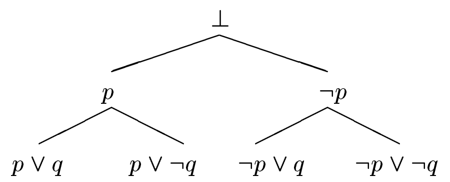


Figure 7.14: A tree resolution refutation from $\{p \vee q, p \vee \neg q, \neg p \vee q, \neg p \vee \neg q\}$.

Proposition 7.40 The semantic tree method can linearly simulate tree resolution.

Proof Let T be a resolution tree of the empty clause $\perp$ from a set of clauses S. First, since T is a tree, it is straightforward to realize that we can make T into a propositional resolution tree T' for a set S' consisting of ground instances of clauses in S by simply inheriting the unifiers from the root down the tree.

Furthermore, when using symbol dags, the size of T' is linear in the size of T. Now, T' is basically a semantic tree for S, one simply has to add additional edge labels and disregard the labels of the non-leaf nodes. For an illustration compare the Figures 7.14 and 7.13. □

Proposition 7.41 Tree resolution can quadratically simulate semantic trees.

Proof Let T be a closed semantic tree for a set of ground clauses S. Construct a propositional resolution refutation of S by iteratively performing the following procedure on T.

- In the current tree, select any unlabelled node N whose two successor nodes N_1 and N_2 are labelled with ground clauses c_1 and c_2, respectively. Let L and $\sim L$ be the literals at the respective edges. If $\sim L$ is in c_1 and L in c_2, then label N with the respective resolvent of c_1 and c_2. Otherwise, prune the tree by connecting the edge incident to N to N_1 if $\sim L$ is not in c_1, or else to N_2.

It is straightforward to realize that the procedure generates a propositional tree resolution refutation of S with a size equal or smaller than T. Because of the explicit representation of the intermediate clauses in the resolution tree, its size may be quadratic in the size of T. □

Consequently, concerning minimal proof lengths and when abstracting polynomial differences, semantic trees and tree resolution are equivalent proof systems. The simulation technique used in the proof above also permits the straightforward construction of a complete resolution procedure which can do with quadratic space. In practice, however, the quadratic overhead of tree resolution wrt. semantic trees is a clear argument in favour of semantic trees. This explains the success of semantic tree procedures like the DPLL method. On the other hand, tree resolution cannot polynomially simulate unrestricted resolution where deductions may be dags [Reckhow, 1976]. So it is clear that semantic trees cannot polynomially simulate resolution. This is an argument for deduction concepts using dags. The most successful dag-based framework of propositional decision procedures are binary decision diagrams (BDD's). Concerning minimal proof lengths BDD's can actually be viewed as refinements of resolution calculi.

7.4 Results for First-Order Clausal Tableaux

When moving from propositional logic to the first-order case, then normally even more complexity differences may be identified. In particular, this holds for resolution calculi. For example, in [Letz, 1993a], it is shown that, in the first-order case, *linear* resolution cannot polynomially simulate unrestricted resolution, a question which is yet unsettled in the propositional case. A resolution deduction is called a *linear* or *ancestor* resolution deduction if any resolvent is used as a parent clause in the subsequent resolution step.

The tableau calculi with standard unification, however, have the following property.

Proposition 7.42 The Herbrand complexity of any unsatisfiable clause set S is a lower bound to the size of any closed clausal tableau with atomic cut.

Proof The clausal tableau calculus, with or without atomic cut, has the following *ground projection property.* If T is a closed clausal tableau (with cut) for a set S, then any ground instance $T\sigma$ is a closed tableau (with cut) for an unsatisfiable set of ground instances of clauses in S. So, we may choose a substitution σ which maps every variable in T to the same constant from the Herbrand universe of S. This guarantees that T and $T\sigma$ have the same size. By the soundness of the calculus and the substitution rule, the set of tableau clauses of $T\sigma$ must be an unsatisfiable set of ground instances of clauses in S. Consequently, $S\sigma$ and hence S cannot be smaller than the Herbrand complexity of S. $\square$

So there are at first sight no difference between the propositional and the first-order case. An obvious first-order feature of clausal tableaux which destroys the ground projection property is the local unification mechanisms. This was already illustrated in Figure 6.5. We will consider now whether this feature can really cause superpolynomial differences. For the subsequent investigations we build on a class of clause sets used, e.g., in [Letz, 1993a, Plaisted and Zhu, 1997], which encodes a binary counter.

Example 7.43 For any n, let S_n be the set of clauses of the following form.

$$\begin{array}{rlcr}
c_0: & & & P(0,\ldots,0),\\
c_1: & \neg P(x_1,\ldots,x_{n-1},0) & \vee & P(x_1,\ldots,x_{n-1},1),\\
c_2: & \neg P(x_1,\ldots,x_{n-2},0,1) & \vee & P(x_1,\ldots,x_{n-2},1,0),\\
 & & \cdots & \\
c_{n-1}: & \neg P(x_1,0,1,\ldots,1), & \vee & P(x_1,1,0,\ldots,0),\\
c_n: & \neg P(0,1,\ldots,1), & \vee & P(1,0,\ldots,0),\\
c_{n+1}: & \neg P(1,\ldots,1) & &
\end{array}$$

where P denotes an n-ary predicate symbol, and 0 and 1 denote different constants.

Proposition 7.44 Any clause set S_n specified in Example 7.43 has a resolution refutation of a linear size.

Proof Note that, for any $1 \leq i < n$, one can deduce the clause

$$\neg P(x_1,\ldots,x_i,0,\ldots,0) \vee P(x_1,\ldots,x_i,1,\ldots,1)$$

in two resolution steps using the clause

$$\neg P(x_1,\ldots,x_{i+1},0,\ldots,0) \vee P(x_1,\ldots,x_{i+1},1,\ldots,1)$$

derived before and the input clause

$$\neg P(x_1, \ldots, x_i, 0, 1, \ldots, 1) \vee P(x_1, \ldots, x_i, 1, 0, \ldots, 0).$$

Consequently, there is a resolution proof of $2n$ inference steps. This proof is even an ancestor resolution proof. □

Let us consider now the Herbrand complexity of an S_n, i.e., the size of a minimally unsatisfiable set of clauses each of which is an ground instance of a clause in S_n.

Proposition 7.45 The class of clause sets specified in Example 7.43 has an exponential Herbrand complexity.

Proof Any unsatisfiable set of ground instances of clauses in an S_n must contain *all* Herbrand instances of clauses in S_n, otherwise one could easily construct a Herbrand model. Since in total these are $2^n + 1$ clauses, the class has an exponential Herbrand complexity. □

The class of Example 7.43 has no polynomial closed tableaux with local unification, but we may use a modification of this class, which has polynomial unit hyper resolution proofs. This will help, because of the following simulation result.

Proposition 7.46 Clausal tableaux with local unification can linearly simulate unit hyper resolution.

Proof Let $D = c_1, \ldots, c_n$ be any unit hyper resolution deduction from a set of clauses S. We prove, by induction on the number of inference steps n in D, that there exists a tableau T with at most one open branch B such that

1. the size of T is linear in the size of D, and
2. every unit clause occurring in S or D appears on the branch B.

The induction base holds trivially. For the induction step, let $D = c_1, \ldots, c_n, c_{n+1}$ be any unit hyper resolution deduction. By the induction assumption, there exists a clausal tableau with local unification T_n which satisfies the two conditions for the deduction $c_1, \ldots, c_n$. The nontrivial case is the one in which the clause c_{n+1} is a unit hyper resolvent of a nucleus clause c_i ($i \leq n$) and electrons appearing before c_{n+1} in D. This can be simulated in the tableau by an expansion step with c_i and subsequent local reduction steps with the electrons on the branch B. □

Obviously, in the propositional or ground case, there is no difference between ordinary and local unification.

Proposition 7.47 For ground clause sets, clausal tableaux can linearly simulate unit hyper resolution.

Now, we have all ingredients at hand for proving the desired result.

Proposition 7.48 Clausal tableaux cannot polynomially simulate clausal tableaux with local unification.

Proof We use the following modification of the above example.

Example 7.49 For any n, let S_n be the set of clauses as specified in Example 7.43. Then the clause set S'_n is defined as follows. For any non-unit clause

$$\neg P(\alpha_1, \ldots, \alpha_n) \vee P(\beta_1, \ldots, \beta_n)$$

in S_n, S'_n contains the unit clause

$$I(p(\alpha_1, \ldots, \alpha_n), p(\beta_1, \ldots, \beta_n)).$$

To the two unit clauses in S_n, there correspond the two unit clauses

$$I(\top, p(1, \ldots, 1)) \text{and} I(p(1, \ldots, 1), \bot)$$

in S'_n. Additionally, S'_n contains the three-literal clause

$$\neg I(x, y) \vee \neg I(y, z) \vee I(x, z)$$

and the unit clause $\neg I(\top, \bot)$. So, in S'_n there occur four different constants $(0, 1, \bot, \top)$, one n-ary function symbol (p), and one binary predicate symbol (I).

Any clause set S'_n represents a certain meta-level representation of S_n. The natural interpretation of the predicate symbol I is the material implication relation. Accordingly, the three-literal clause expresses the transitivity of material implication. S'_n is designed in such a manner that every resolution step between two clauses c_1, c_2 (deduced) from S_n can be simulated within S'_n by performing a unit hyper resolution step with the transitivity clause as nucleus and the corresponding clauses c'_1, c'_2 as electrons. To the empty clause in the old set, there corresponds the clause $I(\top, \bot)$. This explains the necessity of the last clause $\neg I(\top, \bot)$, which is only used to derive the *real* empty clause. The advantage of the new clause set is that *any* resolution derivation from S_n can be linearly simulated by a unit hyper resolution derivation from the set S'_n. In particular, S'_n has a unit hyper refutation of linear size. By Proposition 7.46, S'_n has a closed tableau with local unification of linear size. But the Herbrand complexity of any S'_n is still exponential. □

So local unification can significantly imporve the deductive power of free-variable tableaux. But the use of local unification also has an interesting other effect on tableau proofs. With standard unification, the number of inference steps of a tableau proof gives a reliable measure of the actual complexity of the proof. More precisely, any n-step tableau proof for a formula F has a polynomial size wrt. the size of F plus n. This property is lost when local unification is applied. Using the same technique as in Example 7.49 and the prime number example from [Letz, 1993a], one can easily construct a formula with violates this property.

Chapter 8

Complexities of Search Spaces

While the last chapter was devoted to lengths of minimal proofs, in this chapter we will present fundamental complexity results on the sizes of the search spaces of tableau calculi. Concerning the complexities of search spaces, the work of Plaisted and Lee is of high significance [Plaisted, 1994, Plaisted and Zhu, 1997]. They analyze a wealth of different theorem proving strategies like resolution, model elimination or clause linking according to their worst-case complexities. Plaisted and Lee also compare the search spaces of strategies based on calculi with different behaviour concerning minimal proof lengths, i.e., calculi which cannot polynomially simulate each other. Since the size of a search space is normally at least exponential in the size of the length of the minimal proof, it is clear that this way one may obtain big differences in complexity.

The difference of the results presented there with ours is that we concentrate on the finer differences *within* one framework, namely, the tableau-based paradigm. We also attempt to keep proof length and search space issues separate, in order to precisely identify the influence of a specific refinement on the corresponding search spaces.

8.1 Complexities of Iterative-Deepening Bounds

When using the tableau enumeration approach, the most successful technique is to use iterative deepening methods according to completeness bounds, which are tableau complexity measures. For any tableau search space $\mathcal{S}$, a completeness bound defines a sequence of finite initial segments $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \ldots$ of $\mathcal{S}$. It is important to know about the worst-case complexities of the $\mathcal{S}_i$ with respect to the size of the underlying input formula and tableau calculus. Furthermore, it is instructive to know whether the existence of a proof in some $\mathcal{S}_i$ can be related with some problem class in the area of computational complexity. These topics will be addressed in this section, for the most important families of completeness bounds, the inference, the depth, and the multiplicity bounds.

8.1.1 Upper Bounds of Bounded Search Spaces

We start with providing *upper bounds* for the number of tableaux contained in a certain initial segment of a search space. The values are given in dependence on the number k of clauses, the maximal clause length l and the maximal literal size s in the underlying set of clauses. The upper bounds are defined for the general clausal tableau calculus with folding up. For simplicity, we count any extension step as two steps, as an expansion and a reduction step. Furthermore, we assume that a subgoal selection function is used.

Proposition 8.1 (Inference bound estimate) The number of clausal tableaux with folding up constructed with n inferences is bounded from above by an exponential function in n and the input size.

Proof The number $i(1)$ of clausal tableaux constructed with one inference is k. The number of clausal tableaux with $\leq n+1$ inferences can be estimated by performing at most one inference step at the selected subgoal N of each tableau with $\leq n$ inferences. First, k expansion steps can be applied at N. The depth of the node N is $\leq n$. Consequently, less than $2n$ reduction or E-reduction steps can be performed at N. In total this gives the estimate

$$i(n+1) \leq i(n) \cdot (2n+k) \leq (2n+k)^n,$$

which is an exponential upper bound. The size of any tableau in such an initial segment is quadratic in n and the size of the input. □

Proposition 8.2 (Depth bound estimate) The number of clausal tableaux with folding up of depth n is bounded from above by a doubly exponential function in n and the input size.

Proof We simply use the estimate for the inference bound and exploit that any tableau of depth n has $< l^{n+1}$ nodes and hence less than l^{n+1} inferences. Therefore the number of tableaux with a depth $\leq n$ is $< (2(l^{n+1})+k)^{l^{n+1}}$, which is a doubly exponential upper bound. The size of any tableau in such an initial segment is exponential in n and the size of the input. □

Similarly, results can be obtained for the clause-dependent and the weighted-depth bounds. Note that, for the preferred parameter set of the weighted-depth bound, the increase rate is also doubly exponential in the worst case. For the multiplicity bound, we need a further restriction in order to achieve finiteness of the initial segments. As pointed out in Section 4.1.3, the strictness condition is sufficient.

Proposition 8.3 (Multiplicity bound estimate) The number of strict clausal tableaux with folding up of multiplicity n is bounded from above by a doubly exponential function in the input size plus n.

Proof According to the strictness condition, each of the n copies of any input clause must appear at most once on a branch. This limits the depth of the tableaux to $n \cdot k$ and we can use the result for the depth bound. □

8.1.2 Lower Bounds for a Concrete Example

One might argue that the given upper bounds are much to pessimistic. For this reason, we also give lower complexity bounds for concrete inputs.

Example 8.4

$$\begin{array}{l} \neg P(x), \\ P(a), \\ P(f(x,y)) \vee \neg P(x) \vee \neg P(y). \end{array}$$

Consider the set of clauses given in Example 8.4. If only the first clause is used as a start clause and a standard depth-first subgoal selection strategy is used (i.e., a model elimination strategy), then the number of deductions in each initial segment with bound n is bounded from below by the number of solution substitutions for the top subgoal. It is straightforward to recognize that the number of solution substitutions for the top subgoal is just:

- for the inference bound: the number of all binary trees with $\leq n-1$ nodes, which is an exponential function, and
- for the depth bound: the number of all binary trees with a depth $\leq n-1$, which is a doubly exponential function.

So, the aforementioned upper bounds are tight in the sense that they give the correct order of complexity. One might object, that this example is completely insignificant, since no actual deduction enumeration procedure will explore the described search spaces, but stop after the first proof has been found. In order to invalidate this argument, we simply make the set satisfiable and use the modification given in Example 8.5. For this example, any actual model elimination procedure has the aforementioned cost. Note also that, for this example, none of the developed refinements of model elimination like regularity, failure caching, etc. reduces the search effort.

Example 8.5

$$\begin{array}{ll} \neg P(x) \vee \neg Q(x), & \\ P(a), & Q(b), \\ P(f(x,y)) \vee \neg P(x) \vee \neg P(y), & Q(g(x,y)) \vee \neg Q(x) \vee \neg Q(y). \end{array}$$

Finally, let us consider the multiplicity bound. Interestingly, the depth bound result for Example 8.5 cannot be used to obtain a doubly exponential lower bound for the multiplicity case, since many of the respective tableaux of depth n contain more than n different instances of the three-literal clauses, and hence have a higher multiplicity. Under the multiplicity bound, Example 8.5 gives rise to only a single exponential growth rate. This can be recognized by considering the three-literal clauses. With start clause $\neg P(x) \vee \neg Q(x)$ and under multiplicity limit n, for every closed subtableaux T below the top $\neg P(x)$-subgoal, there are at most n different instances of the clause $P(f(x,y)) \vee \neg P(x) \vee \neg P(y)$ in T. It is straightforward to see that therefore the number of such subtableaux is bounded from above by the number of different rooted dags with $\leq n$ nodes and outdegree 2. The number of such graphs is just $\Pi_{i=1}^{n-1} i^2$ which is a single exponential function.

When using the pure connection tableau calculus without any refinements except strictness, an obvious example with a doubly exponential number of tableaux for the multiplicity bound can be obtained by simplifying Example 8.5 as follows.

Example 8.6

$$\begin{array}{ll}
\neg P(x) \vee \neg Q(x), & \\
P(a), & Q(b), \\
P(a') \vee \neg P(x) \vee \neg P(y), & Q(b') \vee \neg Q(x) \vee \neg Q(y).
\end{array}$$

For this example, in any closed subtableaux below the $\neg P(x)$-subgoal, there are at most four different instances of the three-literal clause $P(a') \vee \neg P(x) \vee \neg P(y)$. Consequently, for $n \geq 4$, the multiplicity bound has no effect. So only the strictness condition works, which merely limits the tableau depth, and the result for the depth bound applies. However, when using refinements like, e.g., regularity, the search space for this example collapses. In order to weaken the effect of regularity, one could replace a' and b' with $f(x)$ and $g(x)$, respectively. Then the search space remains doubly exponential even if regularity is used. But for both examples, (local) failure caching will have a drastic search pruning effect. This show that it is much harder to find a concrete example with a doubly exponential increase rate for the multiplicity bound when powerful search pruning methods are used.

8.1.3 Completeness Results wrt. Complexity Classes

The estimates for the number of tableaux in a certain initial segment is closely related to a certain search paradigm, namely, the actual exploration of the respective search space. However, there might exist completely different approaches to verify the existence of a tableau in a certain initial segment of a search space, and it is imaginable that those other methods are much more efficient. Therefore, we investigate the question whether the existence of a tableau proof of a certain size can be related to some fundamental other problems in the area of computational complexity. More precisely, we will provide completeness results of the respective problems for certain complexity classes. For this purpose, some other problems are needed which are known to be complete for the considered classes. For the results concerning the inference and the depth bound we will directly use reductions from nondeterministic Turing machines. For the multiplicity bound, we use quantified Boolean formulae.

It is not the place here to provide an intuitive understanding of Turing machines as, e.g., in [Aho et al., 1974]. We shall just introduce the minimal machinery that is needed for specifying one-tape nondeterministic Turing machines.

Definition 8.7 (Turing Tape) Let Σ be a finite alphabet and $b \in \Sigma$ a symbol, called the *blank*. If h is an integer and t is a mapping from the integers $\mathbb{Z}$ into Σ such that for almost all $n \in \mathbb{Z} : t(n) = b$ (we shall abbreviate $t(n)$ by writing t_z), then the pair $\langle z, t\rangle$ is called a *tape* for Σ and b; the integer z is called the *head position* of the tape.

Definition 8.8 (Turing machine) A *(nondeterministic) Turing machine* $\mathcal{M}$ is a tuple $\langle K, \Sigma, q_0, q_a, b, r, l, T\rangle$ where K and Σ are two finite sets of symbols, the *internal states* respectively the *tape alphabet* of $\mathcal{M}$; $q_0, q_a \in K$ are the *initial* respectively the *accepting state* of $\mathcal{M}$; $b \in \Sigma$, the blank; r and l are two symbols not contained in Σ, called the *left* and the *right*, respectively; and T is a finite set of quadrupels $\langle c, a, o, c'\rangle$ with $c, c' \in K$, $a \in \Sigma$, and $o \in \Sigma \cup \{r, l\}$. The elements of T are called the *instructions* of $\mathcal{M}$. If an instruction has the symbol r or l at the third tuple position, then it is called a *right* respectively a *left* instruction, otherwise it is called a *write* instruction.

Definition 8.9 (Turing configuration and input) Let $\mathcal{M} = \langle K, \Sigma, q_0, q_a, b, r, l, T\rangle$ be any nondeterministic Turing machine. Any triple $\langle z, t, c\rangle$ where $c \in K$ and $\langle z, t\rangle$ is a tape for Σ and b is called a *configuration for* $\mathcal{M}$. A configuration $\langle z, t, c\rangle$ for $\mathcal{M}$ is said to *succeed* a configuration $\langle z', t', c'\rangle$ for $\mathcal{M}$ if $\mathcal{M}$ has an instruction $\langle c, t_z, s, c'\rangle$ such that

1. either $t' = t$ and $z' = z + 1$ or $z - 1$
2. or $z' = z$ and $t' = t$ except that $t'(z) = s$.

An *input to* a Turing machine is just a tape for the tape alphabet and the blank of $\mathcal{M}$. A *derivation for* $\mathcal{M}$ is a sequence of successive configurations for $\mathcal{M}$. $\mathcal{M}$ is said to *accept* an input $\langle z, t\rangle$ *in* n steps if there is a derivation of length $\leq n$ with $\langle z, t, q_0\rangle$ as first element and a configuration with q_a at the last triple position as last element.

Subsequently, we will use different encodings of Turing machines and their inputs as logical formulae that are best suited for our purposes. In detail, we give transformations θ which map any Turing machine $\mathcal{M}$ and any input tape I into a set of Horn clauses such that $\mathcal{M}$ accepts I if and only if $\theta(\mathcal{M}, I)$ is unsatisfiable. First, we have to explain how tapes are encoded. On every tape for almost all integers z: $t_z = b$. Therefore, every configuration $\langle z, t, c\rangle$ encountered during a

Turing derivation can be finitely represented. Let l and r be the greatest tape position to the left respectively the smallest tape position to the right of position z encountered during the derivation. Then we encode the tape to the left of the head position by the (possibly empty) list term $l = [t_{z-1}, \ldots, t_l]$ and the tape to the right by $r = [t_{z+1}, \ldots, t_r]$—we use the Prolog conventions for denoting list terms. The entire configuration can then be uniquely represented by the list $[l, t_z, r, c]$.

After these preparations, we come to the definition of the first transformation, which might be called a linear encoding. The idea is to use a literal of four arguments $A(l, t_z, r, c)$ with the meaning that the list $[l, t_z, r, c]$ denotes an accepting configuration. Accordingly, any possible transition between two configurations in the Turing machine is encoded as a material implication of the form $A(l, t_z, r, c) \to A(l', t'_z, r', c')$ where $[l', t'_z, r', c']$ can be reached from $[l, t_z, r, c]$ in one step. The input and the accepting configurations can be represented with a negative and a positive unit clause, respectively.

Definition 8.10 (Linear Turing encoding) Given a nondeterministic Turing machine $\mathcal{M} = \langle K, \Sigma, q_0, q_a, b, r, l, T\rangle$ and an input $I = \langle t, z\rangle$ to $\mathcal{M}$, the *linear Turing encoding* $\theta(\mathcal{M}, I)$ is the set of clauses defined as follows. Subsequently, w, x, y and z shall denote four pairwise distinct variables.

1. First, for every write instruction $\langle c, a, a', c'\rangle$, $\theta(\mathcal{M}, I)$ contains the two-literal clause
$$A(x, a, y, c) \lor \neg A(x, a', y, c').$$
2. For every left instruction $\langle c, a, l, c'\rangle$, $\theta(\mathcal{M}, I)$ contains the two two-literal clauses
$$A([\,], a, y, c,) \lor \neg A([\,], b, [a \mid y], c') \text{ and } A([x \mid w], a, y, c) \lor \neg A(w, x, [a \mid y], c').$$
3. For every right instruction $\langle c, a, r, c'\rangle$, $\theta(\mathcal{M}, I)$ contains the two two-literal clauses
$$A(x, a, [\,], c) \lor \neg A([a \mid x], b, [\,], c') \text{ and } A(x, a, [y \mid w], c) \lor \neg A(a \mid x], y, w, c').$$
4. Furthermore, $\theta(\mathcal{M}, I)$ contains the unit clause $A(x, w, y, q_a)$.
5. Finally, for the input tape $I = \langle t, z\rangle$, $\theta(\mathcal{M}, I)$ contains the negative unit clause $\neg A(l, t_z, r, q_0])$ where l and r are the list term encodings of the tape to the left respectively to the right of the head position z.

No other clause is contained in $\theta(\mathcal{M}, I)$.

Note that we need two clauses for every right or left instruction in order to simulate the infinite tape in a finitistic manner. It is straightforward to recognize that a nondeterministic Turing machine $\mathcal{M}$ accepts an input I if and only if the set of clauses $\theta(\mathcal{M}, I)$ is unsatisfiable. In more detail, the following property holds.

Proposition 8.11 A nondeterministic Turing machine $\mathcal{M}$ accepts an input I in n steps if and only if there is a closed regular connection tableau T for $\theta(\mathcal{M}, I)$ of depth $\leq n+1$ and constructed with $\leq n+1$ inference steps.

Proof For the "only-if" direction, we may restrict ourselves to an accepting Turing derivation D of minimal length $m \leq n$. Then, when starting with the top clause $\neg R(l, t_z, r, q_0)$, which corresponds to the input, one can step by step simulate the Turing derivation D by an extension step using exactly one of the clauses corresponding to the instruction used in the respective transition step. From the minimality assumption for D it follows that no configuration is repeated in D. This means that regularity of the tableau is preserved. Furthermore, in any extension step the tableau depth is increased by 1. For the "if" direction, it suffices to note that only inference steps are possible which correspond to transitions in the Turing machine. □

Proposition 8.12 The verification of the existence of a closed regular connection tableau with $\leq n$ inferences for a set of clauses S is NP-complete, where the input size is the size of S plus n.

Proof By Proposition 8.11, we can reduce the verification of whether any nondeterministic Turing machine $\mathcal{M}$ accepts an input I in $n-1$ steps to the verification of whether $\theta(\mathcal{M}, I)$ has a closed regular connection tableau constructed with $\leq n$ inferences, and the input sizes are linearly related. This proves the NP-hardness.

For proving the containment of the problem in NP, we nondeterministically guess a closed regular connection tableau T constructed with $\leq n$ inference steps and the respective pairs of complementary literals. Since T is connected, it has $\leq 2n-1$ nodes and $< n$ branches. When using dags, the size of T is $< 2ns$ where s is the maximal size of a literal in the input set. Finally, the complementarity of the respective literals can be checked in quadratic time in total. □

Note that the complexity of the corresponding problem for tableaux with local unification is open. This is because in n steps one can construct a tableau of a size exponential in n. This also holds for binary resolution. In both cases the critical mechanism is the renaming of variables (see [Letz, 1993a, Letz, 1993b]).

Next, we come to the tableau depth bound. The gist of the encoding employed here is that we use a binary predicate symbol R in order to express the fact that from a certain configuration C a configuration C' can be reached. Now, all instructions are encoded as unit clauses and we have only one clause of length three, which expresses the transitivity of the reachability relation.

Definition 8.13 (Reachability Turing encoding) Given a nondeterministic Turing machine $\mathcal{M} = \langle K, \Sigma, q_0, q_a, b, r, l, T\rangle$ and an input $I = \langle t, z\rangle$ to $\mathcal{M}$, the *reachability Turing encoding* $\theta'(\mathcal{M}, I)$ is the set of clauses defined as follows. Again, w, x, y and z shall denote four pairwise distinct variables.

1. First, for every write instruction $\langle c, a, a', c' \rangle$, $\theta'(\mathcal{M}, I)$ contains the unit clause
$$R([x, a, y, c], [x, a', y, c']).$$

2. For every left instruction $\langle c, a, l, c' \rangle$, $\theta'(\mathcal{M}, I)$ contains the two unit clauses
$$R([[\ ], a, y, c, z], [[\ ], b, [a \mid y], c']) \text{ and } R([[x \mid w], a, y, c], [w, x, [a \mid y], c']).$$

3. For every right instruction $\langle c, a, r, c' \rangle$, $\theta'(\mathcal{M}, I)$ contains the two unit clauses
$$R([x, a, [\ ], c], [[a \mid x], b, [\ ], c']) \text{ and } R([x, a, [y \mid w], c], [a \mid x], y, w, c']).$$

4. Then $\theta'(\mathcal{M}, I)$ contains the unit clause $R([x, w, y, q_a], [x, w, y, q_a])$.

5. For the input tape $I = \langle t, z \rangle$, $\theta'(\mathcal{M}, I)$ contains the negative unit clause $\neg R([l, t_z, r, q_0], [x, y, w, c_a])$ where l and r are the list term encodings of the tape to the left respectively to the right of the head position z.

6. Finally, $\theta'(\mathcal{M}, I)$ contains the transitivity clause for R.
$$R(x, z) \vee \neg R(x, y) \vee \neg R(y, z).$$

No other clause is contained in $\theta'(\mathcal{M}, I)$.

Proposition 8.14 A nondeterministic Turing machine $\mathcal{M}$ accepts an input I in n steps if and only if there is a closed regular connection tableau T for $\theta'(\mathcal{M}, I)$ of depth $< 3 + log_2 n$.

Proof Let $C_1, \ldots, C_n$ be any accepting derivation of minimal length. First, we structure the derivation hierarchically in the form of a binary tree, as follows. Let i be the smallest integer $> n/2$. If $i < n$, we split the derivation into two subderivations $C_1, \ldots, C_i$ and $C_i, \ldots, C_n$. This process is repeated recursively for the subderivations until no more splittings are possible, i.e., until the length of the subderivation is 2. The depth of the resulting tree is $< 1 + log_2 n$. This hierarchical organization of the derivation can be simulated by the reachability Turing encoding, as shown in Figure 8.1 for a derivation of length 5. Let C' be the term list encoding of any Turing configuration. We start with the all-negative clause. To every splitted (sub)derivation $C_j, \ldots, C_m$ with split configuration C_k, there corresponds a subgoal with literal $\neg R(C'_j, C'_m)$. Then we perform an extension step at this subgoal using the transitivity clause, which produces two new subgoals with literals $\neg R(C'_j, C'_k)$ and $\neg R(C'_k, C'_m)$, and so forth. For subgoals corresponding to subderivations of length 2, we simply extend into a unit clause. The depth of the resulting closed tableau $\theta'(\mathcal{M}, I)$ is $< 3 + log_2 n$. The additional 2 come from the start step and the extensions into unit clauses. Regularity is guaranteed by the minimality of the input derivation. $\square$

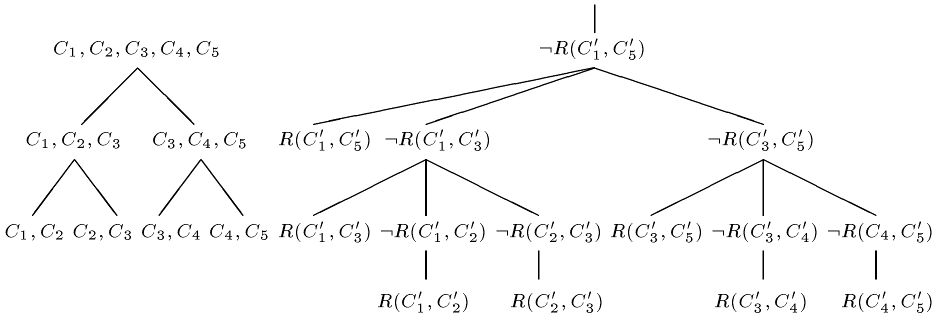


Figure 8.1: Tree-structured simulation of a Turing derivation.

Proposition 8.15 The verification of the existence of a closed regular connection tableau with depth $\leq n$ for a set of clauses S is NEXPTIME-complete, where the input size is the size of S plus n.

Proof By Proposition 8.14, we can reduce the verification of whether any nondeterministic Turing machine $\mathcal{M}$ accepts an input I in $2^n - 3$ steps to the verification of whether $\theta(\mathcal{M}, I)$ has a closed regular connection tableau of depth $\leq n$, and the input sizes are linearly related. This proves the NEXPTIME-hardness.

For proving the containment of the problem in NEXPTIME, we nondeterministically guess a closed regular connection tableau T of depth $\leq n$ and the respective pairs of complementary literals. T has $< l^n$ nodes. When using dags, the size of T is $< 1 + l^n(1 + s)$ where s is the maximal size of a literal in the input set. Finally, the complementarity of the respective literals can be checked in exponential time in total. $\square$

Finally, we come to the multiplicity bound. The existence of a closed tableau with a certain multiplicity is related with the following problem first encountered by Prawitz. He was interested in improvements of the well-known early approaches in automated deduction like the Davis/Putnam procedures [Davis and Putnam, 1960, Davis et al., 1962]. In those procedures the unsatisfiability of a first-order Skolem formula Φ is demonstrated by systematically building increasing sets of ground instances of the quantifier-free part—the *matrix*—of Φ, which are then checked by propositional decision procedures. A crucial point which determines the efficiency of this approach concerns the manner how the respective ground instances are generated. In the original procedures, no information of the connection structure of the matrix of Φ was used, but a systematic substitution of terms from the Herbrand universe of Φ was performed, which is obviously unmanageable in practice. The clause linking method [Lee and Plaisted, 1992] described in Section 4.2.3 presents one significant improvement of the selection of ground instances.

In contrast, Prawitz took another approach [Prawitz, 1960, Prawitz, 1969]. In order to improve the overall procedure, he proposed not to build ground instances of the formula but to work directly on variable-renamed copies of the matrix of Φ

and to determine an unsatisfiable set of ground instances by using unification on the connections in the set of copies. While the recognition of the unsatisfiability of a ground formula is a coNP-complete problem, the complexity of recognizing the existence of an unsatisfiable ground instance of a quantifier-free formula was settled just recently independently in [Voronkov, 1998] and [Letz, 1998a]. Voronkov's paper contains a number of results on the decidability and the worst-case complexities of multiplicity-based formula instantiation problems including different forms of equational reasoning.

Here, we consider the pure clausal problem. The clausal ground instantiation problem is the following problem: given a set of clauses S, find a ground substitution σ for S, i.e., a substitution that maps all variables in S to ground terms, such that $S\sigma$ is unsatisfiable. Interestingly, this problem is closely related with the problem of finding a unifiable spanning mating for a set of clauses occurring in the matings approaches of Andrews [Andrews, 1981] and Bibel [Bibel, 1981] presented in Section 4.1.2. First, if there is a unifiable spanning mating for a set of clauses S with unifier σ, then every ground instance of $S\sigma$ is unsatisfiable, hence S has an unsatisfiable ground instance. In the other direction, when a set of clauses S has an unsatisfiable ground instance $S\sigma$, then there is a unifiable spanning mating for S with simultaneous unifier σ. Furthermore, the mating of any closed regular connection tableau without renaming for a set of clauses S is spanning for S; and when a set of clauses has an unsatisfiable ground instance $S\sigma$, then, by Theorem 5.7, $S\sigma$ has a closed regular connection tableau, hence, by Lemma 5.9, S has a closed regular connection tableau without renaming for S. So all three problems are equivalent.

We will prove now that these problems are complete for the complexity class Σ_2^p in the polynomial hierarchy [Garey and Johnson, 1979]. Σ_2^p is the set of all problems solvable (languages recognizable) in nondeterministic polynomial time with an oracle to a problem in Σ_1^p = NP. The containment of the problems in Σ_2^p can be seen easily by using the problem of finding a unifiable spanning mating.

Proposition 8.16 The problem of finding a unifiable spanning mating for a set of clauses S is contained in Σ_2^p.

Proof First, guess a spanning mating M in S with simultaneous unifier σ. The cardinality of M is quadratically bounded by the number of literals in S. Furthermore, since there are linear unification algorithms [Paterson and Wegman, 1978], the simultaneous unifiability of M by σ can be verified (even decided) in linear time wrt. the size of M. Finally, since the verification of the spanning property is in coNP, the spanning property of M can be decided in polynomial time by an NP-oracle. □

For proving the completeness of the problem for the complexity class Σ_2^p, we use the well-known fact that language classes in PSPACE can be characterized with *quantified boolean formulae* [Garey and Johnson, 1979]. A quantified boolean formulae (QBF) is a boolean formula prefixed by a sequence of quantifications over the boolean variables with the interpretation domain of the variables being the set of the truth values $\{\top, \bot\}$. For example, the quantified boolean formula $\forall p \exists q \exists r (p \rightarrow (q \wedge r))$ is true, since, for every assignment of a truth value to p, there is an assignment to q and r that makes the formula $p \rightarrow (q \wedge r)$ true.

We will prove that there is a polynomial reduction of every quantified boolean formula B of the $\forall^*\exists^*$-type to a set of clauses S such that B is false if and only if S has an unsatisfiable ground instance. It suffices to consider the special case of QBFs whose matrices are in clausal form and in which no clause is *tautological*, i.e., does not contains an atom and its negation, since this restricted language is also complete for Σ_2^p . For any such QBF B, we define a corresponding set of clauses S.

Definition 8.17 (Clause set corresponding to a QBF) Let B be any QBF of the mentioned type, i.e., B is of the form $\forall p_1 \cdots p_m \exists q_1 \cdots q_n C$ with $C = c_1 \wedge \cdots \wedge c_k$, and each c_i, $1 \leq i \leq k$ being a non-tautological clause. Let $Q_1, \ldots, Q_n$ be n first-order predicate symbols with arity m each, $x_1, \ldots, x_m$ first-order variables, and 0 and 1 two constants. Now, for each clause c_i, $1 \leq i \leq k$, a *corresponding* first-order clause c'_i is defined as follows. The clause c'_i contains the literal $(\neg)Q_i(\alpha_1, \ldots, \alpha_m)$ if and only if q_i occurs positively (negatively) in c_i; furthermore, all literals in c'_i have the same arguments

$$\alpha_i = \begin{cases} 1 & \text{if } p_i \text{ occurs positively in } c_i \\ 0 & \text{if } p_i \text{ occurs negatively in } c_i \\ x_i & \text{otherwise.} \end{cases}$$

The clause set S *corresponding to* B is the set of all clauses which correspond to clauses in C.

Example 8.18 In order to facilitate the understanding of the transformation, we give a concrete example of a QBF and its transform. Let B be the formula $\forall p_1 p_2 \exists q_1 q_2 C$ with the clauses in the conjunction C given on the left-hand side. The clauses resulting from the transformation are given on the right-hand side.

$p_1 \vee q_1 \vee q_2$	$Q_1(1, x_2) \vee Q_2(1, x_2)$
$\neg p_1 \vee q_1 \vee q_2$	$Q_1(0, x_2) \vee Q_2(0, x_2)$
$p_2 \vee \neg q_1$	$\neg Q_1(x_1, 1)$
$\neg p_2 \vee \neg q_2$	$\neg Q_2(x_1, 0)$
$\neg p_1 \vee \neg p_2 \vee \neg q_1 \vee q_2$	$\neg Q_1(0,0) \vee Q_2(0,0)$

So the q-literals in a clause of the QBF are captured by respective predicate symbols whereas the p-literals in a clause are uniformly encoded in the arguments of the corresponding clause.

Proposition 8.19 For any quantified QBF B of the aforementioned type, B is false if and only if the set of clauses S corresponding to B has an unsatisfiable ground instance.

Proof For the "if" direction, consider any minimal subset $S' \subseteq S$ such that $S'\sigma$ is unsatisfiable for some ground substitution σ. First, we show that all literals in $S'\sigma$ have the same argument list $\alpha_1, \ldots, \alpha_n$. From the structure of S it is obvious that all literals in the same *clause* in $S'\sigma$ must have the same argument list. Consider any closed connection tableau for $S'\sigma$. Then, after the start step, only clauses can be attached by extension steps with literals of the same argument list. Therefore, by the minimal unsatisfiability of $S'\sigma$, all literals in $S'\sigma$ must have the same argument list. Now we show that B is false. For this consider an arbitrary Boolean valuation v satisfying that, for all $1 \leq i \leq m$, $v(p_i) = \top$ if and only if $\alpha_i = 0$. Consider the interpretation I defined by $I(Q_i(\alpha_1, \ldots, \alpha_n)) = v(q_i)$. By the unsatisfiability of $S'\sigma$, there is a clause $c' \in S'$ such that $I(c'\sigma) = \bot$, i.e., for every literal L contained in $c'\sigma : I(L) = \bot$. Let c be the clause in the matrix of B which was transformed to c'. Then, by the transformation, for every literal L in c with an atom q_i, $I(L) = \bot$. Furthermore, for every literal K in c of the form p_i respectively $\neg p_i$, $v(K) = \bot$, since, by the definition of v, $v(p_i) = \bot$ respectively $\top$ if p_i occurs positively respectively negatively in c. This proves that B is false.

For the "only-if" direction, assume that B is false. This means that there exists a partial evaluation v' with domain $\{p_1, \ldots, p_m\}$ such that B is false for all total extensions of v'. Let σ be the substitution defined by $\sigma(x_i) = 0$ or 1 depending on whether $v'(p_i) = \top$ or $\bot$. We show that $S\sigma$ is unsatisfiable. For this consider an arbitrary interpretation I for $S\sigma$. Define the Boolean valuation v by setting $v(p_i) = v'(p_i)$, for $1 \leq i \leq m$, and $v(q_i) = I(Q_i(\alpha_1, \ldots, \alpha_n))$, for $1 \leq i \leq n$, where $\alpha_i = 0$ or 1 depending on whether $v'(p_i) = \top$ or $\bot$. By assumption, there exists a clause c in the matrix of B such that $v(L) = \bot$ for all literals L in c. Let c' be the clause to which c was transformed. Then, by construction, $I(c'\sigma) = \bot$. So $S\sigma$ is unsatisfiable. □

Proposition 8.20 The verification of the existence of a closed regular connection tableau with multiplicity $\leq n$ for a set of clauses S is Σ_2^p-complete, where the input size is the size of S plus n.

Proof Since the used transformation is quadratic in the size of the QBF, the result immediately follows from Proposition 8.16 (containment in Σ_2^p) and Proposition 8.19 (Σ_2^p-hardness). □

Interestingly, the transformation maps any QBF of the specified type to a set of datalogic clauses, i.e., clauses in which no function symbols of arity greater the 0 occur. Consequently, the considered verification problem is already Σ_2^p-complete for this restricted formula class.

We can observe that there is a striking difference between the containment of the problem in the complexity class Σ_2^p and the doubly exponential upper bound estimate for the number of tableaux with a certain multiplicity. Note that the membership of a verification problem in Σ_2^p means that the problem can be *decided* in single exponential time. So this complexity result shows that the standard iterative-deepening approach seems not suited when used with the multiplicity bound. A naïve algorithm which would enumerate all unifiable matings of the clause set and test whether they are spanning needs only single exponential time, even if no refinements are used.

8.2 The Reductive Power of Refinements

In this section, we will consider the effect of the most important refinements of clausal tableau calculi on the search spaces of certain input formulae. The motivation for any form of search pruning is that the search space decreases. For most of the introduced pruning methods, it is relatively straightforward to demonstrate that there is a class of formulae with infinite search spaces with and finite search spaces without the pruning method. For instance, the set of clauses given in Example 8.6 has a finite search space with and an infinite one without regularity. Similar examples exist for the connection conditions, tautology and tableau clause subsumption, for the use of relevance information, and for (local) failure caching. In Section 5.3.1, it was shown that the matings pruning technique may also have a strong search pruning effect. Moreover, local failure caching is a very powerful method, which cannot be completely captured by structural pruning paradigms. The caching technique proposed in [Astrachan and Stickel, 1992], which permanently stores the solutions of subgoals and uses cached solutions for solving subgoals by lookup instead of search, provides a polynomial decision procedure for propositional Horn sets [Plaisted, 1994, Plaisted and Zhu, 1997]. An interesting other topic, which we will discuss in more detail, is the influence of subgoal selection functions.

8.2.1 Free Subgoal Selection Functions

As discussed in Section 3.4, the used method of subgoal selection may have an effect on the search space. In particular, it is important to emphasize that the restriction to depth-first subgoal selection functions (which is enforced in the chain format of model elimination) can have a very detrimental effect. In order to illustrate this, let us reconsider the set of clauses

$$
\begin{array}{ll}
\neg P(x) \vee \neg Q(x), & \\
P(a), & Q(b), \\
P(f(x,y)) \vee \neg P(x) \vee \neg P(y), & Q(g(x,y)) \vee \neg Q(x) \vee \neg Q(y)
\end{array}
$$

given in Example 8.5. We have shown that the search space of this clause set increases exponentially for the inference bound and double exponentially for the depth bound, for any depth-first selection function. Interestingly, when permitting arbitrary subgoal selection functions, one can improve this behaviour tremendously. Consider any subgoal selection function which always selects a subgoal with a literal of maximal term size. Assume w.r.g. that we select first the $\neg P(x)$-subgoal. The crucial difference of such a selection function from selection functions of the pure depth-first type occurs when the clause

$$P(f(x,y)) \vee \neg P(x) \vee \neg P(y)$$

is entered for the first time. (This must happen after at most four inference attempts, since when the unit clause $P(a)$ is tried first, the failure of the other top subgoal is detected in two further inference attempts.) After the extension step into the renamed clause

$$P(f(x', y')) \vee \neg P(x') \vee \neg P(y')$$

the variable x is instantiated to $f(x', y')$ while the variables x', y' in the new subgoals are not instantiated. When preferring subgoals with literals of maximal term size, in the next inference step the other top subgoal with literal $\neg Q(f(x', y'))$ is selected. The impossibility of solving the subgoal is detected within two further inference attempts. Since no further inferences are possible, the search procedure stops. Consequently, we achieve a reduction of the search space from an exponential respectively doubly exponential size to a constant size of 3, which is even independent of the limit n of the used completeness bound. This is a striking illustration of the power of subgoal selection strategies. The example also exhibits the severe limitations of the chain-oriented model elimination format, which only permits depth-first subgoal selection and hence cannot achieve such a reduction.

Chapter 9

Implementation of Connection Tableaux

All competitive implementations of connection tableaux are iterative-deepening search procedures using backtracking. When one envisages the implementation of such a procedure, one has the choice between fundamentally different architectures, for the following reason. As sketched at the end of Section 4.1, it is straightforward to recognize that SLD-resolution (the inference system underlying Prolog) can be viewed as a refinement of connection tableaux obtained by simply omitting the reduction inference rule. Since highly efficient implementation techniques for Prolog have been developed, one can profit from these efforts and design a *Prolog Technology Theorem Prover (PTTP)*. The crucial characteristics of Prolog technology is that input clauses are *compiled* into procedures of a virtual or concrete machine which permits a very efficient execution of the extension operation. There are even two different approaches of exploiting Prolog technology. On the one hand, one can build on some of the efficient implementation techniques of Prolog and add the ingredients needed for a sound and complete connection tableau proof procedure. On the other hand, one can use Prolog itself as implementation language with the hope that its proximity to connection tableaux permits a short and efficient implementation of a connection tableau proof search procedure. The PTTP approaches, which will be both described in this section, have dominated the implementations of connection tableaux in the last years. The use of Prolog technology, however, has a severe disadvantage, namely, that the framework is not flexible enough for an easy integration of new techniques and new inference rules. This inflexibility has almost blocked the implementations of certain important extensions of connection tableaux, in particular, the integration of inference mechanisms for an efficient equality handling. Therefore, we also present a more natural and modular implementation architecture for connection tableaux, which is better suited for various extensions of the calculus. Although this approach cannot compete with the PTTP approaches concerning the efficiency by which new instances of input clauses are generated, this drawback can

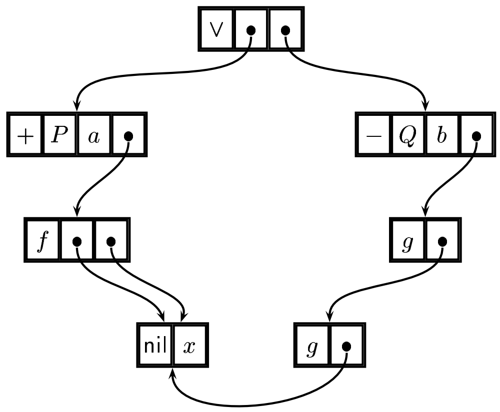


Figure 9.1: Internal representation of the clause $P(a, f(x,x)) \leftarrow Q(b, g(g(x)))$.

be compensated for by an intelligent mechanism of reusing clause copies, so that, for the typical problems occurring in automated deduction, about the same high rates of inferences per seconds can be achieved.

9.1 Basic Data Structures and Operations

When analyzing the actual implementations of connection tableaux, one can identify some data structures and operations that are more or less common to all successful approaches and hence form something like a standard basis. The first subject is the way formulae should be represented internally in order to permit efficient operations on them. It has turned out that all terms, literals, and clauses may be represented in a natural tree manner except variables, which should be shared. In Figure 9.1, such a standard representation of a clause is depicted. The treatment of variables needs some further explanation. Internally, variables are typically represented as structures consisting of their actual bindings and their print names with nil indicating that the variable is currently free. Furthermore, variables are not identified and distinguished by their print names but by the addresses of their structures.

9.1.1 Unification

The next basic ingredient is the employed unification algorithm, which is specified generically in Table 9.1. In the displayed procedures, it is left open how variable bindings are performed and retracted. Unification is specified with two mutually recursive procedures, the first one for the unification of two lists of terms, the other for the unification of two terms. The standard in connection tableau implementations is that a binding is performed destructively by deleting nil from the variable cell and inserting a pointer to the term to be substituted for the

```
procedure unify_lists( args1,args2 )
  if ( args1 = ∅ ) then
    true;
  /* check first arguments */
  elseif ( unify( binding( first( args1 ) ),binding( first( args2 ) ) ) then
    unify_lists( rest( args1 ),rest( args2 ) );
  /* undo variable bindings made in this procedure */
  else
    unbind;
    false;
  endif;


procedure unify( arg1,arg2 )
  if ( is_var( arg2 ) ) then
    if ( occurs( arg2,arg1 ) ) then
      false;
    else
      bind( arg2,arg1 );
      true;
    endif;
  elseif ( is_var( arg1 ) ) then
    if ( occurs( arg1,arg2 ) ) then
      false;
    else
      bind( arg1,arg2 );
      true;
    endif;
  elseif ( functor( arg1 ) == functor( arg2 ) ) then
    unify_lists( args( arg1 ),args( arg2 ) );
  else
    false;
  endif;
```

Table 9.1: The unification procedures.

variable. The resulting bound variable cell then does no more denote a variable but the respective term. The recursive function binding(term) returns term if term is not a bound variable cell, or otherwise binding(first(term)). On backtracking, variable bindings have to be retracted. This is done by simply reinserting nil in the first element of the respective bound variable cells, so that the original state is restored. Note that the unbind procedure is assumed to retract all variable bindings performed in the current unification attempt.

Polynomial unification

It is straightforward to recognize that this unification procedure is linear in space but exponential in time in the worst case. Although this is not a critical weakness for the typical formulae in automated deduction, one may easily improve the given procedure to a polynomial time complexity by using methods described on Page 49. The key idea of such methods is that one attaches an additional tag at any complex term. This tag is employed to avoid that the same pairs of complex terms are successfully unified more than once during a unification operation. Furthermore, this tag can be used to reduce the number of occurs-checks to a polynomial (see [Corbin and Bidoit, 1983, Letz, 1993a]).

Destructive unification using the trail

In order to know which bound variables have to be unbound, the *trail* is used as a typical data structure. The trail is a global list-like structure in the program which contains the pointers to the bound variables in the order in which they have been bound. Since all standard backtracking procedures retract bindings exactly in the reversed order of their generation, a simple one-dimensional list-like structure is sufficient for the trail. The trailmarker is a global variable which gives the current position of the trail. The number of bindings performed may differ from one inference step to another. In order to know how many bindings have to be retracted when an inference step is retracted, there are two techniques. One possibility is to locally store the number of performed bindings or the trail position at which the bindings of the previous inference step start. Alternatively, one can use a special stop label on the trail which is written in a trail cell whenever an inference step ends; in this case, no local information is needed. Depending on which solution is selected, the unification procedure has to be modified respectively.

Figure 9.2 documents the entire binding process and the trail modifications performed during proof search for the set of the four clauses $\neg P(x,y) \vee \neg P(y,x)$, $P(a,z)$, $P(b,v)$, and $Q(a,b)$. The description begins after the start step in which the first input clause has been attached (a). First an extension step using the second input clause is performed, which produces two bindings (b). Then an extension step with the Q-subgoal is attempted: y (and implicitly z) are bound to a, but the unification fails when a (the binding of x) is compared with b (c). Then the two inferences are retracted (d). After extension steps using the third clause (e) and the fourth clause (f), the proof attempt succeeds. This technique permits that backtracking can be done very efficiently.

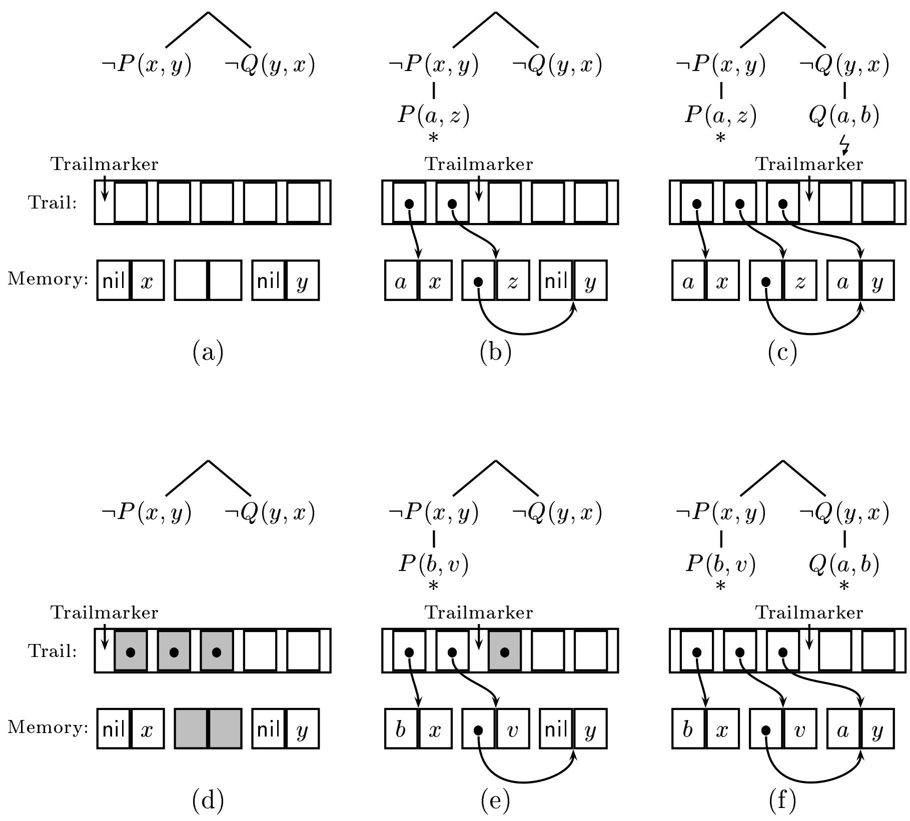


Figure 9.2: An example of the trail modifications during proof search.

9.1.2 The Connection Graph

In order to permit an efficient performance of extension steps, it is necesssary that the connected literals of the current subgoal can be accessed quickly. The set of connections between the literals of a clause set can be represented in an undirected graph, the so-called *connection graph*. When a subgoal is selected for an expansion step during the proof search, it is sufficient to consider the connections involving that subgoal.

Example 9.1 To demonstrate the concept of the connection graph, we consider the clauses $\neg P(g(x)) \vee \neg Q(g(x)) \vee \neg Q(f(x))$, $Q(x) \vee \neg P(f(x))$, $Q(g(y)) \vee P(f(y))$ and $P(g(a))$. The connection graph of this set of clauses is depicted in Figure 9.3. The connections are indicated by the solid lines between the literals. The faint dashed lines show the pairs of literals where, even though they have the same predicate symbol and complementary signs, the argument terms cannot be unified. These literals are not connected. Thus, when the literal $\neg P(g(x))$ has been chosen for an extension step, the clause $Q(g(y)) \vee P(f(y))$ need not be tried.

Since the variables in clauses are all implicitly universally quantified, the connections between literals are independent of any instantiations applied during the proof search. Therefore, the computation of the literal connections can be done statically and used as a filter. If there is a connection between two literals P and Q, then P is also said to have a *link* to Q and vice versa. The links of a literal are stored in its *link list*. The link lists for the literals in Example 9.1 are:

$$\begin{array}{rl}
P(g(a)): & [\neg P(g(x))] \\
\neg P(g(x)): & [P(g(a))] \\
\neg Q(g(x)): & [Q(g(y)), Q(x)] \\
\neg Q(f(x)): & [Q(x)] \\
Q(x): & [\neg Q(g(x)), \neg Q(f(x))] \\
\neg P(f(x)): & [P(f(y))] \\
P(f(y)): & [\neg P(f(x))] \\
Q(g(y)): & [\neg Q(g(x))]
\end{array}$$

The problem of generating clause variants

One of the main difficulties when implementation connection tableaux is how to provide renamed variants of input clauses efficiently, because in every extension step a new variant of an input clause is needed. It is obvious that the generation of a new variant of an input clause by copying the clause and replacing its variables consistently with new ones is an expensive operation, all the more since variables are shared and it is not a tree that has to be copied but a graph. The search for an efficient solution of this problem naturally leads to the use of Prolog technology.

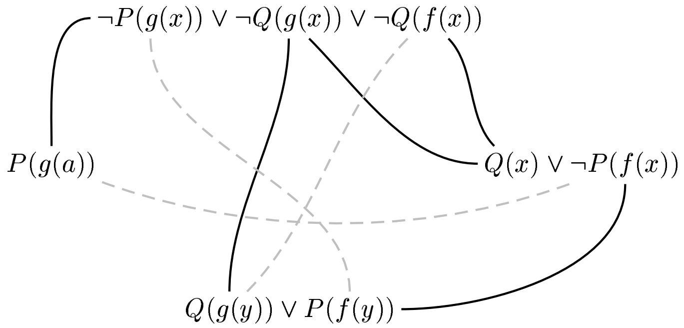


Figure 9.3: The connection graph for the set of clauses $\neg P(g(x)) \vee \neg Q(g(x)) \vee \neg Q(f(x))$, $Q(x) \vee \neg P(f(x))$, $Q(g(y)) \vee P(f(y))$ and $P(g(a))$.

9.2 Prolog Technology Theorem Proving

One reason for the high efficiency of current Prolog systems is the fact that many of the operations to be performed in SLD-resolution steps can be determined in advance depending on the respective clause and its entry literal. This information can be used for compiling every Prolog input clause A :- $A_1,\ldots,A_n$ (which corresponds to the clause $A \vee \sim A_1 \vee \cdots \vee \sim A_n$ with entry literal A) into procedures of some actual or virtual machine. Since SLD-resolution steps are nothing but extension steps, this technique can also be applied to connection tableaux. The first to use such a compilation method for connection tableaux was Mark Stickel [Stickel, 1984] who called his system a PTTP, a Prolog Technology Theorem Prover.

In summary, the main deficiencies of Prolog as far as first-order automated reasoning is concerned are the following:

1. the incompleteness of SLD-resolution for non-Horn formulae,
2. the unsound unification algorithm, and
3. the unbounded depth-first search strategy.

To extend the reasoning capabilities of Prolog to full connection tableaux, it is necessary to extend SLD-resolution to the full extension rule and to add the start rule and the reduction rule.

Contrapositives

In order to implement the full extension rule and further permit the compilation of input clauses into efficient machine procedures, one has to account for the fact that a clause may be entered at every literal. Accordingly, one has to consider all so-called *contrapositives* of a clause $L_1 \vee \cdots \vee L_n$, i.e., the n Prolog-style strings of the form L_i :- $\sim L_1,\ldots,\sim L_{i-1},\sim L_{i+1},\ldots,\sim L_n$. The start rule can also be captured efficiently, by adding, for every input clause $L_1 \vee \cdots \vee L_n$, a contrapositive of the form $\perp$:- $\sim L_1,\ldots,\sim L_n$. Now, with the Prolog query ?- $\perp$ as the single start clause, all start steps can be simulated with extension steps. As a matter of fact, one can use relevance information here and construct such start contrapositives only for those subset of the input formulae which are known to contain a relevant start clause; by default, this will be the set of all-negative input clauses.

Unification in Prolog

Prolog by default uses a unification algorithm that is designed for maximum efficiency but that can lead to incorrect results. A Prolog program like

```
X < ( X + 1 ).
:- ( Y + 1 ) < Y.
```

can prove that there is a number whose successor is less than itself; the reason for the unsoundness is that no occurs-check is performed in Prolog unification. Since the compilation of extension steps into machine procedures also concerns parts of the unification, this compilation process has to be adapted such that sound unification is performed. In special cases, however, efficiency can be preserved, for example, if the respective entry literal L is *linear*, i.e., if every variable occurs only once in L. It is straightforward to recognize that in this case, no occurs-check is needed in extension steps and the highly efficient Prolog unification can be used. For the general case, an optimal method exploits this optimization by distinguishing the first occurrence of a variable in a literal from all subsequent ones. For every first occurrence, the occurs-check may be omitted. In Table 9.2, a procedure is shown which performs an extension step including the generation of a new clause variant in a very efficient manner.

Path information and other extensions

Unfortunately, for the implementation of the reduction inference rule, one definitely has to provide additional data structures. While in SLD-resolution the ancestor literals of a subgoal are not needed, for connection tableaux, the tableau paths have to be stored and every subgoal must have access to its path. This additional effort cannot be avoided. On the other hand, access to the ancestors of a subgoal is necessary for the implementation of basic refinements like regularity, which is also very effective in the pure Horn case.

Finally, the unbounded depth-first search strategy of Prolog has to be extended to incorporate completeness bounds like the inference bound, the depth bound or other bounds discussed in Section 3.3.1. Depending on the used bounds, one has to use different data structures. In order to capture bounds which allocate remaining resources directly to subgoals like the depth bound, every subgoal has to be additionally labelled with its current depth. For the inference bound one needs a global counter. Interestingly, the multiplicity bound is not at all compatible with standard Prolog technology. This is because Prolog has no natural mechanism for instantiating input clause.

Contrapositive: $P(a, f(x, x))$:- $Q(b, g(g(x)))$

```
procedure P( arg1,arg2 )
  variable x,tq,arg21,arg22,trail_position;
  x := new_free_variable;
  arg1 = binding( arg1 );
  /* mark trail position */
  trail_position := trailmarker;
  /* unify clause head: check first arguments */
  if ( is_var( arg1 ) or ( is_const( arg1 ) and arg1 == a ) ) then
    if ( is_var( arg1 ) ) then bind( arg1,a );
    endif;
    /* first arguments unifiable, check second arguments */
    arg2 = binding( arg2 );
    if ( is_var( arg2 ) ) then
      bind( arg2,make_complex_term( f,x,x ) );
      tq := make_complex_term( g, make_complex_term( g,x ) );
      add_subgoal( Q(b,tq) );
      next_subgoal;
    elseif ( is_complex_term( arg2 ) and functor( arg2 ) == f ) then
      arg21 := binding( get_arg( arg2,1 ) );
      arg22 := binding( get_arg( arg2,2 ) );
      if ( unify( arg21,arg22 ) ) then
        tq := make_complex_term( g, make_complex_term( g,arg21 ) );
        add_subgoal( Q(b,tq) );
        next_subgoal;
      endif;
    endif;
  endif;
  /* undo variable bindings made in this procedure */
  unbind( trail_position );
```

Table 9.2: Compilation of a contrapositive into a procedure.

9.3 Extended Warren Machine Technology

The main problem of a two-step compilation, first to some real programming language and then to native code, is that the second compilation process takes too much time for typical applications, which require a quick response time. In order to avoid the second compilation phase, an interpreter for the code generated in the first compilation phase has to be used. Since not the full expressive power of an actual programming language is needed, this has motivated the development of a very restricted abstract language tailored specifically for the processing of Prolog respectively connection tableaux. We begin with outlining the basics of such a machine for Prolog (see [Warren, 1983] and [Schumann, 1991] for a more detailed description).

9.3.1 The Warren Abstract Machine

D.H.D. Warren developed a virtual machine for the execution of Prolog programs [Warren, 1983] which is called the Warren Abstract Machine (WAM). It combines high efficiency, good portability, and the possibility for compiling Prolog programs. The WAM is used widely and has become a kernel for commercial Prolog systems implemented as software emulation or even micro-coded [Taki et al., 1984, Benker et al., 1989] on dedicated hardware. The WAM is structured as a register-based multi-memory machine as shown in Figure 9.4. Its *memory* holds the program (as a sequence of WAM instructions) and data. The *register file* keeps a certain set of often used data and control information. The WAM instruction, which is located in the memory at the place where the *program counter* register points to, is fetched and executed by the *control unit*.

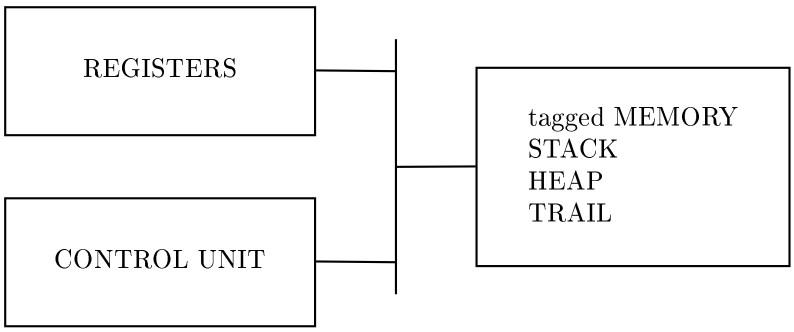


Figure 9.4: The Warren Abstract Machine.

We will describe now how a Prolog program is *compiled* into machine instructions of the WAM. We begin with the special case of a *deterministic* program which corresponds to a situation in which there is only one possibility for extending the current subgoal. In this case no backtracking inside the clause is needed. As already noted, the respective tableau is generated using a depth-first left-to-right selection function. Then the program can be executed in the same manner as in a *procedural* programming language, that is, the head of a clause is considered as the *head* of a procedure and the subgoals as the *procedure calls* of other procedures (the parameter passing, however, is quite different). Accordingly, this can be implemented on a machine level exactly in the way it is done in functional or procedural languages, using a *stack* with *environment control blocks* which hold the control information (return address, dynamic link) and the local variables. Details about this can be found e.g. in [Aho and Ullman, 1977]. The local variables are addressed using a register E pointing to the beginning of the current environment. A Horn clause `H :- G1,...,Gn.` is executed using the following instructions[1].

```
H:                          % entry point for clause H.
        allocate            % generate new environment (on stack) with space for locals
        ...                 % pass parameters (discussed below)
        ...                 % set parameters for G1 (discussed below)
        call         G1     % call first subgoal, remember return address A
A: ...
        ...                 % set parameters for Gn (discussed below)
        call         Gn     % call last subgoal, remember return address
        deallocate          % deallocate control block and return
```

Each environment contains a pointer to the previous environment (*dynamic link*). The entire list represents the *path* from the root to the current node in the tableau, the return addresses in the environments point to the code of the subgoals. The program terminates when the last call in the query returns.

The parameters of the head and the subgoals of the clauses are *terms* in a logical sense consisting of *constants*, *logical variables*, *lists*[2], and *structures* (complex terms). A Prolog term is represented by a *word* of the memory, containing a *value* and a *tag*. The tag distinguishes the type of the term, namely *reference*, *structure*, *list*, and *constant*. The tag type "reference" is used to represent the (logical) variables. Structures are represented in a non structure-sharing manner, i.e. they are copied explicitly with their functors.

For the purposes of parameter passing, the WAM uses two sets of registers, the registers $A_1, \ldots, A_n$ for keeping parameters and temporary registers $T_1, \ldots, T_n$. When a subgoal is to be called, its parameters are provided in the registers A_i by using `put` instructions. There exists one `put` instruction for each data type. In the head of a clause, the parameters in the A registers are fetched and matched against the respective parameter of the head, using a `get` or `unify` instruction. Here again, a separate instruction for each data type is provided. The matching algorithm has to check if constants and functors are equal. If a variable has to be bound to a constant, the value of the constant and the tag "constant" is written into the memory location where the variable resides; if the variable is bound to a structure, a pointer to that structure is written into the variable cell, together with the tag "reference". Structures itself are created in a separate chunk of the memory, the *heap*, which permits a permanent storage of those data.

[1] Actually, the WAM provides a number of different instructions for the sake of optimization, e.g., for tail recursion elimination. Here, only the basic instructions are described.

[2] A list is considered as a data type of its own for reasons of efficiency. A list could also be represented as a binary structure: $list(Head, Tail)$ similar to the Lisp function *cons*.

The following example illustrates the usage of the instructions for parameter passing. Let us assume that a subgoal $P(a, z)$ calls a head of a clause $P(a, [x|y])$:- ... The variables reside in the environment control block and are accessed via an offset from the register E pointing to the current environment. In [Warren, 1983] they are noted as $Y_1, \ldots, Y_n$.

```
    put_constant      a,A1    % put first parameter (constant a) into register A1
    put_variable      Y4,A2   % put variable z (in variable cell #4) into A2
    call              P       % call the "P-clause"

P:
    allocate          2       % allocate space for 2 variables
    get_const         a,A1    % try to unify 1st parameter with constant a
    get_list          A2      % get second argument: must be a list or variable
    unify_variable    Y1      % unify with variable z (in local cell #1)
    unify_variable    Y2      % unify with variable z (in local cell #2)
    ...                       % body of clause comes here
```

The last example also shows that the `get` and `unify` instructions must operate in two modes ("read","write") according to the type of parameter they receive. If the variable z in the subgoal has been bound to some list prior to this call, for example, to `[a|b]`, then the list is broken apart by the `get_list` instruction and x and y in the head are bound to `a` respectively to `b` (read mode). If, however, z in the subgoal has not yet been bound to a list, a *new* list, consisting of two variables is created on the heap by the instructions `get_list` and `unify_variable` (write mode). Note that the creation on the heap is necessary, since the newly created list has to stay in existence even after the execution of the clause `P`.

Finally, let us consider the full case of nondeterministic programs, in which a subgoal is connected to more than one clause head. Now backtracking is needed. Backtracking is implemented by means of so-called *choice points*, control blocks which hold all the information for undoing an inference step. These choice points are pushed onto the stack. The basic information of a choice point is a link to its predecessor, a code address to the entry point of the next clause to be attempted, and the information that is needed to undo all tried extension steps since that choice point was created. This involves a copy of all registers of the WAM as well as the variables which have been bound since the generation of the choice point. For the latter purpose a *trail* is used, in the same manner as described in Section 9.1. Whenever a backtracking action has to be performed, all registers from the current choice point are loaded into the WAM, all stack modifications are undone, and the respective variables are unbound. Then the next clause is attempted. The WAM contains a last alternative optimization, according to which the choice point can be discarded if the last extension clause is tried. The list of different possibilities is coded by the instructions `try_me_else`, and `trust_me_else_fail`, the latter representing the last alternative. Assuming that there be three clauses c1, c2, c3 for extension, the compiled code is shown below.

...

```
   call    P               % call the P-clauses
P:                         % generate a choice-point.
c123:
   try_me_else C2a         % try c1; if this fails, try c2
c1:
     ...                   % code of clause c1
c2a:
   try_me_else C3a         % try c2; if this fails, try c3
c2:
     ...                   % code of clause c2
c3a:
   trust_me_else_fail      % there is only one alternative left
c3:
     ...                   % code of clause c3
```

The WAM has some additional instructions for optimization which we will mention briefly. First, a *dynamic* preselection on the data type of the first parameter is done (`switch_on_term`). Its arguments give entry points of lists of clauses which have to be tried according to the type of the first parameter of the current subgoal (variable, constant, list, structure). Also hash tables are used for the selection of a clause head. This is useful when there is a large number of head literals with constants as first arguments.

9.3.2 The SETHEO Abstract Machine

Motivated by the architecture of the WAM, the connection tableau prover SETHEO [Letz et al., 1992] has been implemented. The central part of SETHEO is the SETHEO Abstract Machine (SAM), which is an extension of the WAM. The concepts introduced there had to be extended and enhanced for attaining a complete and sound proof procedure for the full connection tableau calculus, and for facilitating the use of advanced control structures and heuristics. The layout of the abstract machine is basically the same as in Figure 9.4, except that additional space is reserved for the *proof tree* and the *constraints*, which are discussed in Chapter 10. The *proof tree* stores the current state of the generated tableau, which can be displayed graphically to illustrate the structure of the proof. Additionally, there are global counters, e.g., for the number of inferences performed.

The reduction step

To successfully handle non-Horn clauses in connection tableaux, extension steps and *reduction steps* are necessary. A subgoal in the tableau can be closed by a reduction step if there exists a complementary unifiable literal in the path from the root to the current node. The resulting substitution σ is then applied to the entire tableau. How can such a step be implemented within the concepts of an abstract machine? As described above, the tableau is implicitly represented in the stack of the machine, using a linked list of *environment* control blocks. This linked list just represents the path from the root of the tableau to the current node[3]. Thus, the instruction executing the reduction step searches through this list, starting from the current node, to find a complementary literal which is unifiable with the current subgoal. The respective unification is carried out in the standard way. This procedure, however, requires that additional information must be stored in each environment, namely, the *predicate symbol* of the head literal of a contrapositive, its *sign*, and a pointer to the *parameters* of that literal. The detailed structure of an environment of the SAM is displayed in Figure 9.5, the *base pointer* points to the current environment in the *stack*.

base pointer →	dyn_link	link to previous environment
	ret_addr	return address
	ps_symb	coded predicate symbol and sign
	gpr	pointer to goal environment in the *code*
	variables	local variables for that clause
	...	

Figure 9.5: The SAM environment.

The reduction inference rule itself is nondeterministic in the sense that a subgoal may have more than one connected predecessor literal in the path. Hence, we have to store an additional *pointer* in every choice point, pointing to the environment which corresponds to the node which has to be tried in the next reduction step.

Efficiency considerations

To increase the efficiency of the SETHEO machine, a tagged memory is used. The basic types of variables, terms, constants and reference cells, which are used in the Warren abstract machine, are divided into further subtypes in order to gain a better performance (compare also [Vlahavas and Halatsis, 1987]). Thus, for instance, the type 'variable' has the subtypes: 'free variable' (T_FVAR), 'temporary variable' (T_TVAR), and 'bound variable', i.e. a reference cell (T_BVAR). Also complex terms are tagged differently depending on whether they contain variables or not. The additional information contained in these tags can be used for optimizing the unification operation.

Parameter transfer

In the original WAM, parameters from a subgoal to a head of a clause are transferred via the A_i registers. As a minimal number of registers and a variable number of parameters were required, this solution was not suitable. Instead,

[3]For this, no *tail recursion* optimization may be performed as it is done in the WAM. This optimization tries to delete environments as soon as possible, e.g., before executing the last subgoal of a clause.

the parameters are transferred via an argument vector. This originates from [Vlahavas and Halatsis, 1987], but it had to be adapted. The number of parameters of a subgoal and their types are fixed. The only exception are variables, which may be unbound or bound to an arbitrary term. Consequently, an argument vector is generated during compile time in the code area which contains the values and data types of the parameters. In case of variables an offset into the current environment is given. When dereferencing this address, the binding of the variable can be accessed. The only information directly passed during the execution of a `call` instruction is the address of the beginning of this argument vector. It is put into the register *gp* (goal pointer). After the selection of a clause head, the parameters of the subgoal are unified with the parameters of the head. For each parameter in the head, a separate unify instruction is used. It attempts to unify the respective parameter with the parameter *gp* points to. In case of success, gp is incremented. The following example shows the construction of the argument vector. Consider a subgoal $P(a, x, f(x))$. It will generate something like the following argument vector consisting of three words.

```
gp:      T_CONST    16    % 1st argument: constant a as index into symbol table
         T_VAR1     1     % 2nd: variable x with offset 1 (w.r.t.\ environment)
         T_CREF     term1 % 3rd: pointer to term f(x)
         ...
term1:   T_NGTERM   17    % functor f with index 17
         T_VAR2     1     % variable x (second occurrence)
         T_EOSTR    0     % end of the structure
```

9.4 Prolog as Implementation Language

The preceding parts have shown that it is a considerable effort to implement connection tableaux by extending Prolog technology. Since SLD-resolution is very similar to connection tableaux, many newer implementations of connection tableaux are done directly in Prolog. We will consider now the potential of using Prolog as an implementation language. It is straightforward to see that a basic implementation of connection tableaux can easily be obtained in Prolog. First, we need to provide all contrapositives. Second, the possibility of performing reduction steps has to be provided. Both can be done in a straightforward way, as will be demonstrated with the following formula proposed by J. Pelletier in an AAR newsletter. We have written the problem in Prolog-like notation, i.e., with variables in capital letters and function and predicate symbols in small letters. A semi-colon is used when more than one positive literal is in a clause.

```
<- p(a,b).
<- q(c,d).
p(X,Z) <- p(X,Y), p(Y,Z).
q(X,Z) <- q(X,Y), q(Y,Z).
p(X,Y) <- p(Y,X).
p(X,Y) ; q(X,Y) <-.
```

The transformation starts by forming the Horn contrapositives for the input clauses, as shown in Section 9.2. To simulate the negation sign, predicate symbols are preceded with labels, `p_` for positive literals and `n_` for negative literals. Additionally, start clauses are added as Prolog queries.

Furthermore, to overcome the incompleteness of Prolog for non-Horn formulae, we need to simulate the reduction operation. This is done as follows. First, we are adding the paths as additional arguments to the logical arguments of the respective literals. For optimization purposes, we use two path lists, one for the positive and one for negative literals in the respective path. In each extension steps, the respective path list is extended by the respective literal. Finally, for actually enabling the performance of reduction steps, an additional clause is added for each predicate symbol and sign that tries all unifiable literals in the path list. The output then looks as follows.

```
% Start clauses
false :- p_p(a,b, [ ],[ ]).

false :- p_q(c,d, [ ],[ ]).

% Contrapositives
n_p(a,b, P, N).

n_q(c,d, P, N).

p_p(X,Z, P,N) :- N1 = [ p(X,Z) | N ], p_p(X,Y, P,N1), p_p(Y,Z, P,N1).
n_p(X,Y, P,N) :- P1 = [ p(X,Y) | P ], n_p(X,Z, P1,N), p_p(Y,Z, P1,N).
n_p(Y,Z, P,N) :- P1 = [ p(Y,Z) | P ], n_p(X,Z, P1,N), p_p(X,Y, P1,N).

p_q(X,Z, P,N) :- N1 = [ q(X,Z) | N ], p_q(X,Y, P,N1), p_q(Y,Z, P,N1).
n_q(X,Y, P,N) :- P1 = [ q(X,Y) | P ], n_q(X,Z, P1,N), p_q(Y,Z, P1,N).
n_q(Y,Z, P,N) :- P1 = [ q(Y,Z) | P ], n_q(X,Z, P1,N), p_q(X,Y, P1,N).

p_p(X,Y, P,N) :- N1 = [ p(X,Y) | N ], p_p(Y,X, P,N1).
n_p(Y,X, P,N) :- P1 = [ p(Y,X) | P ], n_p(X,Y, P1,N).

p_p(X,Y, P,N) :- N1 = [ p(X,Y) | N ], n_q(X,Y, P,N1).
p_q(X,Y, P,N) :- N1 = [ q(X,Y) | N ], n_p(X,Y, P,N1).

% Clauses for performing reduction steps
n_p(X,Y, P,N) :- member(p(X,Y), N).
p_p(X,Y, P,N) :- member(p(X,Y), P).
n_q(X,Y, P,N) :- member(q(X,Y), N).
p_q(X,Y, P,N) :- member(q(X,Y), P).

member(X,[ X | R ]).
member(X,[ Y | R ]) :- member(X,R).
```

What is missing in order to perform complete proof search, is the implementation of a completeness bound and the iterative deepening. We consider the case of the tableau depth bound (Section 3.3.1), which can be implemented by adding the remaining depth resource D as an additional argument to the literals in the contrapositives and start clauses. After having entered a contrapositive, it is checked whether the current depth resource is > 0, in which case it is decremented by 1 and the new resource is passed to the subgoals of the clause. For start clauses, the depth may be passed unchanged to the subgoals.

```
% for contrapositives
P(...,D) :- D > 0, D1 is D-1, P1(...,D1), ..., Pn(...,D1).

% for start clauses
false(D):- P1(...,D), ..., Pn(...,D).
```

When posing the query, say, `false(5)`, the Prolog backtracking mechanism will automatically ensure that all connection tableaux up to tableau depth 5 are examined. Finally, the iterative deepening is captured by simply adding the following clause to the end of the program.

```
false(D) :- D1 is D+1, false(D1).
```

After having loaded such a program into Prolog (in certain Prolog systems the clauses have to be ordered such that all predicates occur consecutively), one can start the proof search by typing in the query: `?- false(1)`.

For the discussed example, the Prolog unification (which in general is unsound) poses no problem, since no function symbol of arity > 0 occurs. In the general case, however, one has to use sound unification. Some Prolog systems offer sound unification, often in various ways. Either the system has a sound unification predicate in its library or sound unification can be switched on by setting a global flag. While the latter is more comfortable, it may lead to unnecessary run-time inefficiencies, since the occurs-check is always performed even if it would not be needed according to the optimizations discussed in the previous parts. Such an optimization may also be achieved in a Prolog implementation by linearization of the clause heads (for which then Prolog unification may be used) and a subsequent sound unification of the remaining critical terms (see, for example, [Plaisted, 1984]).

In summary, this illustrates how astoundingly simple it is to implement a pure connection tableau proof search procedure in Prolog. Furthermore, such an implementation also attains a very high performance in terms of inference steps performed per second. The approach of using Prolog, however, is becoming more and more problematic when trying to implement connection tableaux proof procedures including more advanced search pruning mechanisms like, e.g., failure caching.

9.5 A Data-Oriented Architecture

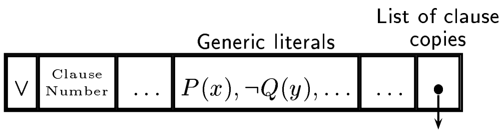
The architectures described so far have all relied on the approach of compiling the input clauses and some parts of the inference system into procedures, the latter created Prolog source code, the others native or abstract machine instructions. The inference rules and important subtasks such as the unification algorithm, the backtracking mechanism, or the subgoal processing are deeply intertwined and standardized in order to achieve high efficiency. Such an approach is suitable when a certain kind of optimized proof procedure has evolved for which no obvious improvements are known. In automated theorem proving, however, this is not the case. New techniques are constantly developed which may lead to significant improvements. Against this background, the most important shortcoming of PTTP provers is their inflexibility. Changing the unification such as to add sorts, for example, or adding new inference rules, e.g., for equality handling, or generalizing the backtracking procedure becomes extremely cumbersome if not impossible in such architectures.

Accordingly, as the last of the architectures, we discuss a more natural or straightforward implementation of the connection tableau procedure, in the sense that the components of the program are modularized and can be identified more naturally with their mathematical definitions. Since the most important difference to PTTP-style provers is that clauses are represented as data structures and do not become part of the prover program, such an approach will be called a data-oriented proof procedure, as opposed to the clause compilation procedures. Unlike the WAM-based architectures, which heavily rely on the implicitit encoding of the proof in the program execution scheme, here the proof object is the clausal tableau, which is completely stored in memory. Although this leads to a larger memory consumption, it causes no problems in practice, as today's computers have enough main storage space to contain the proof trees for practically all *feasible* proof problems. Only very large proofs, that means proofs with more than, say, 100,000 inferences become unfeasible with the data-oriented concept. But in these cases it is to be expected anyway that the connection tableau calculus is not suitable as a proof system.

9.5.1 The Basic Data Structures

The data objects used in this approach can be distinguished into *formula data objects* and *proof data objects*. The basic formula data objects are the formula, the clauses, the clause copies and the subgoals. From these, the formula is implicitly represented by the set of its clauses (as there is only one input formula). Reasonable data structures for the other objects are given here.

Clauses. The most important elements of the clause structure are the original or *generic* literals and the list of clause copies used in the proof. Since clauses may be entered at any subgoal, it is not necessary to compute contrapositives.



Clause copies. In every extension step, a renamed copy of the original clause has to be created and added to the tableau.

	Subgoals				Predecessor
$\vee_c$	$sg_1, \ldots, sg_n$	...	Clause	...	Path

Subgoals. Subgoal objects contain the information about the literal they represent, i.e. the sign, predicate symbol, the argument terms, etc.

sg	Sign	Predicate Symbol	Argument Terms	Extension	Clause Copy	Links	Selection tag	...

As a matter of fact, additional control information can be included in these data objects, which is omitted here for the sake of clarity. Further important data structures utilized in the proof process are the variable trail (which was described in Section 9.1.1) and the list of subgoals. The variable trail is one of the few concepts adopted from the Prolog architecture, since some device for the bookkeeping of the variable instantiations is required.

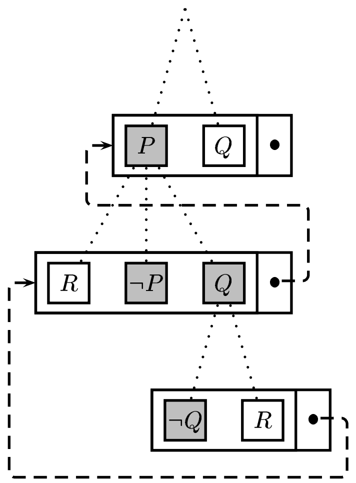


Figure 9.6: The global subgoal list and the corresponding tableau structure.

Global subgoal list. This is the central global data object. It consists of the sequence of subgoals of all clause copies hitherto in the current tableau. Figure 9.6 illustrates how the subgoals of the clause copies constitute the global subgoal list. The dotted lines refer to the underlying tree structure, the dashed arrows indicate the linking between the elements of the global subgoal list. In any inference step, the literal at which an extension or reduction step is performed, is marked as selected, as illustrated in Figure 9.6 as a grey shading over the subgoals.

The global access to the list of subgoals liberates us from the need to conform to some sort of depth-first search. Instead one can employ a subgoal selection function that chooses an arbitrary subgoal for the next inference step. This way, new heuristics become feasible that operate globally on the proof object. For example, free subgoal reordering can be performed easily.

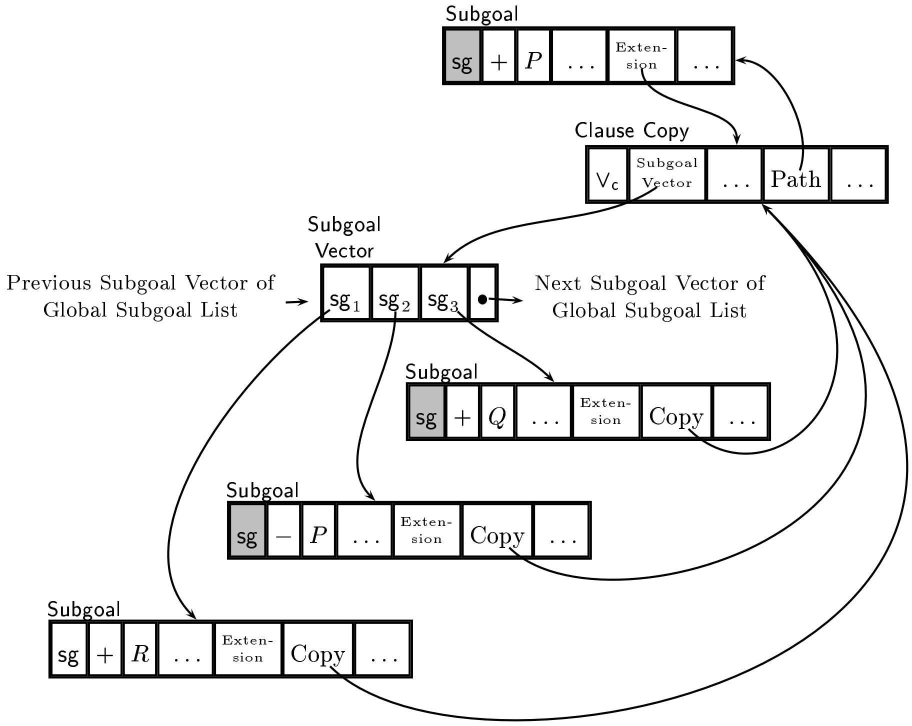


Figure 9.7: Cross-referencing between clause copy and subgoal data structures.

In Figure 9.7, a detailed snapshot of the subgoal and clause copy data structures of a certain proof state is depicted. The tableau structure is only given implicitly. In fact, all information needed for moving through the tableau, as, for instance, during a folding up operation, is provided by extensive cross-referencing among the different data objects. The figure shows the connections between data objects of the various kinds (original clauses as data objects are not contained in the tableau). Again, selected subgoals are distinguished by grey shading. The subgoal P has been chosen for an extension step with the clause $C = \{R, \neg P, Q\}$. A copy C' of C is linked to P via the extension pointer. The subgoals of C' are accessible via the subgoal vector pointed to by C'. This subgoal vector is appended to the global subgoal list. To allow upward movement through the tableau, the copy is linked to the extended subgoal, while the new subgoals are linked to the clause copy. The subgoals P and $\neg P$ are immediately marked as selected, the subgoal Q becomes selected in the next extension step. It should be noted that, since we rely on clauses instead of contrapositives, the connected literal does not have to be the first literal in the clause, as is the case here.

9.5.2 The Proof Procedure

Table 9.3 shows a simplified data-oriented proof procedure (not featuring the reduction rule or start clause selection). Based on the connection graph of the input formula, to each subgoal the list of its links is attached. The procedure solve explores the search space by successively applying the extension rule using the elements in the link list of its subgoal argument *sg*. The reduction rule can be incorporated easily as an additional inferential alternative. The procedure extension checks the resource bound, adds the linked clause to the proof tree, and modifies the global subgoal list. When a literal can be selected, solve is called again with the new subgoals, otherwise a proof has been found and the procedure aborts.

9.5.3 Reuse of Clause Instances

How can high performance be achieved with such an architecture? When analyzing what is the most expensive procedure in this approach, one easily recognizes that it is the generation of a new instance of an input clause, which has to be performed in every extension step. One of the main reasons for the high performance of the PTTP based connection tableau procedures is that this operation is implemented very efficiently. But the question is, whether it is really necessary to generate a new clause instance in every extension step. Typically, proof search procedures based on connection tableaux process relatively small tableaux, but a large amount of them. That is, in theorem proving, the degree of backtracking is extremely high if compared with typical Prolog applications. Many Prolog executions require deep deduction trees including optimizations like tail recursion. For those applications, a new generation of clause instances is indispensable. This striking difference of the deduction trees considered in Prolog and in theorem proving shows that central ingredients of Prolog technology may not be needed in theorem proving.

The key idea for achieving high performance when clause copying is expensive is the *reuse* of clause instances. The clause copies created once are not discarded upon backtracking but kept in a list of available copies for later reuse, as illustrated with the example in Figure 9.8. At startup (subfigure (a)), one uninstantiated copy is provided for each clause. This copy is used in an extension and instantiated, as shown in subfigure (b). Now no other copy is available. When the clause is selected for an extension step again, a new copy has to be created. This situation is shown in subfigure (c). When backtracking occurs during the search process and the extension step that initiated the creation of the copy in subfigure (c) is undone, the copy remains in the list of clause copies and only the pointer to the next available copy is moved backward. This situation is displayed in subfigure (d). This way, over the duration of the proof, a monotonically growing list of clause copies is built and in most cases clause copies can be reused instead of

```
procedure solve( sg, links, resource )
    if ( links ≠ ∅ ) then
      extension( sg, first( links ), resource );
      /* try next alternative */
      solve( sg, rest( links ), resource );
    endif;

procedure extension( sg, link, resource )
    dec_resource := decrement_resource( resource );
    if ( dec_resource > 0 ) then
      clause := new_clause_copy( link );
      head := head( clause, link );
      trail_pos := trailmarker;
      if ( unify_literals( ~sg, head ) ) then
        old_subgoals := subgoals;
        make_new_subgoals( clause, sg, head );
        new_sg := select_subgoal;
        if ( new_sg )
          new_links := links( new_sg );
          new_resource := resource( new_sg );
          solve( new_sg, new_links, new_resource );
        else
          proof_found;
          abort;
        endif;
        /* backtracking */
        unbind( trailpos );
        subgoals := old_subgoals;
      endif;
    endif;
```

Table 9.3: A rudimentary connection tableau proof procedure.

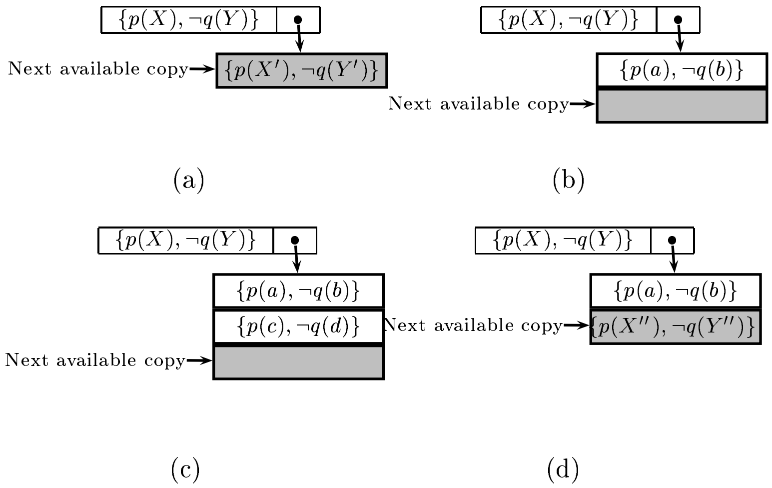


Figure 9.8: Clause instance creation and availability during backtracking.

having to be created. As a matter of fact, this requires that all variable bindings have to be retracted. But when using destructive unification and the trail concept this can be done very efficiently. Experimental results have shown that with such an architecture inference rates may be obtained that are comparable to the ones achieved with PTTP implementations.

9.6 Existing Connection Tableaux Implementations

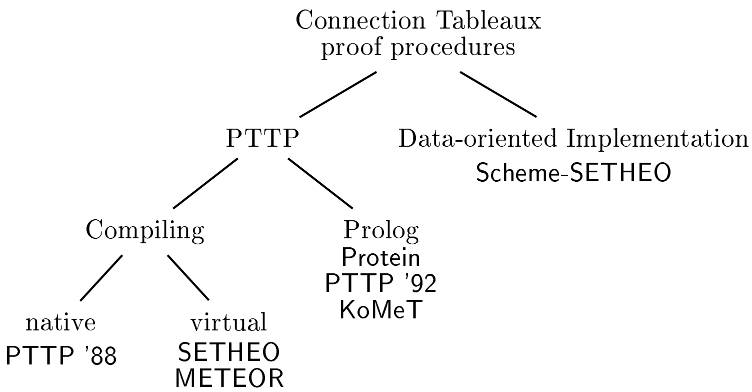


Figure 9.9: An overview of the architectures of connection tableaux systems.

In Figure 9.9, existing implementations of connection tableaux are classified according to the distinctions used in this section. The references for the depicted systems are: PTTP '88 [Stickel, 1988], SETHEO [Letz et al., 1992], METEOR [Astrachan and Loveland, 1991], Protein [Baumgartner and Furbach, 1994], PTTP '92 [Stickel, 1992], KoMeT [Bibel et al., 1994], Scheme-SETHEO (see Section 9.5).

Chapter 10

Constraint Technology

When considering the presented tableau refinements like regularity, tautology, or subsumption freeness, the question may be raised whether it is possible with tenable cost to check these conditions after each inference step. Note that a unification operation in one part of a tableau can produce instantiations which may lead to an irregularity, tautology, or subsumed clause in another distant part of the tableau. The structure violation can even concern a closed part of the tableau. Fortunately, there exists a *uniform* and *highly efficient* technique for implementing many of the presented search pruning mechanisms, namely, *syntactic disequation constraints.*

10.1 Reformulating Refinements as Constraints

Tautology elimination

Let us illustrate the technique first at the example of dynamic tautology elimination. Recall that certain input clauses may have tautological instances, which can be avoided as tableau clauses. When considering the transitivity clause $\neg P(x,y) \vee \neg P(y,z) \vee P(x,z)$ from above, there are two classes of instantiations which may render the formula tautological. Either x and y are instantiated to the same term, or y and z. Obviously, the generation of a tautological instance can be avoided if the unification operation is constrained by forbidding that the respective variables be instantiated to the same terms. In general, this leads to the formulation of *disequation constraints* of the form $s_1, \ldots, s_n \neq t_1, \ldots, t_n$ where the s_i and t_i are terms. Alternatively, one could formulate this instantiation prohibition as a disjunction $s_1 \neq t_1 \vee \cdots \vee s_n \neq t_n$. A disequation constraint is violated if *every* pair $\langle s_i, t_i \rangle$ in the constraint is instantiated to the same term. For the transitivity clause, the two disequation constraints $x \neq y$ and $y \neq z$ can be generated and added to the transitivity formula. The non-tautology constraints for the formulae of a given input set can be generated in a preprocessing phase *before* starting the actual proof process. Afterwards, the tableau construction is performed with *constrained clauses.* Whenever a constrained clause is to be used for tableau expansion, the formula and its constraints are consistently renamed, the tableau expansion is performed with the clause part, and the constraints are added. If the constraints are violated, then a tautological tableau clause has been generated, in which case one can immediately perform backtracking.

Regularity

Regularity can also be captured using disequation constraints. In contrast to non-tautology constraints, however, regularity constraints have to be generated dynamically during the proof search. Whenever a new renamed variant c of a (constrained) clause is attached to a branch in an extension step, then, for every literal L with argument sequence $s_1, \ldots, s_n$ in the clause c and for every branch literal with the same sign and predicate symbol with arguments $t_1, \ldots, t_n$, a disequation constraint $s_1, \ldots, s_n \neq t_1, \ldots, t_n$ must be generated.

Tableau clause subsumption

Tableau clause subsumption is essentially treated in the same manner as tautology elimination. Recall the example from Section 5.1.3 where in addition to the transitivity clause a unit clause $P(a, b)$ is assumed to be in the input set. Then, the disequation constraint $x, z \neq a, b$ may be generated and added to the transitivity clause. Like non-tautology constraints, non-subsumption constraints can be computed and added to the formulae in the input set before the actual proof process is started.[1] Interestingly, this mechanism does not capture *every* case of tableau clause subsumption, as illustrated with the following example. Assume that the transitivity clause and a unit clause $P(f(v), g(v))$ be contained in the input set. In analogy to the other example, a disequation constraint $x, z \neq f(v), g(v)$ could be added to the transitivity formula. But now the constraint contains the variable v, which does not occur in the transitivity clause. Since clauses (and their constraints) are always renamed before being integrated into a tableau, the renaming of the variable v will occur in the constraint only and nowhere else in the tableau. Consequently, this variable can never be instantiated by tableau inference steps, so that the constraint can never be violated and is therefore absolutely useless for search pruning. Clearly, the case of full subsumption cannot be captured in this manner. What the constraint mechanism should avoid is that x and z be instantiated to any terms which have the *structures* $f(t)$ and $g(t)$, respectively, regardless what t is. This can be conveniently achieved by using *universal variables* in addition to the rigid variables. The respective disequation constraint then reads $\forall v \; x, z \neq f(v), g(v)$, which is violated exactly when x and z are instantiated to any terms of the structures $f(s)$ and $g(t)$ with $s = t$.

[1] Note, however, that due to the NP-completeness of subsumption, it might be necessary not to generate *all* possible non-subsumption constraints, since this could involve an exponential preprocessing time.

10.2 Disequation Constraints

After this motivation for the potential of constraints, we will now more rigourously present the framework of disequation constraints, with respect to their use in pruning tableau proof search.

Definition 10.1 (Disequation constraint) A *disequation constraint* C is either true or of the form $\forall u_1 \cdots \forall u_m\ l \neq r$ $(m \geq 0)$ with l and r being sequences of terms $s_1, \ldots, s_n$ respectively $t_1, \ldots, t_n$ $(n \geq 0)$; for any disequation constraint C of the latter form, $l \neq r$ is called the *kernel* of C, n its *length*, $u_1, \ldots, u_m$ its *universal variables*, and the disequation constraints $s_i \neq t_i$ are termed the *subconstraints* of C. Occasionally, we will use the *disjunctive form* of a disequation constraint kernel, which is $s_1 \neq t_1 \vee \cdots \vee s_n \neq t_n$.

An example of a disequation constraint of length one and with one universal variable is

$$\forall z\ f(g(x, a, f(y)), v) \neq f(g(z, z, f(z)), v).$$

Since all considered constraints will be disequation constraints, we will simply speak of constraints in the sequel. Next, we consider what it means that a substitution violates a constraint.

Definition 10.2 (Constraint violation, equivalence) No substitution *violates* the constraint true. A substitution σ *violates* a constraint of the form $\forall u_1 \cdots \forall u_m\, l \neq r$ if there is a substitution τ with domain $u_1, \ldots, u_m$ such that $l\tau\sigma = r\tau\sigma$. When a violating substitution exists for a constraint, we say that the constraint *can be violated*; a constraint *is violated*, if all substitutions violate it. Two constraints are *equivalent* if they have the same set of violating substitutions.

For example, the substitution $\sigma = \{x/f(a)\}$ violates the constraint $\forall y\ x \neq f(y)$, since $x\tau\sigma = f(y)\tau\sigma$ for $\tau = \{y/a\}$.

10.2.1 Constraint Normalization

The question is, how the violation of a constraint may be detected in an efficient manner? The basic property that permits an efficient constraint handling is that constraints can often be simplified. For example, the complex constraint given after Definition 10.1 is equivalent to the simpler constraint $x, y \neq a, a$, which obviously can be handled more efficiently. Constraints can always be expressed in a specific form.

Definition 10.3 (Constraint in solved form) A constraint is in *solved form* if it is either true or otherwise its kernel has the form $x_1, \ldots, x_n \neq t_1, \ldots, t_n$ where all variables on the left-hand side are pairwise distinct and non-universal (i.e., do not occur in the quantifier prefix of the constraint), and no variable x_i occurs in terms of the right-hand side.

For example, the constraint $x, y \neq a, a$ is in solved form whereas the equivalent constraint $x, y \neq y, a$ is not. Every constraint can be rewritten into solved form by using the following nondeterministic algorithm.

Definition 10.4 (Constraint normalization) Let C be any disequation constraint as input. If the constraint is true or the two sides l and r of its kernel are not unifiable, then the constraint true is a *normal form* of C. Otherwise, let σ be any minimal unifier for l and r that contains no binding of the form x/u where u is a universal variable of C and x not. (Such a minimal unifier always exists if l and r are unifiable.) Let $\{x_1/t_1, \ldots, x_n/t_n\}$ be the set of all bindings in σ with the x_i being non-universal in C, and let $u_1, \ldots, u_m$ be the universal variables in C that occur in some of the terms t_i. The constraint $\forall u_1 \cdots \forall u_m \; x_1, \ldots, x_n \neq t_1, \ldots, t_n$ is a *normal form* of C.

Note that, for preserving constraint equivalence and for achieving solved form, the use of a minimal unifier is needed in the procedure, employing just most general unifiers will not always work. Consider, for example, the constraint $f(y) \neq x$ and the most general unifier $\{x/f(x), y/x\}$. This would yield the constraint $x, y \neq f(x), x$ as a normal form, which is not equivalent to $f(y) \neq x$ and which cannot even be violated.

Let us illustrate the effect of the normalization procedure by applying it to the complex constraint $\forall z \; f(g(x, a, f(y)), v) \neq f(g(z, z, f(z)), v)$ mentioned above. First, we obtain the minimal unifier $\{x/a, z/a, y/a\}$. Afterwards, the binding z/a is deleted, since z is universal, which eventually yields the normal form constraint $x, y \neq a, a$. As shown with this example, in the normalization process some constraint variables may vanish. On the other hand, the length of a constraint may increase during normalization.

Proposition 10.5 Any normal form of a constraint C is in solved form and equivalent to C.

Proof If C is true or if the two sides of its kernel are not unifiable, then the normal form of C is true, which is in solved form and equivalent to C. It remains to consider the case of a constraint $C = \forall u_1 \cdots u_m \; l \neq r$ with unifiable l and r. When normalizing C according to the procedure in Definition 10.4, first, a minimal unifier σ for l and r is computed which does not bind non-universal variables to universal ones. The kernel $l' \neq r'$ of the corresponding normal form C' of C contains exactly the subconstraints $x_i \neq t_i$ for every binding $x_i/t_i \in \sigma$ with non-universal x_i. Since minimal unifiers are idempotent, no variable in the domain of σ occurs in terms of its range. Therefore, C' is in solved form. For considering the equivalence of C and C', first note the following. Since σ is a minimal unifier for l and r, it is idempotent and more general than any unifier for l and r. Therefore, a substitution ρ unifies l and r if and only if, for every binding $v/s \in \sigma$, $v\rho = s\rho$. Let now θ be any substitution.

1. If θ violates C, then there exists a substitution τ with domain $\{u_1, \ldots, u_m\}$ and $l\tau\theta = r\tau\theta$. Therefore, for any binding $v/s \in \sigma$, $v\tau\theta = s\tau\theta$, i.e., $\tau\theta$ is a

unifier for l' and r'. Let τ' be the set of bindings in τ with domain variables occurring in C'. Then $\tau'\theta$ unifies l' and r'. Consequently, θ violates C'.

2. If θ violates C', then there exists a substitution τ with its domain being the universal variables in C' and $l'\tau\theta = r'\tau\theta$. Let σ' be the set of bindings in σ which bind universal variables. Then, for any binding $v/s \in \sigma$, $v\sigma'\tau\theta = s\sigma'\tau\theta$, and hence $\sigma'\tau\theta$ unifies l and r. Since $\sigma'\tau$ is a substitution with domain $\{u_1, \ldots, u_m\}$, θ violates C.

10.3 Implementing Disequation Constraints

We will discuss now how the constraint handling can be efficiently integrated into a model elimination proof search procedure. First, we consider the problem of generating constraints in normal form.

10.3.1 Efficient Constraint Generation

Unification is a basic ingredient of the normalization procedure mentioned above. In the successful implementations of model elimination, a destructive variant of the unification procedure specified in Table 9.1 is used. If slightly extended, this procedure can also be used for an efficient constraint generation. First, one must be able to distinguish universal variables from non-universal ones. The best way to do this is to extend the internal data structure for variables with an additional cell where it is noted whether the variable is universal or not. The advantage of this approach is that the type of a variable may change during the proof process which nicely goes together with the feature of local variables mentioned in Section 6.3. Then, the mentioned unification operation must be modified in order to prevent that a non-universal variable is bound to a universal one. After these modifications, the generation and normalization of a constraint can be implemented efficiently by simply misusing the new unification procedure, as follows.

Definition 10.6 (Constraint generation) Given any two sequences l and r of terms that must not become equal by instantiation.

1. Destructively unify l and r and push the substituted variables on the trail.

2. Collect the respective bindings of the non-universal variables only.

3. Finally undo the unification.

After that the term sequences l and r are in their original form, and the collected bindings represents the desired disequation constraint in normal form.

10.3.2 Efficient Constraint Propagation

During proof search with disequation constraints, every tableau is accompanied by a set of constraints. When an inference step is performed, it produces a substitution which is applied to the tableau. In order to achieve an optimal pruning of the search space, after each inference step, it should be checked whether the computed substitution violates one of the constraints of the tableau. If so, the respective inference step can be retracted, we call this a *constraint failure*. If not, the substitution σ has to be propagated to the constraints, i.e., every constraint C has to be replaced by $C\sigma$ before the next inference step is being executed. As a matter of fact, if some of the new constraints can no more be violated, they should be ignored for the further proof attempt. This is important for reducing the search effort, since normally, a wealth of constraints will be generated during proof search.

If the constraints are always kept in normal form, then the mentioned operations can be performed quite efficiently. Assume, for example, that a substitution $\sigma = \{x/a\}$ is applied to the current tableau. Then it is obvious that all constraints in which x does not occur on the left-hand side may be ignored. In case no constraint of the current tableau is violated by the substitution σ, for every constraint C containing x on the left-hand side, a new constraint $C\sigma$ has to be created and afterwards normalized, which is still a considerable effort. In order to do this efficiently, new constraints should not be generated explicitly, but the old constraints should be reused and modified appropriately. For this purpose, it is more comfortable to keep the constraints in disjunctive form. Then, for any such constraint C, only the respective subconstraint $(x \neq t)\sigma$ needs to be normalized to, say C', and the former subconstraint $x \neq t$ in C can be replaced with the subconstraints of C'. This operation may also change the *actual length* of the former constraint. In summary, this results in the following procedure for constraint propagation.

Definition 10.7 (Constraint propagation) All constraints are assumed to be normalized and in disjunctive form. Suppose a substitution $\sigma = \{x_1/s_1, \ldots, x_n/s_n\}$ is performed during the tableau constraints, then successively, for every subconstraint $C_i = x_i/t_i$ (i.e., with x_i in the domain of σ) of every constraint C, compute the normal form C_i' of $s_i \neq t_i\sigma$ with length, say k, and perform the following operation:

1. if $C_i' = \mathsf{true}$, ignore C for the rest of the proof attempt (it cannot be violated),
2. if $k = 0$, decrement the actual length of C by 1; if the actual length 0 is reached, perform backtracking (the constraint is violated),
3. otherwise replace C_i with C_i' and modify the actual length of C by adding $k - 1$.

In order to guarantee efficiency, all modifications performed on the constraints have to be stored intermediately and undone on backtracking.

10.3.3 Internal Representation of Constraints

Obviously, a precondition for the efficiency of the constraint handling is a suitable internal representation of the constraints. When analyzing the described constraint handling algorithms, one has to satisfy the following requirements.

1. after the instantiation of any variable x, a quick access to all subconstraints of the form $x \neq t$ is needed.

2. if a subconstraint C_i of a constraint C is violated, it must be easy to check without considering the other subconstraints of C whether C is violated.

3. whenever a subconstraint C_i of a constraint C normalizes to true, then it must be easy to deactivate C and all other subconstraints of C.

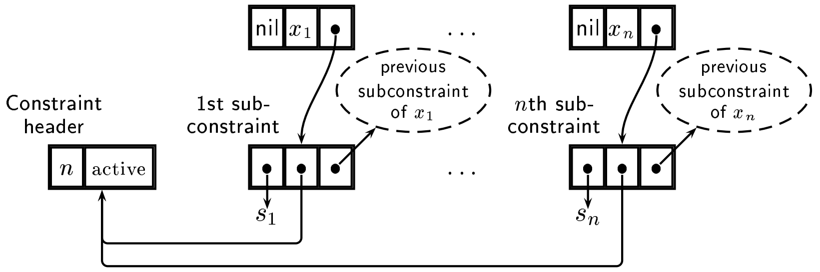


Figure 10.1: Internal representation of a constraint $x_1, \ldots, x_n \neq s_1, \ldots, s_n$.

This can be achieved by using a data structure as displayed in Figure 10.1. In order to have immediate access from a variable x to all subconstraints of the form $x \neq t$, it is reasonable to maintain a list of the subconstraints corresponding to each variable. The best solution is to extend the data structure of a variable by a pointer to the last element in its subconstraint list. From this subconstraint the previous subconstraint of x can be accessed, and so forth. (The aforementioned tag which expresses whether a variable is universal or not is omitted in the figure.)

A constraint itself is separated into a *constraint header* and its subconstraints. The header contains the actual length of the constraint and a tag whether the constraint is already true or whether it can still be violated (active). From each subconstraint there is a pointer to the respective constraint header. If now a subconstraint is violated, then the length counter in the header is decremented by 1. If, on the other hand, a subconstraint normalizes to true, then the tag in the header is set to true. Because of the shared data structure, both modifications are immediately visible and can be used from all other subconstraints of the constraint. Interestingly, an explicit access from a constraint header to its subconstraints is not needed.

It is comfortable to reserve a special part of the memory for the representation of constraints, which we call the *constraint stack*. In order to comprehend the modifications of the constraint stack during the proof process for the case of a

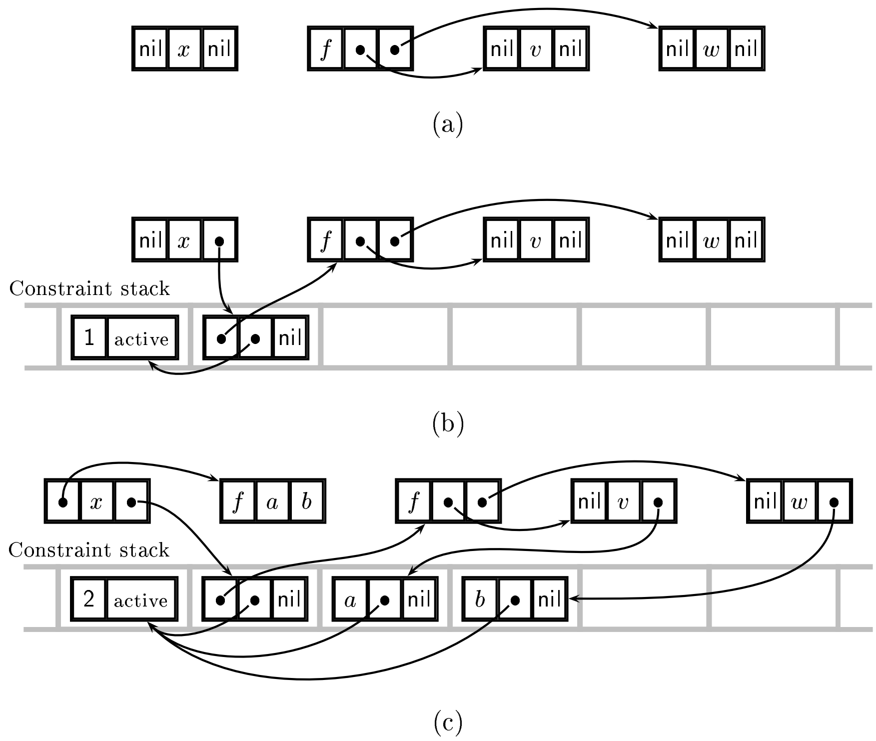


Figure 10.2: The constraint stack.

more complex normalization operation, consult Figure 10.2. Assume we are given a tableau with subgoals $P(x)$ and $\neg Q(x)$ and a predecessor literal $P(f(v,w))$. Assume that no constraints for the variables x, v and w exist (a). Now, a regularity constraint $x \neq f(v,w)$ may be generated, which requires that a constraint header and a subconstraint are pushed on the constraint stack (b). Assume that afterwards an extension step is performed at the subgoal $\neg Q(x)$ with an entry literal $Q(f(a,b))$. The unifier $\sigma = \{x/f(a,b)\}$ has to be propagated to the constraints. This is done by pushing the two new subconstraints $v \neq a$ and $w \neq b$ on the constraint stack, which were obtained after normalization. Furthermore, the subconstraint lists of v and w have to be extended. Finally, the counter in the constraint header has to be incremented by 1. Note that nothing has to be done to the old subconstraint $x \neq f(v,w)$. Since the variable x has been bound, the old subconstraint will simply be ignored by all subsequent constraint checks.

10.3.4 Constraint Backtracking

The entire mechanism of constraint generation and propagation has to be embedded into the backtracking driven proof search procedure of model elimination. Accordingly, also all modifications performed on the constraint stack and in the subconstraint lists of the variables have to be properly undone when an inference step is retracted. For this purpose, after each inference step and the corresponding modifications in the constraint area, one has to remember the following data.

1. the old length values in the affected constraint headers,
2. the old values (active or true) in the second cells of the affected constraint headers, and
3. the old pointers to the previous subconstraints in the affected variables and subconstraints.

This is exactly the information that has to be stored for backtracking. A comfortable method for doing this would be the use of a *constraint trail* similar to the variable trail except that here also the old values need to be stored—note that the variable trail only has to contain the list of bound variables. Additionally, in order to permit the reuse of the constraint stack, one has to remember the top of the constraint stack before each sequence of constraint modifications.

10.4 Disequation Constraints in Prolog

Some Prologs offer the possibility of formulating disequation constraints. As an example, we consider the Prolog system Eclipse [Wallace and Veron, 1993]. Here, using the infix predicate ~= one can formulate syntactic disequation constraints. This permits that constraints resulting from structural tableau conditions can be easily implemented. We describe the method for regularity constraints on the first contrapositive of the transitivity clause

```
p_p(X,Z, P,N) :- N1 = [p(X,Z)|N], p_p(X,Y, P,N1), p_p(Y,Z, P,N1).
```

taken from the Prolog example in Section 9.4. We show how regularity can be integrated by modifying the clause as follows.

```
p_p(X,Z, P,N) :- N1 = [ p(X,Z) | N ],
                 not_member(p(X,Z), P),
                 not_member(p(X,Y), N1),
                 not_member(p(Y,Z), N1),
                 p_p(X,Y, P,N1), p_p(Y,Z, P,N1).
```

where not_member is defined as:

```
not_member(_,[ ]).
not_member(E,[F|R]) :- E ~= F, not_member(E,R).
```

With similar methods an easy integration of tautology and subsumption constraints can be achieved. However, when it comes to the integration of more sophisticated constraints like the ones considered next, it turns out that an efficient Prolog implementation is very hard to obtain.

10.5 Constraints for Global Pruning Methods

In this section, we describe how the matings pruning and the local failure mechanism can be implemented efficiently and even improved by using constraints technology.

Improving the matings pruning using constraints

With the matings pruning mechanism described in Section 5.3.1 one can avoid that certain permutations of matings are considered more than once. The idea was to impose an ordering on the literals in the input formula, which is inherited to the tableau nodes. Now, a reduction step from a subgoal N to an ancestor node N' may be avoided if the entry node N'' immediately below N' is smaller than N in the ordering. In fact, this method can also be captured and even improved by using disequation constraints, as follows. The prohibition to perform a reduction step on N using N' may be reexpressed as a disequation constraint $l \neq r$ where l and r are the argument sequences of the literals at N and N', respectively. Interestingly, such a constraint does prune not only the respective reduction step, but all tableaux in which the literals at N and N'' become equal by instantiation. In [Letz, 1998b] it is proven that this extension of the matings pruning preserves completeness, the main reason being that the matings pruning is compatible with regularity.

Failure caching using constraints

The failure caching mechanism described in Section 5.3.3 can also be implemented using disequation constraints. Briefly, the method requires that when a subgoal N is solved with a solution substitution σ and the remaining subgoals cannot be solved with this substitution, then σ is turned into a failure substitution and, for any alternative solution substitution τ for N, σ must not be more general than τ.

Definition 10.8 (Constraint of a failure substitution) Let σ be a failure substitution generated at a subgoal N and V the set of variables on the path with leaf N in the last tableau in which the subgoal N was selected for an inference step. The *constraint of the failure substitution* σ is the normal form of the constraint $\forall u_1 \cdots \forall u_m \; x_1, \ldots, x_n \neq t_1, \ldots, t_n$ where $u_1, \ldots, u_m$ are the variables occurring in terms of σ that are not in V.

It is straightforward to recognize that a failure substitution σ of a tableau node N is more general than a solution substitution τ of N if and only if the constraint of the failure substitution σ is violated by τ. Consequently, the constraint handling mechanism can be used to implement failure caching. In order to capture failure caching adequately with constraints, the use of universal variables is also needed, like for the case of tableau clause subsumption (Section 10.1). This can be seen by considering, for example, a subgoal N with failure substitution $\sigma = \{x/f(z,z)\}$ where z is a variable not occurring in the set V. The constraint of σ is $\forall z\ x \neq f(z,z)$. When N can be solved with a solution substitution $\tau = \{x/f(a,a)\}$, then σ is more general than τ and, in fact, the constraint $\forall z\ x \neq f(z,z)$ is violated by τ. Obviously, without universal variables it is impossible to capture such a case.

Centralized management of constraints

It is apparent that structural constraints resulting from different sources, tautology, regularity, subsumption, or matings, need not be distinguished in the tableau construction. Furthermore, in general, the constraints need not even be tied to the respective tableau clauses, but the constraint information can be kept separate in a special constraint store. This also fits in with the method of forgetting closed parts of a tableau and working with subgoal trees instead, for all relevant structure information of the solved part of the tableau is contained in the constraints. However, when structural constraints are used in combination with constraints resulting from failure substitutions, in certain states of the proof process constraints have to be deactivated, as shown in Section 5.3.2 and Section 5.3.3. In this case, it is necessary to take the tableau positions into account at which the respective constraints were generated.

Conclusion

Summary

This work is an attempt to provide a comprehensive presentation of tableau and connection calculi for automated deduction in classical logic. We have introduced the essential concepts of semantic tableaux for classical first-order logic both for the closed formula and the free variable case. Structural differences like confluence and nondestructiveness have been discussed, and we have expounded the consequences of the violation of these conditions for proof search. With the integration of connections as "active" control structures into the tableau framework, a new quality is achieved. We have presented and compared a number of such connection conditions concerning structural properties like confluence and nondestructiveness and their use for the pruning of search spaces. Due to the rich structure of tableau deductions, if compared with flat calculi like resolution, a wealth of methods for redundancy elimination have been developed. Those techniques can be nicely classified into local and global methods. Local methods work by identifying structural deficiencies at single tableaux, whereas global methods consider entire sets of deductions and perform an inter-tableau pruning. We have analyzed in detail the most influential techniques from each class and have shown which of the refinements can be combined. An interesting result to be emphasized here is that the minimality concept from the matings framework is not compatible with the regularity restriction on tableaux.

A further central topic of this work is the consideration of issues of computational complexity. A number of results on the polynomial simulation between different tableau variants are given, including a complete and comprehensible proof of the polynomial simulation of clausal tableaux with atomic cut by connection tableaux with folding up. We have also addressed the problem of estimating the sizes of search spaces of bounded search procedures. For all three paradigms, inference, depth, and multiplicity bound, completeness results with respect to important complexity classes are given, some of which have interesting consequences for the assessment of certain paradigms of proof search. For example, there is an exponential decision procedure for multiplicity-bounded search, but the standard iterative-deepening paradigm needs doubly exponential time. An interesting open problem here is whether the latter also holds when all pruning methods are used that have been considered in this work.

The third topic of this work is the presentation of the techniques for an efficient implementation of clausal tableaux using the strict connection condition, which is the most successful tableau calculus in automated deduction. Because of the close relationship of connection tableaux with SLD-resolution, the basis of the programming langugae Prolog, all of the existing implementations use Prolog technology. There are two completely different paradigms which we both describe in detail. One approach is to extend a Prolog compiler like the Warren abstract machine towards a theorem prover for full clause logic, the other is to use Prolog itself as a programming language. Motivated by the missing flexibility of both paradigms, we also describe a non-compilative approach, which is more modular and hence offers a higher degree of flexibility, which may be needed for the implementation of future connection tableau systems. The key idea of achieving efficiency with this method is the extensive reuse of generated data structures. Independently of the used architecture, a method for the efficient implementation of the developed refinements is needed. Here the use of constraint technology is the most promising alternative. Therefore we have presented all machinery that is needed to represent pruning methods like regularity or failure caching by using term disequation constraints.

Future Research

The connection tableau approach is an interesting an successful paradigm in automated deduction. There are, however, severe deficiencies, which have to be addressed in the future. One of the fundamental weaknesses of connection tableaux is the handling of equality. The naïve approach, which is to simply add the congruence axioms of equality, suffers from the severe deficiency that equality specific redundancy elimination techniques are ignored. The most successful paradigm for treating equality in saturation-based theorem proving, *ordered paramodulation*, is not compatible with connection tableaux. There have been attempts to integrate *lazy paramodulation*, a variant of paramodulation without orderings which is compatible with model elimination. This method is typically implemented by means of a transformation (like Brand's modification method), which eliminates the equality axioms and compiles certain equality inferences into the formula. A certain search space pruning might be obtained by using limited ordering conditions [Bachmair et al., 1998], preferable implemented as ordering constraints. This would fit well with the constraint technology applicable in connection tableaux.

Another, more general weakness of the search procedure is that it typically performs poor on formulae with relatively long proofs. On the one hand, this has directly to do with the methodology of iterative-deepening search. On the other hand, when proofs are becoming longer, the goal-orientedness loses its reductive power. To prove difficult formulae in one big leap by reasoning backwards from the conjecture is very hard. An interesting perspective here is the use of *lemmata*, intermediate results typically deduced in a forward manner from the axioms. Some progress has been made in this direction by the development of powerful filtering techniques.

A further interesting line of research could be the use of pruning methods based on semantic information. One could, for example, use small models of the axioms in order to detect the unsolvability of certain subgoals. Finally, the consideration of confluent and possibly even nondestructive integrations of connection conditions into the tableau framework definitely deserves attention.

Bibliography

[Aho et al., 1974] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA.

[Aho and Ullman, 1977] Aho, A. V. and Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley, Reading, MA, USA. See also the much expanded subsequent book [Aho et al., 1986].

[Andrews, 1981] Andrews, P. B. (1981). Theorem proving through general matings. *Journal of the Association for Computing Machinery*, 28:193–214.

[Astrachan and Stickel, 1992] Astrachan, O. and Stickel, M. (1992). Caching and Lemmaizing in Model Elimination Theorem Provers. In Kapur, D., editor, *Proceedings, 11th International Conference on Automated Deduction (CADE), Saratoga Springs, NY, USA*, volume 607 of *LNAI*, pages 224 – 238. Springer.

[Astrachan and Loveland, 1991] Astrachan, O. L. and Loveland, D. W. (1991). METEORs: High performance theorem provers using model elimination. Technical Report Technical report DUKE–TR–1991–08, Department of Computer Science, Duke University. Printed copies available from T.R. Librarian, Dept. of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129.

[Baaz and Fermüller, 1995] Baaz, M. and Fermüller, C. G. (1995). Non-elementary speedups between different versions of tableaux. In *Proceedings of the 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 217–230.

[Baaz and Leitsch, 1992] Baaz, M. and Leitsch, A. (1992). Complexity of resolution proofs and function introduction. *Annals of Pure and Applied Logic*, 57(3):181–215.

[Bachmair et al., 1998] Bachmair, L., Ganzinger, H., and Voronkov, A. (1998). Elimination of equality via transformation with ordering constraints. In Kirchner, C. and Kirchner, H., editors, *Proceedings, 15th International Conference on Automated Deduction (CADE), Lindau, Germany*, volume 1421 of *LNAI*, pages 175–190. Springer.

[Baumgartner, 1998] Baumgartner, P. (1998). Hyper tableau — the next generation. In de Swart, H., editor, *Proceedings, International Conference Tableaux'98, Oisterwijk, The Netherlands*, volume 1397 of *LNAI*, pages 60–76.

[Baumgartner et al., 1999] Baumgartner, P., Eisinger, N., and Furbach, U. (1999). A confluent connection calculus. In Ganzinger, H., editor, *Proceedings, 16th International Conference on Automated Deduction (CADE), Trento, Italy*, LNAI 1632, pages 329–343. Springer.

[Baumgartner and Furbach, 1994] Baumgartner, P. and Furbach, U. (1994). PROTEIN: A PROver with a Theory Extension INterface. In Bundy, A., editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 769–773, Berlin. Springer.

[Baumgartner and Furbach, 1998] Baumgartner, P. and Furbach, U. (1998). Variants of clausal tableaux. In Bibel, W. and Schmitt, P. H., editors, *Automated Deduction — A Basis for Applications*, volume I: Foundations, pages 73–101. Kluwer, Dordrecht.

[Beckert and Hähnle, 1998] Beckert, B. and Hähnle, R. (1998). Analytic tableaux. In Bibel, W. and Schmitt, P. H., editors, *Automated Deduction — A Basis for Applications*, volume I: Foundations, pages 11–41. Kluwer, Dordrecht.

[Beckert et al., 1993] Beckert, B., Hähnle, R., and Schmitt, P. (1993). The even more liberalized δ-rule in free variable semantic tableaux. In *Computational Logic and Proof Theory, Proceedings of the 3rd Kurt Gödel Colloquium*, pages 108–119.

[Beckert and Posegga, 1994] Beckert, B. and Posegga, J. (1994). leanT^AP: Lean tableau-based theorem proving. extended abstract. In Bundy, A., editor, *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France*, LNCS 814, pages 793–797. Springer.

[Benker et al., 1989] Benker, H., Beacco, J. M., Bescos, S., Dorochevsky, M., Jeffré, T., Pöhlmann, A., Noyé, J., Poterie, B., Sexton, A., Syre, J. C., Thibault, O., and Watzlawik, G. (1989). KCM: A knowledge crunching machine. In Yoeli, M. and Silberman, G., editors, *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 186–194, Jerusalem, Israel. IEEE Computer Society Press.

[Bernays and Schönfinkel, 1928] Bernays, P. and Schönfinkel, M. (1928). Zum Entscheidungsproblem der Mathematischen Logik. *Mathematische Annalen*, pages 342–372.

[Beth, 1955] Beth, E. W. (1955). Semantic Entailment and Formal Derivability. *Mededlingen der Koninklijke Nederlandse Akademie van Wetenschappen*, 18(13):309–342.

[Beth, 1959] Beth, E. W. (1959). *The Foundations of Mathematics.* North–Holland, Amsterdam.

[Bibel, 1981] Bibel, W. (1981). On Matrices with Connections. *Journal of the Association for Computing Machinery*, pages 633–645.

[Bibel, 1987] Bibel, W. (1987). *Automated Theorem Proving.* Vieweg, Braunschweig, second revised edition.

[Bibel et al., 1994] Bibel, W., Bruening, S., Egly, U., and Rath, T. (1994). KoMeT. In *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France*, volume 814 of *LNAI*, pages 783–787. Springer.

[Billon, 1996] Billon, J.-P. (1996). The disconnection method: a confluent integration of unification in the analytic framework. In Migliolo, P., Moscato, U., Mundici, D., and Ornaghi, M., editors, *Proceedings of the 5th International Workshop on Theorem Proving with analytic Tableaux and Related Methods (TABLEAUX)*, volume 1071 of *LNAI*, pages 110–126, Berlin. Springer.

[Boy de la Tour, 1990] Boy de la Tour, T. (1990). Minimizing the number of clauses by renaming. In Stickel, M. E., editor, *10th International Conference on Automated Deduction (CADE)*, LNCS, pages 558–572, Kaiserslautern, Germany. Springer.

[Bry and Yahya, 1996] Bry, F. and Yahya, A. (1996). Minimal model generation with positive unit hyper-resolution tableaux. In Miglioli, P., Moscato, U., Mundici, D., and Ornaghi, M., editors, *5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX '96)*, LNAI, pages 143–159, Terrasini, Palermo, Italy. Springer.

[Chang and Lee., 1973] Chang, C. and Lee., R. (1973). *Symbolic Logic and Mechanical Theorem Proving.* Academic Press.

[Church, 1936] Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics.*

[Cook, 1971] Cook, S. A. (1971). The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, pages 151–158.

[Cook and Reckhow, 1974] Cook, S. A. and Reckhow, R. A. (1974). On the lengths of proofs in the propositional calculus. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pages 135–148.

[Corbin and Bidoit, 1983] Corbin, J. and Bidoit, M. (1983). A Rehabilitation of Robinson's Unification Algorithm. In *Information Processing*, pages 909–914. North–Holland.

[d'Agostino, 1999] d'Agostino, M. (1999). Tableau methods for classical propositional logics. In D'Agostino, M., Gabbay, D., Hähnle, R., and Posegga, J., editors, *Handbook of Tableau Methods*, pages 45–124. Kluwer.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem proving. *Communications of the Association for Computing Machinery*, pages 394–397.

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, pages 201–215.

[Dowling and Gallier, 1984] Dowling, W. and Gallier, J. (1984). Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1:267–284.

[Eder, 1985] Eder, E. (1985). Properties of Substitutions and Unifications. *Journal of Symbolic Computation*, 1:31–46.

[Egly, 1997] Egly, U. (1997). Non-elementary speed-ups in proof length by different variants of classical analytic calculi. In *Proceedings of the International Conference on Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX)*, pages 158–171.

[Fitting, 1990] Fitting, M. C. (1990). *First-Order Logic and Automated Theorem Proving*. Springer.

[Fitting, 1996] Fitting, M. C. (1996). *First-Order Logic and Automated Theorem Proving*. Springer, second revised edition.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman.

[Gentzen, 1935] Gentzen, G. (1935). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431. Engl. translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[Gödel, 1930] Gödel, K. (1930). Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360.

[Goller et al., 1994] Goller, C., Letz, R., Mayr, K., and Schumann, J. M. P. (1994). SETHEO V3.2: Recent developments. In Bundy, A., editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 778–782, Berlin. Springer.

[Hähnle and Pape, 1997] Hähnle, R. and Pape, C. (1997). Ordered tableaux: Extensions and applications. In Galmiche, D., editor, *Proceedings, International Conference Tableaux'97, Pont-a-Mousson, France*, volume 1227 of *LNAI*, pages 173–76.

[Hähnle and Schmitt, 1994] Hähnle, R. and Schmitt, P. (1994). The liberalized δ-rule in free variable semantic tableaux. *Journal of Automated Reasoning*, pages 211–221.

[Harrison, 1996] Harrison, J. (1996). Optimizing proof search in model elimination. In McRobbie, M. A. and Slaney, J. K., editors, *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, volume 1104 of *LNAI*, pages 313–327, Berlin. Springer.

[Herbrand, 1930] Herbrand, J. J. (1930). Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et des Lettres de Varsovie, Cl. III, math.-phys.*, pages 33–160.

[Hilbert and Ackermann, 1928]
Hilbert, D. and Ackermann, W. (1928). *Grundzüge der theoretischen Logik.* Springer. Engl. translation: Mathematical Logic, Chelsea, 1950.

[Hintikka, 1955] Hintikka, K. J. J. (1955). Form and Content in Quantification Theory. *Acta Philosophica Fennica*, 8(7):7–55.

[Huet, 1976] Huet, G. (1976). *Resolution d'equations dans les languages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université de Paris VII.

[Huet, 1980] Huet, G. (1980). Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the Association for Computing Machinery*, pages 797–821.

[Kleene, 1967] Kleene, S. C. (1967). *Mathematical Logic.* Wiley.

[Klingenbeck and Hähnle, 1994] Klingenbeck, S. and Hähnle, R. (1994). Semantic tableaux with ordering restrictions. In Bundy, A., editor, *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France*, LNCS 814, pages 708–722. Springer.

[Korf, 1985] Korf, R. E. (1985). Iterative-deepening-A: An optimal admissible tree search. In Joshi, A., editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 1034–1036, Los Angeles, CA. Morgan Kaufmann.

[Kowalski and Kuehner, 1971] Kowalski, R. and Kuehner, D. (1971). Linear resolution with selection function. *Artificial Intelligence*, 2:227–260.

[Kowalski and Hayes, 1969] Kowalski, R. A. and Hayes, P. (1969). Semantic tress in automated theorem proving. *Machine Intelligence*, pages 87–101.

[Kowalski and Kuehner, 1970] Kowalski, R. A. and Kuehner, D. (1970). Linear resolution with selection function. Technical report, Metamathematics Unit, Edinburgh University, Edinburgh, Scotland.

[Krivine, 1971] Krivine, J.-L. (1971). *Introduction to Axiomatic Set Theory.* Reidel, Dordrecht.

[Lassez et al., 1988] Lassez, J.-L., Maher, M. J., and Marriott, K. (1988). Unification Revisited. In *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann.

[Lee and Plaisted, 1992] Lee, S.-J. and Plaisted, D. (1992). Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, pages 25–42.

[Letz, 1993a] Letz, R. (1993a). *First-order calculi and proof procedures for automated deduction.* PhD thesis, TH Darmstadt.

[Letz, 1993b] Letz, R. (1993b). On the polynomial transparency of resolution. In Bajcsy, R., editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI), Chambery, France*, pages 123–129. Morgan Kaufmann.

[Letz, 1998a] Letz, R. (1998a). On the complexity of the formula instantiation problem. Technical report, Technische Universität München.

[Letz, 1998b] Letz, R. (1998b). Using matings for pruning connection tableaux. In Kirchner, C. and Kirchner, H., editors, *Proceedings, 15th International Conference on Automated Deduction (CADE), Lindau, Germany*, volume 1421 of *LNAI*, pages 381–396. Springer.

[Letz, 1999a] Letz, R. (1999a). First-order tableaux methods. In D'Agostino, M., Gabbay, D., Hähnle, R., and Posegga, J., editors, *Handbook of Tableau Methods*, pages 125–196. Kluwer.

[Letz, 1999b] Letz, R. (1999b). Properties and relations of tableaux and connection calculi. In Hölldobler, S., editor, *Intellectics and Computational Logic.* Kluwer. To appear.

[Letz et al., 1994] Letz, R., Mayr, K., and Goller, C. (1994). Controlled integration of the cut rule into connection tableau calculi. *Journal of Automated Reasoning*, 13(3):297–338.

[Letz et al., 1989] Letz, R., Schumann, J., and Bayerl, S. (1989). SETHEO: A SEquentiell THEOremprover for first order logic. Technical Report FKI-97-89, Technische Universität München, München, Germany.

[Letz et al., 1992] Letz, R., Schumann, J., Bayerl, S., and Bibel, W. (1992). SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212.

[Loveland, 1968] Loveland, D. W. (1968). Mechanical theorem proving by model elimination. *Journal of the Association for Computing Machinery*, 15(2):236–251. Reprinted in: [Siekmann and Wrightson, 1983].

[Loveland, 1969] Loveland, D. W. (1969). A simplified format for the model elimination theorem-proving procedure. *Journal of the Association for Computing Machinery*, 16(3):349–363.

[Loveland, 1972] Loveland, D. W. (1972). A unifying view of some linear Herbrand procedures. *Journal of the Association for Computing Machinery*, 19(2):366–384.

[Loveland, 1978] Loveland, D. W. (1978). *Automated theorem proving: A logical basis*. North Holland, New York.

[Loveland, 1991] Loveland, D. W. (1991). Near-horn Prolog and beyond. *Journal of Automated Reasoning*, 7:1–26.

[Manthey and Bry, 1988] Manthey, R. and Bry, F. (1988). Satchmo: a theorem prover implemented in prolog. In *Proceedings of the 9th Conference on Automated Deduction (CADE)*, pages 456–459.

[Martelli and Montanari, 1976] Martelli, A. and Montanari, U. (1976). Unification in Linear Time and Space: a Structured Presentation. Technical report, Ist. di Elaboratione delle Informatione, Consiglio Nazionale delle Ricerche, Pisa, Italy.

[Martelli and Montanari, 1982] Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, pages 258–282.

[Mayr, 1993] Mayr, K. (1993). Refinements and extensions of model elimination. In Voronkov, A., editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning (LPAR'93)*, volume 698 of *LNAI*, pages 217–228, St. Petersburg, Russia. Springer Verlag.

[Moser et al., 1997] Moser, M., Ibens, O., Letz, R., Steinbach, J., Goller, C., Schumann, J., and Mayr, K. (1997). SETHEO and E-SETHEO—the CADE-13 systems. *Journal of Automated Reasoning*, 18(2):237–246.

[Ohlbach, 1991] Ohlbach, H.-J. (1991). Semantics Based Translation Methods for Modal Logics. *Journal of Logic and Computation*, 1(5):691–746.

[Orevkov, 1979] Orevkov, V. P. (1979). Lower bounds for increasing complexity of derivations after cut elimination. *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im V. A. Steklova AN SSSR*, pages 137–161.

[Paterson and Wegman, 1978] Paterson, M. S. and Wegman, M. N. (1978). Linear Unification. *Journal of Computer and Systems Sciences*, pages 158–167.

[Plaisted, 1984] Plaisted, D. A. (1984). The occur-check problem in prolog. In *1984 International Symposium on Logic Programming*. IEEE, New York, USA ISBN 0 8186 0522 7. U.S. Copyright Clearance Center Code: CH2007-3/84/000-0272$01.00.

[Plaisted, 1994] Plaisted, D. A. (1994). The search efficiency of theorem proving strategies. In Bundy, A., editor, *Proceedings of the 12th International Conference on Automated Deduction (CADE)*, LNAI 814, pages 57–71. Springer.

[Plaisted and Greenbaum, 1986] Plaisted, D. A. and Greenbaum, S. (1986). A structure preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304.

[Plaisted and Zhu, 1997] Plaisted, D. A. and Zhu, Y. (1997). *The Efficiency of Theorem Proving Strategies*. Vieweg.

[Prawitz, 1960] Prawitz, D. (1960). An improved proof procedure. *Theoria*, 26:102–139. Reprinted in [Siekmann and Wrightson, 1983].

[Prawitz, 1969] Prawitz, D. (1969). Advances and Problems in Mechanical Proof Procedures. In *Automation of Reasoning, 1983 (reprinted)*, pages 285–297. Springer.

[Reckhow, 1976] Reckhow, R. A. (1976). *On the Lenghts of Proofs in the Propositional Calculus*. PhD thesis, University of Toronto.

[Reeves, 1987] Reeves, S. V. (1987). Adding equality to semantic tableau. *Journal of Automated Reasoning*, 3:225–246.

[Robinson, 1965] Robinson, J. A. (1965). A Machine-oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, pages 23–41.

[Robinson, 1968] Robinson, J. A. (1968). The Generalized Resolution Principle. *Machine Intelligence*, pages 77–94.

[Schumann, 1991] Schumann, J. (1991). *Efficient Theorem Provers based on an Abstract Machine*. PhD thesis, TU Mnchen.

[Shostak, 1976] Shostak, R. E. (1976). Refutation graphs. *Artificial Intelligence*, 7:51–64.

[Siekmann and Wrightson, 1983] Siekmann, J. and Wrightson, G., editors (1983). *Automation of Reasoning*. Springer, Berlin. Two volumes.

[Smullyan, 1968] Smullyan, R. (1968). *First-Order Logic*. Springer.

[Statman, 1979] Statman, R. (1979). Lower Bounds on Herbrand's Theorem. *Proceedings American Math. Soc.*, pages 104–107.

[Stickel, 1984] Stickel, M. E. (1984). A prolog technology theorem prover. In *1984 International Symposium on Logic Programming*. IEEE, New York, USA ISBN 0 8186 0522 7.

[Stickel, 1988] Stickel, M. E. (1988). A prolog technology theorem prover. In Lusk, E. and Overbeek, R., editors, *9th International Conference on Automated Deduction (CADE)*, LNCS, pages 752–753, Argonne, Ill. Springer.

[Stickel, 1992] Stickel, M. E. (1992). A prolog technology theorem prover: a new exposition and implementation in prolog. *Theoretical Computer Science*, 104:109–128.

[Sutcliffe et al., 1994] Sutcliffe, G., Suttner, C., and Yemenis, T. (1994). The TPTP problem library. In Bundy, A., editor, *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France*, LNCS 814, pages 708–722. Springer. Current version available on the *World Wide Web* at the URL `http://www.cs.jcu.edu.au/ftp/users/GSutcliffe/TPTP.HTML`.

[Taki et al., 1984] Taki, K., Yokota, M., Yamamoto, A., Nishikawa, H., ichi Uchida, S., Nakashima, H., and Mitsuishi, A. (1984). Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 398–409, ICOT Research Center, Tokyo, Japan. ICOT.

[Tarski, 1936] Tarski, A. (1936). Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1.

[Tseitin, 1970] Tseitin, G. (1970). On the complexity of proofs in propositional logics. *Seminars in Mathematics*, 8.

[Turing, 1936] Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, pages 230–265.

[van Orman Quine, 1955] van Orman Quine, W. (1955). A Way to Simplify Truth Functions. *American Mathematical Monthly*.

[Vlahavas and Halatsis, 1987] Vlahavas, I. and Halatsis, C. (1987). A new abstract prolog instruction set. In *Expert systems and their applications (Proceedings)*, pages 1025–1050, Avignon.

[Voronkov, 1998] Voronkov, A. (1998). Herbrand's theorem, automated reasoning and semantics tableaux. In *IEEE Symposium on Logic in Computer Science*.

[Wallace and Veron, 1993] Wallace, M. and Veron, A. (1993). Two problems – two solutions: One system – ECLiPSe. In *Proceedings IEE Colloquium on Advanced Software Technologies for Scheduling*, London.

[Wallen, 1989] Wallen, L. (1989). *Automated Deduction for Non-Classical Logic*. MIT Press, Cambridge, Mass.

[Warren, 1983] Warren, D. H. D. (1983). An Abstract PROLOG Instruction Set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA.

Index