First-Order Calculi and Proof Procedures for Automated Deduction

Reinhold Letz

July 1993

Contents

Introduction

The field of automated deduction has reached a state of maturity and seriousness, in which the hope for finding a proof method which is *simple*, *uniform*, *and successful in general* has been rightly given up. On the other hand, a wealth of isolated techniques has been developed so far, which, when combined in an appropriate non-trivial manner, may give rise to such successful proof methods. In a situation like this there are three essential tasks. First, the existing mechanisms, which are typically formulated in different frameworks, need to be *compared* and *classified*, in order to make the similarities and differences transparent. Also, this way a lot of redundant work in related formalisms can be avoided. Secondly, the techniques need to be *evaluated* with respect to two properties particularly relevant for automated deduction, namely, their *inferential* and *reductive* power. Finally, promising ways of *combining* and *integrating* the selected mechanisms need to be identified.

In this work contributions are made to all three of the mentioned tasks. Thus, as opposed to many other investigations in automated deduction, we compare and put forward mechanisms in *different* frameworks in parallel. Furthermore, we develop conceptual tools for *classifying* inference rules and inference systems in a rigorous and computationally reliable manner. For the evaluation of the mechanisms we are using the well-established notion of *polynomial simulation*. Finally, we illustrate that by means of *integrating* methods from different frameworks new promising proof systems for automated deduction can be obtained.

The work is organized in four chapters. In the following we shall give a detailed overview on the main contents of the chapters and their interdependencies.

In the first chapter, which is of a preparatory nature, we discuss basic representation techniques and modification operations on logical expressions. After a complete presentation of the syntax and semantics of first-order logic, which we have included in order to render the work self-contained, we concentrate on more compact representation formats for logical expressions than the ordinary string or tree notation. The standard motivation for employing such representations is that they are needed to make the *unification operation* perform in polynomial time. We show that the failure of achieving polynomial unification with the ordinary data structures is just one *symptom* of their weakness, the more elementary reason being that an *iterative application of substitutions* may lead to an exponential behaviour. We present two types of compact data structures. One are the well-known *directed acyclic graphs*, which are well-suited for a direct transmission to the computer, but unconvenient for a *textual* handling. Therefore, we additionally develop a *string* notation, the language of *definitional expressions*, which permits equally compact formulations as graphs but is better suited for the human use. After introducing the modification operations of matching and unification for ordinary expressions, we describe their polynomial variants using the compact representation formats. We conclude the preparatory chapter with reviewing the most widely used sublanguages and normal forms of the first-order language.

The wealth of logical systems currently developed increases the interest in conceptual frameworks for classifying and comparing different systems. The second chapter of this work is devoted to the development of notions which are fundamental for analyzing the *structural* and *computational* properties of logic calculi. After an explication of the general concepts of logical *relations* and logical *problems*, we stress the importance of distinguishing between the *declarative* and the *operational* or *transitional* aspects of logical systems. Then we present a general framework for measuring the computational *complexities* of arbitrary transition relations and deductions, which are treated as particular transition relations. In order to be able to compare complexities on a level which is as abstract as possible, we subscribe to abstractions modulo *polynomials*, as usual in complexity theory. The central notions emerging this way are the properties of *polynomial transparency* and *weak polynomial transparency.* The polynomial transparency of a transition relation guarantees that the number of rewrite steps in *any* transition sequence represents an adequate measure for the actual computational complexity of the sequence. Weak polynomial transparency is the adequate concept for evaluating the *indeterministic powers* of special transition relations, called *proof relations*, by restricting attention to *shortest proofs* only. The benefit of the framework is twofold, not only does it facilitate the abstract classification of deduction systems, it also may give advice how to improve the systems. This is illustrated most significantly in the fourth chapter, when the developed notions are used on the resolution calculus.

Since the basic design decisions for first-order calculi are settled on the *propositional* level, we follow the common practice of first considering in an extra chapter the propositional or *ground* versions of the calculi developed later on. Also, propositional logic is important in its one right, since it plays a central role in complexity theory, due to the *NP-completeness* of the satisfiability problem. Although the traditional *generative* types of logic calculi, *Frege/Hilbert systems*, *natural deduction* and *sequent calculi*, are relatively strong with respect to *indeterministic power*, i.e., permit the formulation of relatively short proofs, the systems are not suited to a direct automation. This is because the calculi contain too much indeterminism and are lacking in goal-orientedness, which renders it almost impossible to actually *find* short proofs using those systems.

We study in detail two families of logic calculi which are particularly appropriate for the purposes of automated deduction. The first family consists of *resolution systems* and *semantic tree procedures*, which have in common that they use a condensed variant of the *cut rule* from sequent systems. The close relationship between both types of calculi becomes apparent when considering their *declarative proof objects*, which are identical for certain subsystems. Since resolution applies the cut rule in a *forward*, i.e., generative, manner, just like sequent systems, resolution is not suited as a basis for deciding the logical status of propositional formulae. The semantic tree format, which applies the cut rule in a *backward* manner, has proven as the optimal framework for solving propositional formulae in practice. This is because in propositional logic, where the number of interpretations is fixed, the backward cut rule can be viewed as a mechanism of enumerating sets of interpretations in a particularly efficient way.

The other family investigated in this work consists of *tableau systems* and *connection calculi*, which, by their very nature, are *cut-free* proof systems. Since in first-order logic the backward application of the cut rule is problematic, the systems are excellently suited as bases for developing successful *first-order* calculi. By a straightforward amalgamation of the central ideas in both types of calculi, we obtain the *connection tableau* framework, which generalizes the *model elimination calculus*. The main characteristic of connection tableau calculi is their missing *proof-confluence*, that is, not every proof attempt of a provable formula can be completed successfully. This possibility of making irreversible decisions in the calculus demands a different organization of the proof process as in resolution or systematic tableau procedures, namely, as a *deduction* enumeration instead of a *formula* enumeration procedure. Since in connection tableau procedures, in general, *all* deductions need to be enumerated, we consider a number of *structural* refinements which extremely reduce the numbers of deductions with a certain resource. We also show that those refinements may weaken the *indeterministic powers* of the calculi. Due to their cut-freeness, connection tableau calculi are significantly weaker concerning indeterministic power than semantic trees or resolution systems. In order to remedy this weakness, we develop a new controlled variant of the cut rule, the *folding up operation*, which can be applied without introducing to much additional indeterminism. This technique, which is properly more powerful than the *factorization rule* in connection calculi, is presented as an efficient way of *integrating lemmata* into the connection tableau calculus. The folding up operation also gives rise to an additional structural refinement of tableaux, which produces a new promising calculus for automated deduction.

In the fourth chapter we discuss first-order calculi and proof procedures for automated deduction belonging to three classes. First, we consider Herbrand procedures; then fundamental properties of resolution calculi are studied; finally, we develop advanced connection tableau calculi and proof procedures. The presentation follows the historical course of scientific development in the field, since the sixties. Accordingly, we start with a review of the *Herbrand compactness property*, which directly suggests a *two-step methodology* of proving Skolemized first-order formulae, so-called *Herbrand procedures*. While the *naïve* approach works by really enumerating sets of *ground* instances, which afterwards are decided by propositional means, a significant improvement can be achieved by enumerating so-called *multiplicities* of the input, which then are decided by checking whether there exist *unifiable spanning matings*. Although the second approach is superior to the naïve one, it still suffers from the two-step methodology, by employing two relatively *independent* subprocedures. The consideration of sets of Herbrand instances also motivates the introduction of the notion of *Herbrand complexity* as an important complexity measure for the classification of first-order calculi. The Herbrand complexity of a set of formulae S is the minimal size of an unsatisfiable set of ground instances of the formulae in S. This measure gives rise to a natural generalization of the notion of Herbrand procedures to so-called Herbrand *calculi*, which is the class of all calculi for which Herbrand complexity is a lower bound to the sizes of proofs.

Subsequently, in concordance with the historical development, we move over to the first-order resolution calculus, which from the mid-sixties on for fifteen years almost completely absorbed the efforts in automated deduction. The relative success of resolution in automated deduction is due to a particularly prosperous combination of two inference mechanism, namely the *forward cut rule* and the *unification operation*, which achieves optimal variable instantiations. This also illustrates that automated deduction in propositional logic and automated deduction in first-order logic have completely different emphases, with respect to the problems considered as relevant for the respective domain. Thus, in first-order logic, normally, nondenumerably many interpretations exist, so that interpretation-oriented methods like semantic tree procedures cannot be applied. A first-order variant of semantic trees, in which the backward cut rule is generalized appropriately, seems not to exist either. Since resolution refinements and resolution proof procedures have been thoroughly investigated in the literature, we restrict ourselves to the presentation of two closely related fundamental results on resolution. First, we demonstrate that resolution is not polynomially bounded by Herbrand complexity, so that there may exist significantly shorter proofs than in Herbrand calculi. On the other hand, however, first-order resolution lacks polynomial transparency, even in the weak sense. Consequently, the number of inferences in a resolution proof does not give a representative measure of the *actual* complexity of the proof, even if only shortest proofs are considered. We present a class of formulae which have resolution proofs with a polynomial number of inference steps, but for which the *size* of any proof is exponential. Both the superiority of resolution over Herbrand calculi and the intransparency of resolution are due to the possibility of *renaming* the variables in derived clauses, which is a fundamental deduction mechanism. This result motivates the development of new data structures for the representation of formulae.

Since the beginning of the eighties, with the *connection method* and *model elimination*, non-resolution frameworks for automated deduction in first-order logic have been reconsidered. The motivation for the development of alternatives to resolution is the fact that not the *cut rule* is the main reason for the relative success of resolution but the *unification operation*. Consequently, other propositional inference systems than resolution can be made into successful first-order calculi by integrating unification. In fact, the first-order versions of the connection tableau calculi, on which we concentrate, can even more easily be lifted to the first-order case, since only unification is needed for first-order completeness and no additional mechanism, like the factoring rule in resolution. We develop new powerful pruning mechanisms, which can be implemented in a very efficient way, and illustrate the superiority of the tableau format over frameworks like model elimination, by demonstrating the reductive potential of using *free* selection functions. The folding up operation can be integrated smoothly into the first-order version of connection tableaux. We conclude our work with the discussion of two important aspects of connection tableau proof procedures. On the one hand, we show that due to the *permutability* of tableaux, pure uninformed enumeration procedures contain a source of redundancy, which can be removed if information about the *matings* corresponding to the tableaux is used. On the other hand, we point to a further fundamental redundancy, which results from the very nature of any logic calculi working by decomposing problems into subproblems and solving the subproblems separately. In order to avoid this redundancy it is necessary to apply *global* deletion methods which compare *alternative* deductions. This observation motivates the future development of global pruning methods employing information from the proof search itself.

Acknowledgements

Im am indebted to a number of people without whom this work never would have been completed. Most of all I like to thank Eveline Krebs for her constant encouragement and psychological support over the last years. Also, I want to express my gratitude to Eike Jessen, Bertram Fronhöfer, and Ulrich Furbach for their patience in providing the financial and contractual requirements of this undertaking. Scientifically, I have benefitted a lot from the stimulating discussions with Klaus Mayr. Thanks go also to Johann Schumann, Graham Wrightson, and Christoph Goller for proof-reading parts of the work. Theodor Gemenis was a tireless advisor in any questions of lay-out and text formatting.

Furthermore, I would like to thank my co-reviewer Christoph Walther, who identified and helped remove a systematic error in the unification part of the thesis. Finally, I am grateful to my advisor Wolfgang Bibel for introducing me to the subject of automated deduction, for his support during the doctorate procedure, and his comments on an earlier version of the text which resulted in a number of significant improvements.

Munich, July 1993 Reinhold Letz

Chapter 1

First-Order Logic

This chapter presents the basic components of first-order logic. After some computational preliminaries in the first section, in Section 2 the syntax and semantics of ordinary first-order logic is introduced. The automatic processing of logic has revealed that the standard representation of logical expressions is not optimally suited to an efficient computational treatment. For this reason, more compact representational formats are discussed. Section 3 presents the well-known *graphical* encoding of logical expressions by use of directed acyclic graphs. Since graphical representations are very hard to handle textually, in Section 4 a string variant of the graphical encoding is developed, the *definitional first-order language*. Subsequently, we discuss the basic modification mechanisms used in automated deduction, namely, the instantiation operations of *matching* and *unification*. In Section 5 these operations are introduced for ordinary logical expressions, and in Section 6 matching and unification are generalized to the handling of definitional expressions. Section 7 concludes this chapter with the discussion of important sublanguages and normal forms of the first-order language.

1.1 Computational Preliminaries

This work is concerned with giving complexity measures on the space needed for encoding various mathematical objects on a computer and on the space and time needed for manipulating the represented objects. For this purpose uniform and realistic representation models are necessary for describing space and time consumption.

1.1.1 Basic Abstract Machine Models

In order to make the complexity measures independent of actually existing hardware, which is diverse and rapidly changing, it is reasonable to base the considerations on *abstract machine models*, mathematical idealizations of real computers. There are a number of *basic abstract computation* and *machine models* like *Tur-*

ing machines [Aho et al., 1974]—the standard mathematical model for string-oriented computation—or *random access machines* [Cook and Reckhow, 1973]—the idealized von Neumann computer. All these models have in common that one can distinguish between a finite *program*—formulated with a *finite* number of elementary symbols—which is to operate on machine *states* or *configurations*. The program determines for any machine state a set of possible *subsequent* machine states, hence, mathematically, the program defines a *transition relation* between machine states. A *computation* on a machine can then be defined as a sequence of successive machine states. While Turing machines, random access machines, and other generally accepted basic machine models differ in their space and time measures, there seems to be the general assumption that all "realistic" frameworks can simulate each other within a constant factor overhead in space and a polynomially bounded overhead in time (as formulated, for instance, in [van Emde Boas, 1990] which provides an introduction to this subject). In fact, this assumption can be used to *define* realistic machines if, for example, Turing machines are taken as realistic.

1.1.2 Sequences and Strings

In this work, we subscribe to a string-oriented computation model, which is the most natural representation framework for the objects we are dealing with. In order to introduce strings formally some basic definitions are needed.

Definition 1.1.1 (Partial sequence and sequence) Any mapping[1] with its domain being a subset of the positive integers $\mathbb{N}$, while its range may be any set of objects, is called a *partial sequence*. The *length* of a sequence S, written $\mathrm{length}(S)$, is its cardinality, $\mathrm{card}(S)$. A partial sequence S is called *connected* if for arbitrary integers $i < j < k$: whenever $i \in \mathrm{domain}(S)$ and $k \in \mathrm{domain}(S)$, then $j \in \mathrm{domain}(S)$. A connected partial sequence S is named a *sequence* if $1 \in \mathrm{domain}(S)$. A partial connected sequence S is said to be a *subsequence of* a partial connected sequence S' if $S \subseteq S'$. If a subsequence S' of a sequence S is itself a sequence, then S' is termed a *prefix* of S.

Notation 1.1.1 We denote the values $S(i)$ of partial sequences with S_i. Connected partial sequences $\{\langle i, S_i\rangle, \langle i{+}1, S_{i+1}\rangle, \langle i{+}2, S_{i+2}\rangle, \ldots\}$ are written ${}^{i}(S_i, S_{i+1}, S_{i+2}, \ldots)$ where the left index i is omitted for sequences, i.e., for $i = 1$.

Example 1.1.1 Given a sequence of letters S = ('s','t','a','r','t','i','n','g'). The sequence ('s','t','a','r','t') is a prefix of S, the partial connected sequence $S' = {}^{3}$('a','r','t') is a subsequence of S, and the partial sequence $S'' = \{\langle 1, \text{'s'}\rangle, \langle 2, \text{'t'}\rangle, \langle 4, \text{'r'}\rangle, \langle 6, \text{'i'}\rangle, \langle 7, \text{'n'}\rangle, \langle 8, \text{'g'}\rangle\}$, which is a subset of S, is no subsequence of S.

[1] As usual, we view relations and mappings as sets of ordered pairs.

Definition 1.1.2 (Occurrence) For any partial sequence S a partial function nth_S: $\mathbb{N} \longrightarrow \mathrm{range}(S)$ can be defined that maps any positive integer $i \leq \mathrm{card}(S)$ to that value S_j for which there are i elements $\langle k, S_k \rangle$ in S with $k \leq j$. The mapping nth_S is a sequence and is named the *sequence normalization* of the partial sequence S. If S'' is the sequence normalization of a subsequence S' of a sequence S, then S' is called an *occurrence* of S'' in S. Also, all objects in the range of a partial sequence S are said to *occur* in S. We call an occurrence S' of a sequence S'' (an occurrence ${}^{i}(o)$ of an object o) in a sequence S the *k-th occurrence* of S'' (of the object o) in S if there are $k-1$ occurrences of S'' (of the object o) in S such that their least domain numbers are smaller than the least domain number of S' (of ${}^{i}(o)$).

Referring to the sequences given in Example 1.1.1, the sequence (‘s’,‘t’,‘r’,‘i’,‘n’,‘g’) is the sequence normalization of the partial sequence S'', and the subsequence ^{3}(‘a’,‘r’,‘t’) of S is an occurrence of its sequence normalization (‘a’,‘r’,‘t’) in S.

Definition 1.1.3 (Alphabet) An *alphabet* is a countable set of objects, called *symbols.*

Definition 1.1.4 (String) A *string* or *word over* an alphabet is a finite sequence of symbols from the alphabet. A *symbol string* is a string of length 1. Any subsequence S of a string S' is called a *substring* of the string S', and if S is a symbol string, it is said to be a *subsymbol* of S'.

Notation 1.1.2 Normally, we abbreviate any string $(s_1, \ldots, s_n)$ by just writing its symbols side by side as $s_1 \cdots s_n$. For the case of a symbol s, this induces the ambiguity that we do not know whether by writing ‘s’ the symbol or the unary string of this symbol is meant. We shall systematically exploit this ambiguity in that we shall normally not distinguish between a symbol and its symbol string. The context will clear up possible uncertainties.

Definition 1.1.5 (Concatenation of strings) If $W_1, W_2, \ldots, W_n$ are strings with lengths $l_1, l_2, \ldots, l_n$, respectively, then we call the string defined by

$$W(i) = \begin{cases} W_1(i) & \text{for } 1 \leq i \leq l_1 \\ W_2(i - l_1) & \text{for } l_1 < i \leq l_1 + l_2 \\ \quad \cdots & \\ W_n(i - l_1 - \cdots - l_{n-1}) & \text{for } l_1 + \cdots + l_{n-1} < i \leq l_1 + \cdots + l_n \end{cases}$$

the *concatenation* of $W_1, W_2, \ldots, W_n$, which is written $W_1 W_2 \cdots W_n$.

1.1.3 Space and Time Complexity Measures

The space complexity of the objects treated in this work is measured in terms of encodings of the objects as strings over some alphabet. The idea of such encodings is that any unstructured object o is represented as a string over the alphabet, and any finite structure of objects $o_1, \ldots, o_n$ is encoded as a string composed of the object representations plus an appropriate encoding of the structure.

Note By *unstructured* objects we mean objects such that, for any two of them, it can merely be determined whether they are different or identical. Accordingly, the only information expressed in a set S of unstructured objects is the cardinality of the set.

We use a number of space complexity measures. Only one of them is realistic with respect to actual physical devices, the others are all unrealistic but convenient. A *realistic size* of any object o, written $\#(o)$, is the length of an appropriate, i.e., structure-preserving, string representation of the object over a given finite alphabet. All realistic sizes of an object differ only by a constant factor, thus reflecting the assumed correspondence in space complexity between the generally accepted basic machine models mentioned above. Besides the realistic measure, we use various more abstract space complexity measures, written $\mathrm{size}(o)$. The simplest of these *unrealistic* measures takes the length of an appropriate string representation of the object over an *infinite* alphabet as its size. This measure is extremely convenient, because it permits that any unstructured object can be encoded as a symbol string, and hence has the size 1. Since any representation of an object using an infinite alphabet can be encoded as a string of a length of the order $\mathrm{O}(n \log n)$ over a finite alphabet, a realistic size of an object can be easily computed from the mentioned unrealistic size. Occasionally, we shall go further and use situation-dependent space complexity measures which are even more convenient. Unrealistic measures are computationally adequate for those abstract considerations where the trade-off between the realistic and the employed unrealistic measure does not matter. Whenever the representativeness of an unrealistic model is doubtful, we shall relate it to the realistic model—this will be the case in Section 2.3.

The complexity of a computation will be measured in terms of its time complexity, since the space complexity of a computation, as the maximal size of its states, gives only a very rough complexity measure[2]. The time complexity is finer and also has implications on the space complexity. A useful time complexity measure for any computation in a basic machine model is the number of transition steps, i.e., the length of the sequence of configurations minus 1. In order to make this *uniform time* measure a *realistic* measure, it is necessary that the hidden factor, namely, the time spent for a single transition operation, be

[2] Note, however, the strong influence of the increase in size on the time complexity, which is elaborated in Section 2.3.

bounded in a certain way. Here, Turing machines provide a realistic model by permitting the manipulation of only one symbol in each step—manipulations of arbitrarily large structures in one transition, like in the case of a random access machine, demand different measures, like *logarithmic time*, to render them realistic. Under this assumption, all variants of Turing machines and the logarithmic time versions of random access machines are polynomially related (again consult [van Emde Boas, 1990]).

The mentioned realistic measures are defined in terms of the machine resources only, they do not consider the input of a computation as a complexity parameter. Accordingly, a natural abstraction from the realistic time measures is to quantify machine resources with respect to the input sizes of computations. This unrealistic measure is very useful for the computational assessment of a given procedure, since one is often not so much interested in the actual computing time of the procedure but in the *relation* between the input size and the computing time. Also, from this unrealistic measure the actual computing time can be easily obtained. Consequently, this measure, with the abstraction modulo polynomials, will be the standard time complexity measure in our investigations. In Section 2.3, we shall use the input size and the number of steps of a computation as the two ingredients to define a generalized class of basic machine models, related through the property of *polynomial transparency.*

1.2 Syntax and Semantics of First-Order Logic

The language of first-order logic has a structure which is a very convenient and powerful formal abstraction from expressions and concepts occurring in natural language, and, most significantly, in mathematical discourse. The expressions of a first-order language are particular strings over an infinite alphabet of elementary symbols.

1.2.1 First-Order Signatures

Definition 1.2.1 (First-order signature) A *first-order signature*[3] is defined as a pair $\Sigma = \langle \mathcal{A}, a \rangle$ consisting of a denumerably infinite alphabet $\mathcal{A}$ and a partial mapping a: $\mathcal{A} \longrightarrow \mathbb{N}_0$, associating natural numbers with certain symbols in $\mathcal{A}$, called their *arities*, such that $\mathcal{A}$ 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] In this work solely untyped signatures will be used.

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 distinct symbols (,), and , , without arities.

Notation 1.2.1 Normally, variables and function symbols will be denoted 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. With the same letters the corresponding symbol strings will be denoted, too. We shall extend the terminology in such a way that unary strings containing variables, function, or predicate symbols are also called *variables, function*, or *predicate symbols*, respectively—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.

1.2.2 First-Order Expressions

Given a first-order signature Σ, the corresponding *first-order language* is defined inductively[4] as a set of specific strings over the alphabet of the signature. In the following, let $\Sigma = \langle \mathcal{A}, a \rangle$ be a fixed first-order signature.

Definition 1.2.2 (Atomic term) Every (symbol string of a) constant or variable in $\mathcal{A}$ is said to be an *atomic term over* Σ.

Definition 1.2.3 (Term) (inductive)

1. Every atomic term over Σ is a *term over* Σ.

2. If f is (the symbol string of) an n-ary function symbol in $\mathcal{A}$ with an arity $n \geq 1$ and $t_1, \ldots, t_n$ are terms over Σ, then the concatenation $f(t_1, \ldots, t_n)$ is a *term over* Σ.

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

Definition 1.2.4 (Atomic formula) (inductive)

1. Every (symbol string of a) nullary predicate symbol in $\mathcal{A}$ is an *atomic formula*, or just *atom*, *over* Σ.
2. If P is (the symbol string of) an n-ary predicate symbol in $\mathcal{A}$ with an arity $n \geq 1$ 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.

Definition 1.2.5 (Formula) (inductive)

1. Every atom over Σ is a *formula over* Σ.
2. If F and G are formulae over Σ and x is (the symbol string of) a variable in $\mathcal{A}$, then the following concatenations are also *formulae over* Σ:
   $\neg F$, called the *negation* of F,
   $(F \wedge G)$, called the *conjunction* of F *and* G,
   $(F \vee G)$, called the *disjunction* of F *and* G,
   $(F \rightarrow G)$, called the *material implication* of G *by* F,
   $(F \leftrightarrow G)$, called the *material equivalence* of F *and* G,
   $\forall x F$, called the *universal quantification* of F *in* x, and
   $\exists x F$, called the *existential quantification* of F *in* x.

Definition 1.2.6 ((Well-formed) expression) All terms and formulae over Σ are called (*well-formed) expressions over* Σ.

Definition 1.2.7 (First-order-language) The set of all (well-formed) expressions over Σ is called the *first-order language over* Σ, which we write $\mathcal{L}_\Sigma$.

Definition 1.2.8 (Complement) If a first-order formula F has the structure $\neg G$, then G is the *complement* of F, otherwise, i.e., in case F is not a negated formula, then the *complement* of F is $\neg F$.

Notation 1.2.2 The complement of a formula F is denoted with $\sim F$.

Definition 1.2.9 (Subexpression) If an expression Φ is the concatenation of strings $W_1, \ldots, W_n$, in concordance with the Definitions 1.2.2 to 1.2.5, then any expression among these strings is called an *immediate subexpression* of Φ. The sequence obtained by deleting out all strings from $W_1, \ldots, W_n$ which are not expressions is called the *immediate subexpression sequence* of Φ. Among the strings $W_1, \ldots, W_n$ there is a unique symbol string W whose symbol is a connective, a quantifier, a function symbol, or a predicate symbol; W and its symbol are called the *dominating string* and the *dominating symbol* of Φ, respectively. An expression Φ' is said to be a *subexpression* of an expression Φ if the pair $\langle \Phi', \Phi \rangle$ is in the transitive closure of the immediate subexpression relation. Analogously, the notions of (immediate) subterms and (immediate) subformulae are defined.

Example 1.2.1 Presupposing our conventions of denoting symbols and symbol strings, a formula $P(x, f(a, y), x)$ 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 (string).

Definition 1.2.10 (Scope of a quantifier occurrence) Let $S = {}^{i}(Q)$ be a subsymbol of an expression Φ where Q is a quantifier. Due to the fact that Φ is an expression, there must exist an occurrence $S' = {}^{i}(Q, x, \ldots)$ of a quantification in Φ. The substring S' is called the *scope* of S in Φ. Any substring of S' is said to be *in* the scope of S.

Example 1.2.2 In a formula $\forall x \exists y (P(x, y) \lor \forall x \neg P(x, y))$, i.e., in the string $(\forall, x, \exists, y, (, P, (, x, , , y,), \lor, \forall, x, \neg, P, (, x, , , y,),),),)$, the scope of the first quantifier occurrence $(\forall)$ is the whole formula, whereas the scope of ${}^{13}(\forall)$ is the substring ${}^{13}(\forall, x, \neg, P, (, x, , , y,),)$. The subsymbol ${}^{18}(x)$ is both in the scope of $(\forall)$ and in the scope of ${}^{13}(\forall)$.

Definition 1.2.11 (Bound and free variable occurrence) If an occurrence ${}^{i}(x)$ of a variable x in an expression Φ is in the scope S of a quantifier subsymbol ${}^{j}(Q)$ which is immediately followed by an occurrence of the same variable x, and if ${}^{i}(x)$ is not in the scope of some quantifier occurrence ${}^{k}(Q')$ immediately followed by an occurrence of x in a proper substring of S, then ${}^{i}(x)$ is said to be *bound by* ${}^{j}(Q)$. A variable occurrence ${}^{i}(x)$ is called *free* in an expression Φ if ${}^{i}(x)$ is not bound by some quantifier subsymbol of Φ.

Referring to Example 1.2.2, the subsymbol ${}^{18}(x)$ is bound by ${}^{13}(\forall)$, but not by $(\forall)$ or ${}^{3}(\exists)$. Clearly, every occurrence of a variable in a well-formed expression is bound by at most one quantifier subsymbol of the expression.

Definition 1.2.12 (Closed formula) Any formula which does not contain free variable occurrences is called a *closed formula.*

Definition 1.2.13 (Closure of a formula) Let F be a formula F with $\{x_1, \ldots, x_n\}$ being the set of free variables occurring in F, then the formula $\forall x_1 \cdots \forall x_n F$ is called a *universal closure* of F, and the formula $\exists x_1 \cdots \exists x_n F$ is called an *existential closure* of F.

1.2.3 Semantics of First-Order Logic

A *logic* can be viewed as a pair $\langle \mathcal{L}, \mathcal{R} \rangle$ consisting of a logical language $\mathcal{L}$—in our case a first-order language—and a relation $\mathcal{R}$ on the expressions in $\mathcal{L}$. Among the relations on logical expressions, the binary relations of *logical consequence* are most important. Any logical consequence relation attempts to formalize an intuitively given paradigm of correct reasoning. Historically the first definitions of logical consequence relations were formulated in purely syntactic terms, by specifying systems of structural rules for *deducing* logical expressions from logical expressions [Frege, 1879, Hilbert and Ackermann, 1928, Łukasiewicz and Tarski, 1930, Gentzen, 1935]. The sensibility of one thus defined system can be motivated by making plausible that every rule in the system corresponds to an accepted rule in an intuitively given paradigm of correct reasoning. This way, different systems were developed and indeed turned out to define the same consequence relations, thus providing a completely formal motivation for the significance of the specified consequence relations.[5] Among the consequence relations, the relation of *classical logical consequence*, written $\vdash$, plays a central role.

Initiated by the work of Tarski [Tarski, 1936], an alternative way of characterizing logics became customary. Tarski showed that it is possible to give *declarative meaning* to the expressions of a logic language, in analogy to the situation in natural language where certain expressions can be interpreted as denoting objects in the real world. The standard way of giving semantics to a formal language is by specifying mappings, called *interpretations*, from the signature and the expressions of the language to mathematical objects.

Definition 1.2.14 (Universe) Any non-empty set of objects is called a *universe.*

Notation 1.2.3 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 $\bigcup_{n\in\mathbb{N}_0}\mathrm{pow}(\mathcal{U}^n)$ where $\mathrm{pow}(S)$ denotes the power set of a set S. 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.

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 1.2.15 ($\mathcal{L}$-structure, interpretation) An $\mathcal{L}$-*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* an $\mathcal{L}$-structure $\langle\mathcal{L},\mathcal{U}\rangle$ is a mapping $\mathcal{I}$: $\mathcal{F}\cup\mathcal{P}\longrightarrow\mathcal{U}_{\mathcal{F}}\cup\mathcal{U}_{\mathcal{P}}$ such that

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

Definition 1.2.16 (Variable assignment) A *variable assignment from* a first-order language $\mathcal{L}$ *to* a universe $\mathcal{U}$ is a mapping $\mathcal{A}$: $\mathcal{V}\longrightarrow\mathcal{U}$.

[5] Just like the equivalence of different formalizations of computability furnishes a completely formal support for *Church's Thesis.*

Definition 1.2.17 (Term assignment) (inductive)
Let $\mathcal{I}$ be an interpretation for an $\mathcal{L}$-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 $\mathcal{I}^{\mathcal{A}}$: $\mathcal{T} \longrightarrow \mathcal{U}_{\mathcal{F}}$ defined as follows.

1. For every variable x in $\mathcal{V}$: $\mathcal{I}^{\mathcal{A}}(x) = \mathcal{A}(x)$.
2. For every constant c in $\mathcal{F}$: $\mathcal{I}^{\mathcal{A}}(c) = \mathcal{I}(c)$.
3. If f is a function symbol of arity $n > 0$ and $t_1, \ldots, t_n$ are terms, then
$$\mathcal{I}^{\mathcal{A}}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathcal{I}^{\mathcal{A}}(t_1), \ldots, \mathcal{I}^{\mathcal{A}}(t_n)).$$

Definition 1.2.18 (Formula assignment) (by simultaneous induction)
Let $\mathcal{I}$ be an interpretation for an $\mathcal{L}$-structure, 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 $\mathfrak{I}^{\mathcal{A}}$: $\mathcal{W} \longrightarrow \mathcal{U}_{\mathcal{P}}$ defined as follows. Let F and G denote arbitrary formulae of $\mathcal{L}$.

1. For any nullary predicate symbol p in the signature of $\mathcal{L}$: $\mathfrak{I}^{\mathcal{A}}(p) = \mathcal{I}(p)$.
2. If P is a predicate symbol of arity $n > 0$ and $t_1, \ldots, t_n$ are terms, then
$$\mathfrak{I}^{\mathcal{A}}(P(t_1, \ldots, t_n)) = \begin{cases} \top & \text{if } \langle\mathcal{I}^{\mathcal{A}}(t_1), \ldots, \mathcal{I}^{\mathcal{A}}(t_n)\rangle \in \mathcal{I}(P) \\ \bot & \text{otherwise.} \end{cases}$$
3. $$\mathfrak{I}^{\mathcal{A}}((F \vee G)) = \begin{cases} \top & \text{if } \mathfrak{I}^{\mathcal{A}}(F) = \top \quad \text{or} \quad \mathfrak{I}^{\mathcal{A}}(G) = \top \\ \bot & \text{otherwise.} \end{cases}$$
4. $$\mathfrak{I}^{\mathcal{A}}(\neg F) = \begin{cases} \top & \text{if } \mathfrak{I}^{\mathcal{A}}(F) = \bot \\ \bot & \text{otherwise.} \end{cases}$$
5. $$\mathfrak{I}^{\mathcal{A}}((F \wedge G)) = \mathfrak{I}^{\mathcal{A}}(\neg(\neg F \vee \neg G)).$$
6. $$\mathfrak{I}^{\mathcal{A}}((F \rightarrow G)) = \mathfrak{I}^{\mathcal{A}}((\neg F \vee G)).$$
7. $$\mathfrak{I}^{\mathcal{A}}((F \leftrightarrow G)) = \mathfrak{I}^{\mathcal{A}}(((F \rightarrow G) \wedge (G \rightarrow F))).$$
8. Let $\mathcal{A}_x$ be the collection of variable assignments from $\mathcal{L}$ to $\mathcal{U}$ differing from $\mathcal{A}$ in the value of x only.
$$\mathfrak{I}^{\mathcal{A}}(\forall x F) = \begin{cases} \top & \text{if } \mathfrak{I}^{\mathcal{A}'}(F) = \top \text{ for all elements } \mathcal{A}' \text{ of } \mathcal{A}_x \\ \bot & \text{otherwise.} \end{cases}$$
9. $$\mathfrak{I}^{\mathcal{A}}(\exists x F) = \mathfrak{I}^{\mathcal{A}}(\neg\forall x \neg F).$$

Note The induction in Definition 1.2.18 runs over the noetherian ordering on the expressions in $\mathcal{L}$ defined as follows: every expression in the definiens is smaller than the expression in the definiendum. Moreover, *every* formula in $\mathcal{L}$ occurs at the definiendum position in this ordering. Therefore, for any interpretation $\mathcal{I}$ for an $\mathcal{L}$-structure and any variable assignment from $\mathcal{L}$ to $\mathcal{U}$, the respective formula assignment is a total mapping on the language $\mathcal{L}$.

We are particularly interested in interpretations for closed formulae. From the definition of interpretations (item 8) it follows that, for any closed formula and any interpretation $\mathcal{I}$, the respective formula assignments are all identical, and hence do not depend on the variable assignments. Consequently, for closed formulae, we shall speak of *the* formula assigment of an interpretation $\mathcal{I}$, and write it $\mathfrak{I}$.

To comprehend the manner in which formula assignments give meaning to expressions, see Example 1.2.3. The example illustrates how formulae are interpreted in which a variable is associated with different quantifier occurrences. Loosely speaking, Definition 1.2.18 guarantees that variable assignments obey "dynamic binding" rules (in terms of programming), 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 1.2.3 Consider two closed formulae $\Phi = \forall x(\exists x F(x) \wedge G(x))$ and $\Psi = \forall x \exists x(F(x) \wedge G(x))$. Given a universe $\mathcal{U} = \{u_1, u_2\}$, and an interpretation $\mathcal{I}(F) = \mathcal{I}(G) = \{u_1\}$, then $\mathfrak{I}(\Phi) = \bot$ and $\mathfrak{I}(\Psi) = \top$.

Definition 1.2.19 (Model) Let Γ be any set of formulae of a first-order language $\mathcal{L}$. An interpretation $\mathcal{I}$ for an $\mathcal{L}$-structure $\langle\mathcal{L},\mathcal{U}\rangle$ is called a *model for* Γ if, for each variable assignment $\mathcal{A}$ from $\mathcal{L}$ to $\mathcal{U}$, $\mathfrak{I}^{\mathcal{A}}(\Phi) = \top$ for each formula $\Phi \in \Gamma$. If Γ is a singleton set $\{\Phi\}$, we also shall say that $\mathcal{I}$ is a *model for* the formula Φ.

Definition 1.2.20 (Satisfiability, validity) Suppose Γ is any set of formulae of a first-order language $\mathcal{L}$. We call Γ *satisfiable* if there exists a model for Γ. If Γ is not satisfiable, it is named *unsatisfiable*. We say that Γ is *valid* if, for every universe $\mathcal{U}$, every interpretation for $\langle\mathcal{L},\mathcal{U}\rangle$ is a model for Γ. If Γ is not valid, it is termed *invalid*.

Definition 1.2.21 ((Logical) implication, equivalence) Let Γ and Δ be two sets of formulae of a first-order language $\mathcal{L}$. We say that Δ is *(logically) implied by* Γ, written $\Gamma \models \Delta$, if every model for Γ is a model for Δ. If Γ and Δ imply each other, they are named *(logically) equivalent*, written $\Gamma \equiv \Delta$. Again, if one or both sets are singletons, we use the same terminology for their elements.

According to this definition, any first-order formula is logically equivalent to any-one of its universal closures.

Note The logical implication relation $\models$ on the set S of formulae of a first-order language is identical to the classical logical consequence relation $\vdash$ on S, which is defined purely syntactically (this will be shown in Chapter 4 of this work). Accordingly, declarative semantics provides an alternative non-procedural characterization of logical consequence. As such, declarative semantics can be very helpful for many purposes. For instance, it is often much easier to prove the equivalence of two logical rule systems by relating them via the declarative semantics.

The following important equivalences between first-order formulae can be demonstrated easily.

Proposition 1.2.1 *Let F, G, and H be arbitrary first-order formulae.*

(a) $F \equiv \neg\neg F$.

(b) $(F \wedge F) \equiv F$. (∧-idempotency)

(c) $(F \vee F) \equiv F$. (∨-idempotency)

(d) $(F \wedge G) \equiv (G \wedge F)$. (∧-commutativity)

(e) $(F \vee G) \equiv (G \vee F)$. (∨-commutativity)

(f) $(F \leftrightarrow G) \equiv (G \leftrightarrow F)$. (↔-commutativity)

(g) $((F \wedge G) \wedge H) \equiv (F \wedge (G \wedge H))$. (∧-associativity)

(h) $((F \vee G) \vee H) \equiv (F \vee (G \vee H))$. (∨-associativity)

(i) $((F \leftrightarrow G) \leftrightarrow H) \equiv (F \leftrightarrow (G \leftrightarrow H))$. (↔-associativity)

(j) $\neg(F \wedge G) \equiv (\neg F \vee \neg G)$. (De Morgan law for ∧)

(k) $\neg(F \vee G) \equiv (\neg F \wedge \neg G)$. (De Morgan law for ∨)

(l) $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$. (∨-distributivity)

(m) $F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$. (∧-distributivity)

(n) $F \rightarrow G \equiv \neg G \rightarrow \neg F$. (Contraposition)

(o) $\neg\exists x F \equiv \forall x \neg F$. (∃∀-conversion)

(p) $\neg\forall x F \equiv \exists x \neg F$. (∀∃-conversion)

(q) $\forall x(F \wedge G) \equiv (\forall x F \wedge \forall x G)$. (∀∧-permutability)

(r) $\exists x(F \vee G) \equiv (\exists x F \vee \exists x G)$. (∃∨-permutability)

Notation 1.2.4 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 structure $F \circ_1 (G \circ_2 H)$ may be abbreviated by writing just $F \circ_1 G \circ_2 H$ (right bracketing).

Logical formulae possess the fundamental property that under certain conditions subformulae can be substituted by equivalent subformulae without changing the meaning of the formulae.

Lemma 1.2.2 (Replacement Lemma) *Given a formula Φ of a first-order language $\mathcal{L}$ and any subformula F of Φ. If G is any formula which is logically equivalent to F where F and G possess the same sets of free variables, and Ψ is a formula obtained from Φ by replacing some occurrences of F in Φ with G, then Φ and Ψ are logically equivalent.*

Proof Straightforward from the definition of formula assignments. □

The concepts of material (object-level) implication and logical (meta-level) implication of first-order logic are connected in the following simple manner.

Theorem 1.2.3 (Implication Theorem) *Given two closed first-order formulae Φ and Ψ. $\Phi \models \Psi$ if and only if the formula $\Phi \rightarrow \Psi$ is logically valid.*

Proof For the "if"-part, assume $\Phi \rightarrow \Psi$ is logically valid. Let $\mathcal{I}$ be an arbitrary model for Φ. Then, $\mathfrak{I}(\Phi) = \top$. By assumption and Definition 1.2.18, $\mathfrak{I}(\Phi) = \bot$ or $\mathfrak{I}(\Psi) = \top$. Consequently, $\mathfrak{I}(\Psi) = \top$, and $\mathcal{I}$ is a model for Ψ. For the "only-if"-part, suppose $\Phi \models \Psi$. Let $\mathcal{I}$ be an arbitrary interpretation for $\Phi \rightarrow \Psi$. Either, $\mathfrak{I}(\Phi) = \bot$; then, by Definition 1.2.18, $\mathfrak{I}(\Phi \rightarrow \Psi) = \top$. Or, $\mathfrak{I}(\Phi) = \top$; in this case, by assumption, $\mathfrak{I}(\Psi) = \top$, too; hence, by Definition 1.2.18, $\mathfrak{I}(\Phi \rightarrow \Psi) = \top$. Therefore, in either case $\mathcal{I}$ is a model for $\Phi \rightarrow \Psi$. □

1.3 Graphical Representation of Logical Expressions

The ordinary string representation of logical expressions suffers from two weaknesses. On the one hand, the representation does not easily reveal the internal compositional structure of an expression. On the other hand, a certain subexpression may occur multiply within an expression, so that the ordinary string representation is not the most compact format for encoding logical expressions.

1.3.1 Directed Acyclic Graphs

An alternative two-dimensional framework for representing logical expressions is offered by certain graphs.

Definition 1.3.1 (Directed graph) A *directed graph* is a triple $\langle V, E, f \rangle$ where V and E are disjoint sets of objects called *vertices* (or *nodes*) and *edges*, respectively, and f is a total mapping from E into $V \times V$. If $f(e) = \langle v_1, v_2 \rangle$, then the edge e is said to *go out of* or *begin in* the vertex v_1 and to *go into* or *end in* the vertex

v_2; furthermore, the vertex v_1 is called a *predecessor* of the vertex v_2 and v_2 is a *successor* of v_1 in the graph. Any vertex without successors is called a *leaf*, and any vertex without predecessors is called a *root*. A *path in* or *through* a directed graph is any sequence S of edges $(e_1, e_2, e_3, \ldots)$ taken from E such that if e_i ends in a vertex v_j, then e_{i+1} begins in v_j, for every $i > 0$. A *branch* in a dag is a path $S = (e_1, e_2, e_3, \ldots)$ beginning in a root and satisfying, for every e_i in S, whenever e_i ends in a non-leaf vertex, then S contains also an edge e_{i+1}. A directed graph is called *rooted* if it contains exactly one vertex without a predecessor. A directed graph is said to be *acyclic* if no path in the graph contains the same vertex twice.

Definition 1.3.2 (Isomorphy of directed graphs) Two directed graphs $t_1 = \langle V_1, E_1, f_1\rangle$ and $t_2 = \langle V_2, E_2, f_2\rangle$ are said to be *isomorphic* if there are two total and injective mappings α_V from V_1 onto V_2 and α_E from E_1 onto E_2 such that, for any edge $e \in E_1$ with $f_1(e) = \langle v_1, v_2\rangle$: $f_2(\alpha_E(e)) = \langle \alpha_V(v_1), \alpha_V(v_2)\rangle$.

Notation 1.3.1 (Dag) A *dag* is a directed graph which is rooted and acyclic.

Definition 1.3.3 (Dag consistency) A set S of dags is said to be *consistent* if for any two dags $t_1 = \langle V_1, E_1, f_1\rangle$ and $t_2 = \langle V_2, E_2, f_2\rangle$ in S:

1. for every edge $e \in E_1 \cap E_2$: $f_1(e) = f_2(e)$, and
2. for every vertex $v \in V_1 \cap V_2$: each edge going out of v in t_1 is an outgoing edge of v in t_2.

Definition 1.3.4 (Subdag) Let $t = \langle V, E, f\rangle$ be a dag. A dag $\langle V', E', f'\rangle$ is called a *subdag* of t if $V' \subseteq V$, $E' \subseteq E$, $f' \subseteq f$, and $\{t, t'\}$ is consistent.

In Figure 1.1 three dags t_0, t_1, and t_2 are displayed. Nodes are represented by circles and edges by arrows. The entire graph, the dag t_0, is consistent with t_1, and t_1 is consistent with t_2, whereas t_0 and t_2 are not consistent, because the root of t_2 has merely three outgoing edges. Accordingly, t_1 is a subdag of t_0, but t_2 is not.

Definition 1.3.5 (Ordered dag) An *ordered dag* is a pair $\langle t, O\rangle$ consisting of a dag t and a mapping O associating with every vertex a strict linear ordering on its outgoing edges. We say that an edge which is the i-th element in such an ordering is the *i-th edge* of the respective source vertex. A set of ordered dags is said to be *consistent* if the contained dags are consistent and the outgoing edges of any node are ordered in the same way in every dag of the set.

In general, the vertices and edges of dags are only used as index sets and will be labelled with certain objects.

Definition 1.3.6 (Labelled (ordered) dag) A *labelled* (ordered) dag is a pair $\langle t, \lambda\rangle$ consisting of an (ordered) dag and a (possibly partial) labelling function λ on its vertices and edges. A set S of labelled (ordered) dags is said to be *consistent* if the contained (ordered) dags are consistent and every labelled vertex and edge is labelled with the same object in every element of S.

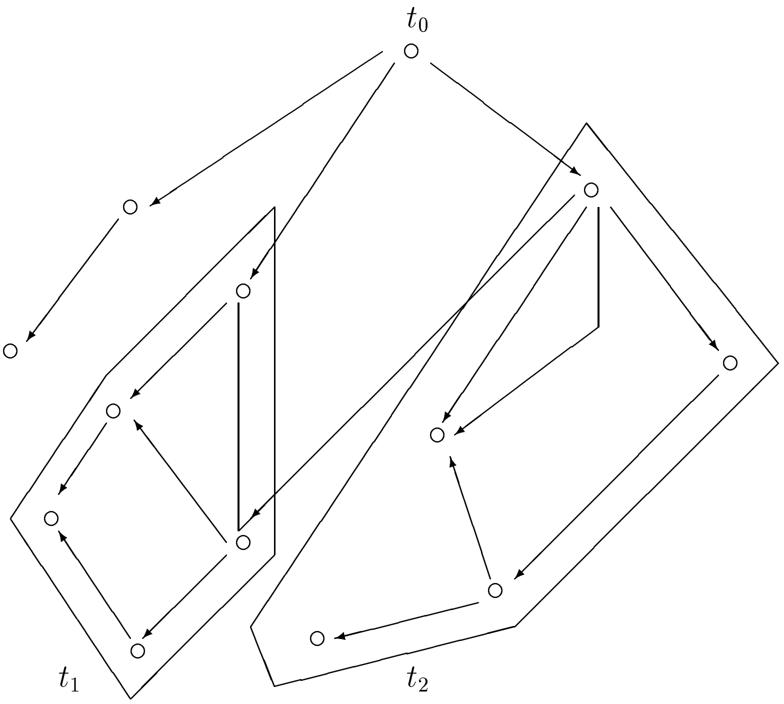


Figure 1.1: Three rooted directed acyclic graphs.

Convention We will graphically represent labelled dags by drawing arrows for the edges and by marking them with (names of) their labels, if existing. In labelled dags the nodes are normally not explicitly depicted, instead we display (names of) their labels. If ordered dags are displayed, we shall assume the order to be from left to right.

1.3.2 Symbol Dags

Logical expressions can be represented with dags by labelling their *vertices* with symbols.

Definition 1.3.7 (Symbol dag) (inductive)

1. Any labelled ordered dag $T = \langle t, \lambda \rangle$ where t consists just of one vertex v labelled with the symbol of a symbol string s is a *symbol dag of* s.
2. Suppose Φ is an expression with the immediate subexpression sequence $\Phi_1, \ldots, \Phi_n$ and the dominating symbol s, and let $T_1, \ldots, T_n$ be consistent symbol dags of the expressions $\Phi_1, \ldots, \Phi_n$, respectively. Any labelled ordered dag obtained by forming the union of the dags $t_1, \ldots, t_n$, adding a new root vertex r, labelled with s, and adding n new edges $e_1, \ldots, e_n$ connecting r with the roots of the $t_1, \ldots, t_n$, in the respective order, is a *symbol dag of* the expression Φ.

One and the same logical expression may have symbol dags of different structures, i.e., non-isomorphic underlying dags, as illustrated in Figure 1.2.

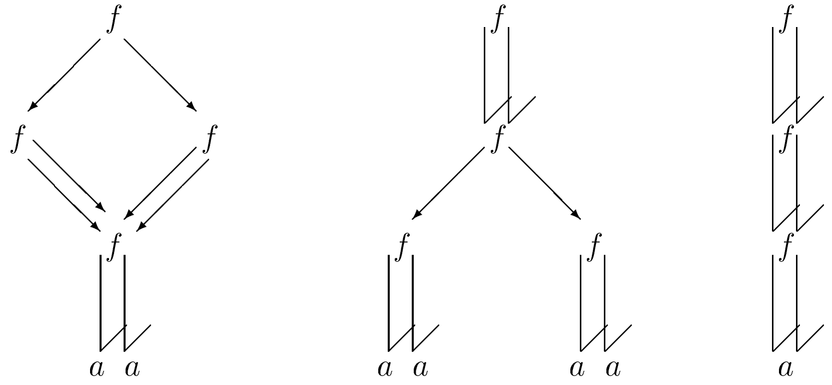


Figure 1.2: Symbol dags of a term $f(f(f(a,a),f(a,a)),f(f(a,a),f(a,a)))$.

Definition 1.3.8 ((Edge) size of a symbol dag) The *(edge) size* of a symbol dag t, $\mathrm{size}(t)$, is the number of its edges.

As will be shown in the next section, any symbol dag T can be appropriately represented as a string over a finite alphabet such that the realistic size $\#(T)$ is of the order $\mathrm{O}(n \log n)$ with respect to $\mathrm{size}(T)$, where n is the number of edges of T. Consequently, it is natural to take the number of edges as a representative size measure of a symbol dag; note that the number of nodes may not be representative.

Definition 1.3.9 (Minimal symbol dag) A symbol dag T of an expression Φ is called *minimal* if no symbol dag T' of Φ has a smaller edge size than T. A symbol dag T of an expression Φ is called *strongly minimal* if it is minimal and no symbol dag T' of Φ has a smaller number of nodes than T.

In strongly minimal symbol dags of an expression every subexpression is represented only once, and in minimal symbol dags every complex, i.e., non-atomic, subexpression is represented only once. Note, however, that no edge size reduction can be achieved by representing atomic expressions only once.

Proposition 1.3.1 *Any two strongly minimal symbol dags for an expression have isomorphic underlying dags.*

Proof Let $T_1 = \langle\langle\langle V_1, E_1, f_1\rangle, O_1\rangle, \lambda_1\rangle$ and $T_2 = \langle\langle\langle V_2, E_2, f_2\rangle, O_2\rangle, \lambda_2\rangle$ be strongly minimal symbol dags of a first-order expression Φ. Then, $\mathrm{card}(E_1) = \mathrm{card}(E_2)$, and $\mathrm{card}(V_1) = \mathrm{card}(V_2)$. Furthermore, due to the strong minimality, no two distinct symbol subdags of T_1 are symbol dags of one and the same expression, and also for T_2. We define two mappings α_V and α_E, as follows. First, given any vertex $v_i \in V_1$ being the root of a symbol dag T_i of a subexpression Φ_i of Φ, we set $\alpha_V(v_i) = v_i'$ where v_i' is the vertex in V_2 being the root of the symbol dag T_i' of Φ_i. Clearly, α_V: $V_1 \longrightarrow V_2$ is total, injective, and surjective. Secondly, for any vertex $v_i \in V_1$, let $V = (e_1, \dots, e_k)$ be the sequence of edges beginning in v_i, in the order induced by O_1, and suppose $V' = (e_1', \dots, e_l')$ to be the sequence of edges beginning in $\alpha_V(v_i)$, in the order induced by O_2. Since the symbol dags with roots v_i and v_i' represent the same expression, by the definition of symbol dags, $k = l$. Set $\alpha_E(e_j) = e_j'$, for any $1 \leq j \leq k$. Clearly, α_E: $E_1 \longrightarrow E_2$ is total, injective, and surjective, too; furthermore, for any edge $e \in E_1$ with $f_1(e) = \langle v_1, v_2 \rangle$: $f_2(\alpha_E(e)) = \langle \alpha_V(v_1), \alpha_V(v_2) \rangle$. □

Accordingly, we can speak of *the* strongly minimal symbol dag of an expression, which can be seen as a normal form. In Figure 1.2, the rightmost symbol dag is the strongly minimal symbol dag of the respective expression.

Note Given any symbol dag t of an expression Φ, it can be normalized, i.e., transformed into the strongly minimal symbol dag of Φ, with linear cost with respect to the size of t; the transformation works in a bottom-up manner level by level (starting at the leaves) by identifying lists of edges pointing to the same vertices. Consequently, in principle, one could always work with strongly minimal symbol dags (but consider the remarks at the end of the next section).

As a very useful specialization of rooted dags we introduce the concept of trees.

Definition 1.3.10 (Tree) A *tree* is a rooted dag in which no vertex has more than one predecessor. The *depth* of a vertex in a tree is the number of nodes dominating N.

Convention Subtrees are defined in analogy to subdags. Trees will normally be displayed with roots upward, and since the direction of edges in trees is always assumed downward, we shall often omit the arrow heads.

Definition 1.3.11 (Symbol tree) If the symbol dag of an expression Φ consists of a tree, then it is called a *symbol tree of* Φ.

As in the dag notation, in the tree representation there is no need for punctuation symbols. But in contrast to symbol dags, *all* symbol trees of an expression are isomorphic—they may differ in their index sets only. Therefore, we will speak of *the* symbol tree of an expression. In Figure 1.3 the symbol tree of the expression from Figure 1.2 is displayed. From the viewpoint of space complexity it is important that the string and the tree representation of logical expressions are *polynomially related* representation schemes.

Proposition 1.3.2 *There are constants* c_1, c_2 *such that for the symbol tree* t *of any expression* Φ*:* $\text{size}(t) < c_1(\text{length}(\Phi))$ *and* $\text{length}(\Phi) < c_2(\text{size}(t))$.

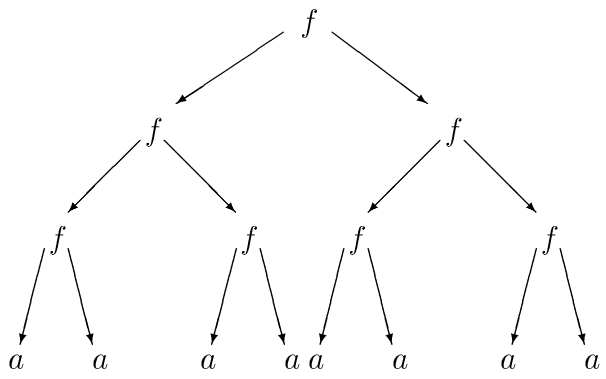


Figure 1.3: Symbol tree of a term $f(f(f(a,a),f(a,a)),f(f(a,a),f(a,a)))$.

Apparently, the second of these two facts does not hold for the dag representation of logical expressions, even if we replace 'constant' by 'polynomial'.

Proposition 1.3.3 *For every polynomial p there is a symbol dag t of an expression Φ such that* $\mathrm{length}(\Phi) > p\,(\mathrm{size}(t))$.

Proof Immediate from Figures 1.2 and 1.3. □

Consequently, the dag format permits a more compact representation of logical expressions, and hence a considerable extension of the power and applicability of logic.

Note Unfortunately, there are two reasons for the fact that the dag representation of logical expressions is not really used in logical practice. The first reason is a conceptual one. It is based on the misunderstanding that the question of how logical expressions are to be represented ought not concern the designer of logical languages and calculi, but belong to the task of *implementing* logical systems in an optimal way. Since implementations contain many irrelevant details, such a position impedes the study of essential complexities of logical systems. The other reason for the fact that the dag notation is not used by logicians is simply that it is very uncomfortable and complicated to draw graphs and to communicate and process graphical information textually[6], so that the graph representation is not sufficiently supported.

1.4 The Language of Definitional Expressions

In general, there are two different principal approaches of solving the representation problem of logical expressions. On the one hand, one can leave logical expressions suboptimal with respect to compactness, and put an additional layer on top of logical expressions, where more compact representations like graphs are

[6] Although directed graphs can be implemented very efficiently on a computer.

at hand. The advantage is that different representations for one and the same logical expression are handled on the meta-level, so that equality of logical expressions on the object level remains string identity. But this may not help in practice, for complexity assessments will then be made for the representations and not for the object level, thus making the object level superfluous. Consequently, we shall pursue the other possibility and generalize the object level itself.

1.4.1 Definitional Expressions

Since something as powerful as the dag representation seems necessary, but two-dimensional information is too hard to handle textually, the natural approach is to look for a compact one-dimensional (i.e., string) encoding of dags which is convenient for the logician. Customary encodings of graphs on computers are *adjacency matrices* or *adjacency lists* (see [van Leeuven, 1990]), which should be used when it comes to representing symbol dags on a computer. Unfortunately, both representations are also two-dimensional, and their string variants are too overloaded to be appealing to the human. Instead we shall design a string variant of the dag notation which facilitates to formulate logical expressions in both a compact and a convenient manner. The basic idea is that the power of symbol dags comes from their ability to *abbreviate* expressions. This can also be achieved on the string level by extending the ordinary logical language with the possibility of using abbreviations or *definitional expressions.*

First, the alphabet of the logical language is extended.

Definition 1.4.1 (Definitional alphabet and signature) Suppose $\Sigma = \langle \mathcal{A}, a \rangle$ is a first-order signature. Let $\mathcal{D}_T$ and $\mathcal{D}_F$ be two countably infinite sets of symbols, called *term definition symbols* and *formula definition symbols*, respectively, such that all three sets are pairwise disjoint. $\mathcal{D} = \mathcal{A} \cup \mathcal{D}_T \cup \mathcal{D}_F$ is called a *definitional (first-order) alphabet*, and $\Sigma_D = \langle \mathcal{D}, a \rangle$ is said to be a *definitional (first-order) signature.*

Notation We shall use lower-case gothic letters for denoting definition symbols.

Let, in the sequel, Σ_D be a definitional first-order signature.

Definition 1.4.2 (Potential definitional expression) (by simultaneous induction)

1. Every (ordinary) logical term, formula, and expression according to the Definitions 1.2.2 to 1.2.6 is a *potential definitional term, formula*, and *expression*, respectively.

2. If D is a potential definitional term or formula and $\mathfrak{d}$ is a term or formula definition symbol, respectively, then the concatenation $\mathfrak{d}D$—we prefer to write it by left-indexing ${}_{\mathfrak{d}}D$—is a *potential definitional term* or *formula*, respectively. The string ${}_{\mathfrak{d}}D$ is called a *term* or *formula definition*, respectively; $\mathfrak{d}$ is named its *definiendum* and D its *definiens.*

3. If D is a potential definitional expression and D' is a subterm[7] or subformula of D, then any string obtained by replacing an occurrence of D' in D with a term definition or term definition symbol string or with a formula definition or formula definition symbol string, respectively, is a *potential definitional expression.*

Definition 1.4.3 (Definiens of a definition symbol) Let D be a potential definitional expression. If a definition symbol $\mathfrak{d}$ is the definiendum of a definition ${}_{\mathfrak{d}}D'$ occurring in D, then we call D' a *definiens* of $\mathfrak{d}$ *in* D; we shall also say that $\mathfrak{d}$ is *defined by* D'.

Definition 1.4.4 (Definition dependency) Let D be a potential definitional expression. A definition symbol $\mathfrak{d}_2$ is said to *immediately depend on* a definition symbol $\mathfrak{d}_1$ *in* D, $\mathfrak{d}_1 \prec_d \mathfrak{d}_2$, if $\mathfrak{d}_1$ occurs in a definiens of $\mathfrak{d}_2$ in D. The *definition dependency* relation *on* D is the transitive closure $\prec_d^+$ of the immediate dependency relation $\prec_d$ on the definition symbols in D.

Definition 1.4.5 (Well-defined or definitional expression) A potential definitional expression D is called *well-defined* or a *definitional expression* in case

1. any definition symbol $\mathfrak{d}$ in D is defined exactly once in D, i.e., $\mathfrak{d}$ occurs exactly once as a definiendum in D, and
2. the definition dependency relation $\prec_d^+$ on D is well-founded[8].

Proposition 1.4.1 (Well-definedness of an expression) *For any potential definitional expression S, it can be checked with linear cost with respect to the input size whether S is well-defined.*

Proof The linear complexity of checking the first condition is apparent. The second condition can be examined in the same way as the cycle-freeness of a directed graph, which can be done in linear time. □

Definition 1.4.6 (Definitional language) The *definitional (first-order) language* $\mathcal{L}_\Sigma$ *over* Σ is the set of definitional expressions over Σ.

Example 1.4.1 (Definitional term) The following string denotes a definitional term:

$$f({}_{\mathfrak{f}_1}f({}_{\mathfrak{f}_2}f({}_{\mathfrak{a}}a, \mathfrak{a}), \mathfrak{f}_2), \mathfrak{f}_1)$$

[7] Subexpressions of potential definitional expressions are defined in analogy to the manner ordinary subexpressions were introduced in Subsection 1.2.2, with the only extension that certain complex expressions—the definitions—do not possess dominating symbols.

[8] A binary relation $\prec$ is *well-founded* if every nonempty subset M of the field of $\prec$ contains a minimal element with respect to $\prec$, i.e., there is an element $m \in M$ such that for no $m' \in M$: $m' \prec m$ (see, for instance, [Krivine, 1971]).

The ordinary expression *represented* by a definitional expression can be defined as the *expansion* of the definitional expression.

Definition 1.4.7 (Expansion) (inductive)

1. The *expansion* of any (ordinary) expression D is D.
2. If D is a definitional expression but no (ordinary) expression, and if D' is a definitional expression obtained from D by replacing a single definition ${}_{\mathfrak{d}}D''$ occurring in D and every other occurrence of $\mathfrak{d}$ in D by its definiens D'', then any expansion of D' is an *expansion* of D.

Evidently, expansions are properly defined and all expansions of a definitional expression are identical and an (ordinary) logical expression. The expansion of Example 1.4.1 is the term denoted by the symbol dags of Figures 1.2 and 1.3.

Convention As the declarative meaning of a definitional expression we take simply the meaning of the expansion of the expression.

Just as in the case of symbol dags, different definitional expressions may have one and the same expansion, which can be viewed as their normal form. The significant advantage of the definitional language is that it provides a compact logical notation for expressions without having to rely on two-dimensional notation. It should be mentioned that definitional expressions differ from ordinary logical expressions in certain respects. First, the well-formedness of a definitional expression D is no longer a local condition which is automatically inherited from the subexpressions of D, like for ordinary expressions; instead, the well-formedness can only be determined globally. This renders the composition and decomposition of definitional expressions more difficult, though tractable. Furthermore, the equality of definitional expressions does not remain string identity but becomes string identity of the expansions of the definitional expressions, as discussed below.

Note The presented format of definitional expressions is but one possibility of modelling dag structures with strings. An alternative related framework would be to work with pairs $\langle D, \mathfrak{D} \rangle$ consisting of a potential definitional expression D, in the sense above but *without* definitions, and a collection of definitions $\mathfrak{D}$, this way keeping the definitions alongside the expressions. Although, conceptually, such an approach may be more elegant, it has the big disadvantage that the expression part D might degenerate to a single definition symbol, with the consequence that the entire structure information would have to be expressed in the definition part. Therefore, from the point of view of readability, definitional expressions seem to be more convenient.

1.4.2 Definitional Expressions vs Symbol Dags

It is interesting to investigate the correspondence between symbol dags and definitional expressions. A transformation from symbol dags to definitional expressions might work as follows.

Procedure 1.4.1 [(From symbol dags to definitional expressions) Let t be a symbol dag of an expression D. Annotate every vertex v_i of the dag with a distinct definition symbol $\mathfrak{d}_i$ of the appropriate type. Let $\mathfrak{d}_1$ be the annotation of the root vertex v_1. Starting with the unary string $(\mathfrak{d}_1)$, iteratively perform the following operation.

> Select a definition subsymbol $o = {}^{i}(\mathfrak{d})$ where $\mathfrak{d}$ is not yet defined in the string and annotates a vertex v labelled with a symbol s. If v is a leaf vertex, replace the subsymbol o with the definition ${}_{\mathfrak{d}}s$. If v is no leaf vertex, suppose $(\mathfrak{d}_1, \ldots, \mathfrak{d}_n)$ is the sequence of definition symbols annotating the vertices succeeding v, in the order and multiplicity of the edges starting at v; let D' be the potential definitional expression which is the concatenation determined by the symbol that labels v as dominating symbol and the immediate subexpression sequence $(\mathfrak{d}_1, \ldots, \mathfrak{d}_n)$; replace the subsymbol o with the definition ${}_{\mathfrak{d}}D'$.

The resulting string is a definitional expression and represents the expression D. It is evident that the length of the output string of this procedure is polynomially (i.e., linearly) related with the size of the input dag t.

Definition 1.4.8 (Strict dag expression) Any string obtained from a symbol dag by Procedure 1.4.1 is called a *strict dag expression*.

Because of this correspondence between symbol dags and definitional expressions, any manipulations on symbol dags can be directly performed on the string level of the corresponding definitional expressions.[9] But the correspondence between symbol dags and definitional expressions is not one to one, not even if the difference in definition symbols is disregarded and only non-isomorphy with respect to consistent renaming of definition symbols is considered. This is because the definitional framework is slightly more general and permits the formulation of strings which are not strict dag expressions. On the one hand, a definitional expression may contain non-atomic substrings composed only of definition symbols, and on the other hand, not every non-definition subexpression D in a definitional expression need be abbreviated, i.e., D need not be the definiendum of a definition. This is the case for Example 1.4.1 where the expression itself is not abbreviated. Both phenomena cannot occur in strict dag expressions.

[9] In fact, in this framework also general directed graphs can be encoded, namely, by dropping the well-foundedness condition in Definition 1.4.5. Such a generalization would permit to express what on the level of non-logical expressions are called *infinite terms* (see [Colmerauer, 1982] or [Courcelle, 1983]), which is not our concern in this work.

Yet, one can simply associate a symbol dag with any definitional expression by transforming it into a strict dag expression. To be most general, we present this operation for potential definitional expressions.

Definition 1.4.9 (Potential strict dag expression) Any string D' obtainable from a potential definitional expression D by Procedure 1.4.2 is called a *potential strict dag expression.* D' is said to *correspond* to D.

Procedure 1.4.2 (From potential definitional expressions to potential strict dag expressions) Let D be a potential definitional expression. Abbreviate any non-abbreviated occurrence of any subexpression in D which is neither a definition nor a definition symbol with a new distinct definition symbol of the appropriate type. Afterwards, iteratively, replace any non-atomic substring ${}^{i}(\mathfrak{d}_1, \ldots, \mathfrak{d}_{n-1}, \mathfrak{d}_n)$ of definition symbols with its rightmost definition symbol $\mathfrak{d}_n$, and substitute any other occurrences of the other definition symbols $\mathfrak{d}_1, \ldots, \mathfrak{d}_{n-1}$ in the string by $\mathfrak{d}_n$, too.

The output of this procedure is a potential definitional expression. Apparently, if D is a definitional expression, then its corresponding potential strict dag expression is a strict dag expression. Furthermore, the input and the output have the same expansions, and the input length is polynomially related with the output length. With this intermediate operation, the definitional expression from Example 1.4.1 can be viewed as corresponding to the rightmost (i.e., the minimal) dag depicted in Figure 1.2. Granted this transformation, the leftmost and the middle dag of Figure 1.2 can be seen as encoded, for example, by the definitional expressions $f(f({}_{\mathfrak{f}_1}f(a,a), \mathfrak{f}_1), f(\mathfrak{f}_1, \mathfrak{f}_1))$ and $f({}_{\mathfrak{f}_1}f(f(a,a), f(a,a)), \mathfrak{f}_1)$, respectively.

A slight generalization of strict dag expressions turns out to be a central notion for technical purposes.

Definition 1.4.10 ((Potential) dag expression) A (potential) definitional expressions D is called a *(potential) dag expression* if each occurrence of a subexpression that is not a definition is abbreviated, i.e., immediately preceded by a definition symbol.

Apparently, any (potential) strict dag expression is a (potential) dag expression, while the converse does not hold, since in (potential) dag expressions non-atomic substrings of definition symbols may occur. We have introduced this weaker notion because it is the optimal framework for all kinds of modification operations, like identification, matching, or unification. Yet, from the point of view of readability, it is more convenient not to abbreviate *every* subexpression in a string that is no definition, but only those which correspond to subdags with more than one ingoing edge. Consequently, we shall work with arbitrary definitional expressions, and whenever a modification has to be performed, we shall transform them into (strict) dag expressions.

Definition 1.4.11 (Minimal definitional expression) A definitional expression D is called *minimal* if for any definitional expression D' with the same expansion:

$$\text{length}(D') \geq \text{length}(D).$$

It is apparent that minimal definitional expressions satisfy the following properties.

Proposition 1.4.2 (Structure of minimal definitional expressions)

Let D be a minimal definitional expression and D' an arbitrary subexpression of D.

(a) If $\text{length}(D') > 1$, *then there is exactly one occurrence of D' in D.*

(b) If $\text{length}(D') = 1$ *(D' is atomic), then D does not contain D' as a definiens.*

Note that the dag expression ${}_{\mathfrak{f}_1}f({}_{\mathfrak{f}_2}f({}_{\mathfrak{f}_3}f({}_{\mathfrak{a}}a, \mathfrak{a}), \mathfrak{f}_3), \mathfrak{f}_2)$ encoding the minimal symbol dag on the righthand side of Figure 1.2 is not a minimal definitional expression, because there is a shorter one, namely, $f({}_{\mathfrak{f}_1}f({}_{\mathfrak{f}_2}f(a, a), \mathfrak{f}_2), \mathfrak{f}_1)$, which does neither abbreviate the constant a nor the entire expression. But apparently, any minimal definitional expression encodes a minimal symbol dag via Procedure 1.4.2, and any minimal definitional expression is only linearly shorter than one of its corresponding (strict) dag expressions.

In order to illustrate the differences between the variants of definitional expressions, in the following chart, for any type a significant example is displayed. All strings have the same expansion, namely, the ordinary expression given in the first line.

Chart 1.4.1 (Types of definitional expressions)

ordinary expression	$g(f(g(a, g(f(b), a))), g(a, g(f(b), a)))$
arbitrary definitional expression	$g(f(g({}_{\mathfrak{a}_1\mathfrak{a}_2}a, {}_{\mathfrak{g}}g(f(b), \mathfrak{a}_2))), g(\mathfrak{a}_2, \mathfrak{g}, a)))$
minimal definitional expression	$g(f({}_{\mathfrak{g}}g(a, g(f(b), a))), \mathfrak{g})$
dag expression	${}_{\mathfrak{g}_0}g({}_{\mathfrak{f}_0}f({}_{\mathfrak{g}_3\mathfrak{g}_2\mathfrak{g}_1}g(\mathfrak{a}, {}_{\mathfrak{g}_4}g({}_{\mathfrak{f}_1}f({}_{\mathfrak{b}}b), {}_{\mathfrak{a}}a))), \mathfrak{g}_1)$
(minimal) strict dag expression	${}_{\mathfrak{g}_0}g({}_{\mathfrak{f}_0}f({}_{\mathfrak{g}_1}g(\mathfrak{a}, {}_{\mathfrak{g}_2}g({}_{\mathfrak{f}_1}f({}_{\mathfrak{b}}b), {}_{\mathfrak{a}}a))), \mathfrak{g}_1)$

1.4.3 Identification of Definitional Expressions

Since definitional expressions with the same expansion are intended to be identified by viewing them just as different *representations* of their expansion, the interesting question arises how expensive it is to determine whether two definitional expressions have the same expansion. Fortunately, this problem can de decided in time polynomially bounded by the sizes of the definitional expressions and does not depend on the sizes of the respective expansions.

It is not necessary that the strings to be identified be definitional expressions, in the general case it is sufficient that both are potential definitional expressions with respect to a certain context. A useful notion to express this formally is the following.

Definition 1.4.12 (Well-defined sequence) A sequence $S = (D_1, \ldots, D_n)$ of potential definitional expressions is *well-defined*, if each definition symbol occurring in some member of S is defined exactly once in exactly one element D_i, $1 \leq i \leq n$, of S, and if the transitive closure of the union of the definition dependency relations on the members of S is well-founded.

According to this definition, if a potential definitional expression D_i, $1 \leq i \leq n$, is contained more than once in a well-defined sequence $S = (D_1, \ldots, D_n)$, i.e., if for each $1 \leq j \leq n$: $j \neq i$ entails $D_j \neq D_i$, then no definition can occur in D_i. Given a well-defined sequence of potential definitional expressions, we can make any of its members D_i a definitional expression, as follows. We call this operation *making D_i independent of its context S.*

Procedure 1.4.3 (Context independency operation) If $S = (D_1, \ldots, D_n)$ is a well-defined sequence of potential definitional expressions and D_i is a member of S, then, starting with the string D_i, iteratively, perform the following operation.

> As long as the current string is no definitional expression, select any occurrence of a definition symbol which is undefined in the string, and replace it with its definition in some member of S.

Lemma 1.4.3 (Context independency) *Suppose $S = (D_1, \ldots, D_n)$ is a well-defined sequence of potential definitional expressions and D_i is a member of S. If D'_i is the result obtained by making D_i independent of S, then*

$$\text{length}(D'_i) \leq \sum_{j=1}^{n} \text{length}(D_j).$$

Proof Immediate from the definition of a well-defined sequence. □

An algorithm for identifying well-defined sequences of potential definitional expressions is presented in Procedure 1.4.4. We use an informal functionally-oriented[10] language for specifying algorithms. The meaning of its instructions is self-explanatory to anyone familiar with languages like LISP [McCarthy et al., 1962]. The main characteristics of this language are the following ones. It permits the use of both local program variables and global structures, the latter destructively assignable (:=). The output of any sequence of instructions is the value of the last function call, just like in LISP, or, in case of indeterminism, *one* of the possible outputs. Global structures and predefined program components are set in bold (sans serif) font. For convenience, we also make use of the meta-function apply, which permits to compose function calls from function names and the respective argument lists; thus, apply('function', $a_1, \ldots, a_n$) is equivalent to the function call function($a_1, \ldots, a_n$).[11]

[10] Occasionally, we shall permit indeterminism to occur in operations, so that, strictly speaking, we have a relational framework.

[11] For this semi-formal language, we do not introduce the full terminology of the λ-calculus.

Procedure 1.4.4 (Identification Algorithm)

```
{ define identification($D_1, D_2$)
    input two strings $D_1, D_2$ such that $(D_1, D_2)$ is a well-defined sequence
    output boolean
    initialization of a global structure:
    a partial mapping id: $\mathcal{D}_T \cup \mathcal{D}_F \longrightarrow \mathcal{D}_T \cup \mathcal{D}_F$, initially defined by

        $\mathsf{id}(\mathfrak{d}) := \begin{cases} \mathfrak{d} & \text{for any definition symbol in } D_1 \text{ or } D_2 \\ \text{undefined} & \text{otherwise} \end{cases}$

    identify($D_1, D_2$) }
```

```
{ define identify($D'_1, D'_2$)
    input two potential definitional expressions $D'_1, D'_2$
    output boolean
    let $\langle D_1, D_2 \rangle$ = unfold($D'_1, D'_2$),
    if   $D_1$ is a symbol (string) : $D_1 = D_2$
    else if   $D_2$ is a symbol (string) : false
        else let $S_1, S_2$ be the immediate subexpression sequences
            of $D_1, D_2$ and $o_1, o_2$ their dominating symbols, resp.,
            if   $o_1 = o_2$ : sequences('identify', $S_1, S_2$)
            else false }
```

Procedure 1.4.5 (Unfolding Algorithm)

```
{ define unfold($D_1, D_2$)
    input two potential definitional expressions
    output a pair of potential definitional expressions
    if   $D_1$ is a definition ${}_{\mathfrak{d}_1}D'_1$ or a definition symbol (string) $\mathfrak{d}_1$ :
        if   $D_2$ is a definition ${}_{\mathfrak{d}_2}D'_2$ or a definition symbol (string) $\mathfrak{d}_2$ :
            if   id($\mathfrak{d}_1$) = id($\mathfrak{d}_2$): $\langle \mathfrak{d}_1, \mathfrak{d}_1 \rangle$
            else id($\mathfrak{d}_2$) := id($\mathfrak{d}_1$) : unfold(definiens($\mathfrak{d}_1$), $D_2$))
        else unfold(definiens($\mathfrak{d}_1$), $D_2$)
    else if   $D_2$ is a definition ${}_{\mathfrak{d}_2}D'_2$ or a definition symbol (string) $\mathfrak{d}_2$ :
            unfold($D_1$, definiens($\mathfrak{d}_2$))
        else $\langle D_1, D_2 \rangle$ }
```

Procedure 1.4.6 (Sequence Meta-Algorithm)

```
{ define sequences($Function\_name, S_1, S_2$)
    input a function name and two finite sequences of strings
    output boolean
    if   $S_1 = \emptyset$ and $S_2 = \emptyset$ : true
    else if   apply($Function\_name$,first($S_1$),first($S_2$)) :
            sequences($Function\_name$,rest($S_1$),rest($S_2$))
        else false }
```

Description of the Identification Algorithm (Procedure 1.4.4)
The input to the algorithm are two strings which form a well-defined sequence. First, a global mapping id is defined associating with every definition symbol occurring in the input strings a representative of the set of definition symbols which have already been identified to abbreviate the same expansions. As a matter of fact, initially, every definition symbol $\mathfrak{d}$ is the representative of the singleton set $\{\mathfrak{d}\}$.[12] The algorithm employs three functions: identify and sequences, which call each other mutually recursively, and unfold, which is a subroutine of identify. The function identify takes two potential definitional expressions and calls the auxiliary unfold procedure, which unfolds the expressions with respect to the definitions formulated in the input (lines (4),(5), and (7)), until neither of them are definitions or definition symbols; except the respective definition symbols have already been identified (line (3)), in which case no unfolding is done, instead a pair of identical definition symbols is returned. In case both expressions have to be unfolded, during the unfolding procedure it is noted that the respective definition symbols must have the same expansions, by modifying the mapping id and making equal the id-values of both definition symbols (line (4)). The output strings of unfold are processed further by identify. The procedure checks whether at least one of them is a symbol string; in this case the value of their syntactic comparison is returned (lines (2) and (3)). If both strings are non-atomic and have the same dominating symbol, the procedure sequences is called with a name of the identify function and the immediate subexpression sequences as arguments (line (6)); otherwise false is returned (line (7)). The procedure sequences either returns true, in case both sequences are empty (line (1)); or splits apart the first pair of expressions from the input sequences and calls identify (line (2)); if the result is true, sequences proceeds recursively with the rest of both sequences (line (3)), otherwise the procedure returns false (line (4)).

The working of the procedure on a concrete input is illustrated in Example 1.4.2.

Example 1.4.2 (Identification process) Given two definitional expressions $D_1 = g({}_{\mathfrak{f}_1}f(a,a), f(a,a), \mathfrak{f}_1)$, and $D_2 = g(\mathfrak{f}_2, \mathfrak{f}_3, {}_{\mathfrak{f}_2\mathfrak{f}_3}f(a,a))$, the following sequence of operations and function calls will be executed.

```
identification($D_1, D_2$)
⍈id := $\{\langle\mathfrak{f}_1,\mathfrak{f}_1\rangle, \langle\mathfrak{f}_2,\mathfrak{f}_2\rangle, \langle\mathfrak{f}_3,\mathfrak{f}_3\rangle\}$
  identify($D_1, D_2$)
  ⍈unfold($D_1, D_2$)
  ⍇$\langle D_1, D_2\rangle$
```

[12] Although, in a successful sequence of identification operations, it is not necessary to initialize the id mapping at the beginning of each identification operation, one can correctly work with the same id mapping throughout the sequence, which allows for a further increase in efficiency. This is possible for matching and unification, too.

```
⍈sequences(‘identify’,$({}_{\mathfrak{f}_1}f(a,a), f(a,a), \mathfrak{f}_1), (\mathfrak{f}_2, \mathfrak{f}_3, {}_{\mathfrak{f}_2\mathfrak{f}_3}f(a,a))$)
  ⍈identify$({}_{\mathfrak{f}_1}f(a,a), \mathfrak{f}_2)$ [if true: sequences(‘identify’,$(f(a,a), \mathfrak{f}_1), (\mathfrak{f}_3, {}_{\mathfrak{f}_2\mathfrak{f}_3}f(a,a))$)]
    ⍈unfold$({}_{\mathfrak{f}_1}f(a,a), \mathfrak{f}_2)$
      ⍈id$(\mathfrak{f}_2) := \mathfrak{f}_1$
        unfold$(f(a,a), {}_{\mathfrak{f}_3}f(a,a))$
        ⍈unfold$(f(a,a), f(a,a))$
    ⍇⍇⍇$\langle f(a,a), f(a,a)\rangle$
    ⍈sequences(‘identify’,$(a,a), (a,a)$)
      ⍈identify$(a,a)$ [if true: sequences(‘identify’,$(a), (a)$)]
      ⍇true
      ⍈sequences(‘identify’,$(a), (a)$)
        ⍈identify$(a,a)$ [if true: sequences(‘identify’,$(), ()$)]
        ⍇true
        ⍈sequences(‘identify’,$(), ()$)
  ⍇⍇⍇⍇true
  ⍈sequences(‘identify’,$(f(a,a), \mathfrak{f}_1), (\mathfrak{f}_3, {}_{\mathfrak{f}_2\mathfrak{f}_3}f(a,a))$)
    ⍈identify$(f(a,a), \mathfrak{f}_3)$ [if true: sequences(‘identify’,$(\mathfrak{f}_1), ({}_{\mathfrak{f}_2\mathfrak{f}_3}f(a,a))$)]
      ⍈unfold$(f(a,a), \mathfrak{f}_1)$
      ⍇$\langle f(a,a), f(a,a)\rangle$
      ⍈sequences(‘identify’,$(a,a), (a,a)$) $\cdots$ (see above) $\rightsquigarrow$
    ⍇⍇true
    ⍈sequences(‘identify’,$(\mathfrak{f}_1), ({}_{\mathfrak{f}_2\mathfrak{f}_3}f(a,a))$)
      ⍈identify$(\mathfrak{f}_1, {}_{\mathfrak{f}_2\mathfrak{f}_3}f(a,a))$ [if true: sequences(‘identify’,$(), ()$)]
      ⍇true (since id$(\mathfrak{f}_1)$ = id$(\mathfrak{f}_2)$)
      ⍈sequences(‘identify’,$(), ()$)
⍇⍇⍇⍇⍇true
```

The termination and the total correctness of the Identification Algorithm can be verified easily. The only non-trivial point is that identification of the id-values is performed *before* the respective strings have been proved to have the same expansion. Due to the fact that an iterated unfolding of a definition symbol $\mathfrak{d}$ can never produce the same definition symbol—this follows from the acyclicity guaranteed by the well-foundedness of the definition dependency relation (Definition 1.4.5)—line (4) may not give rise to possible incorrectness. More interesting is the question of the computational cost of the identification algorithm in the worst case.

Proposition 1.4.4 (Polynomial identification of dag expressions) *There is a polynomial p (of order $O(n^2)$) such that for any two potential dag expressions D_1, D_2 which form a well-defined sequence (D_1, D_2): if the procedure* identification *is called with both strings as input, then it terminates within $p(\text{length}(D_1) + \text{length}(D_2))$ steps.*

Proof Let D_1, D_2 be as assumed. First of all, the cost for initialization is linearly bounded. We shall prove that there are at most quadratically many function calls.

To begin with, note that, clearly, the identification of two *ordinary* expressions is linearly bounded by the input. The case of definitional expressions is more complex. Whenever two distinct definitions or definition symbols are compared by identify, then their values under id are identified by unfold, before identify proceeds with their definientia. Therefore, whenever D_1, D_2 are compared again, a pair of identical definition symbols is returned (line (3) in unfold), which leads to an immediate success in line (2) of identify, and hence induces no further function calls. Because of the strict dag format, exactly those occurrences of subexpressions in D_1 and D_2 which are neither occurrences of definitions nor occurrences of definition symbols are abbreviated. Consequently, if d is the number of distinct definition symbols in D_1 and D_2, the number of non-recursive exits from unfold which are no pairs of identical definition symbols (i.e., exits via line (8)), must be $< d^2$. This entails that the number of calls of sequences from within identify (line (6)) must be bounded by d^2. While in the case of ordinary expressions each occurrence of a subexpression sequence in the input is processed only once by the procedure sequences, in the definitional case, each *pair* of occurrences of subexpression sequences need to be processed at most once. Therefore, the number of calls of sequences is quadratically bounded by the input. The number of calls of identify is at most one more (the initial call) than the number of calls of sequences, and the maximal depth of recursive calls of unfold is bounded by the value 2, due to the strict dag format. It remains to be noted that the arising low-level cost, like examining and identification of id-values or performing definition unfolding, is computationally innocuous. □

Since any potential definitional expression can be transformed into the strict dag format with linear cost, we get the following corollary.

Corollary 1.4.5 *Any pair of potential definitional expressions which form a well-defined sequence can be identified with cost quadratically bounded by the input.*

Note From the complexity point of view, the gist of the identification algorithm—which guarantees its polynomial run time—is that the procedure remembers whenever pairs of expressions have been identified before, lines (3) and (4) of the procedure unfold. It should be noted that all polynomial unification algorithms make use of this simple technique, which therefore is nothing intrinsic to unification itself but completely independent of the unification problem.

It is important to emphasize that the (strict) dag format is necessary for making the procedure polynomial.[13] In Example 1.4.3 two classes of definitional expressions are given, which are not dag expressions, i.e., in which not every occurrence of a subexpression is abbreviated that is no occurrence of a definition or a definition symbol. For those classes of definitional expressions, the identification procedure needs exponential time.

[13] In fact, the dag format is sufficient, for *strict* dag expressions the quadratic bound can be obtained.

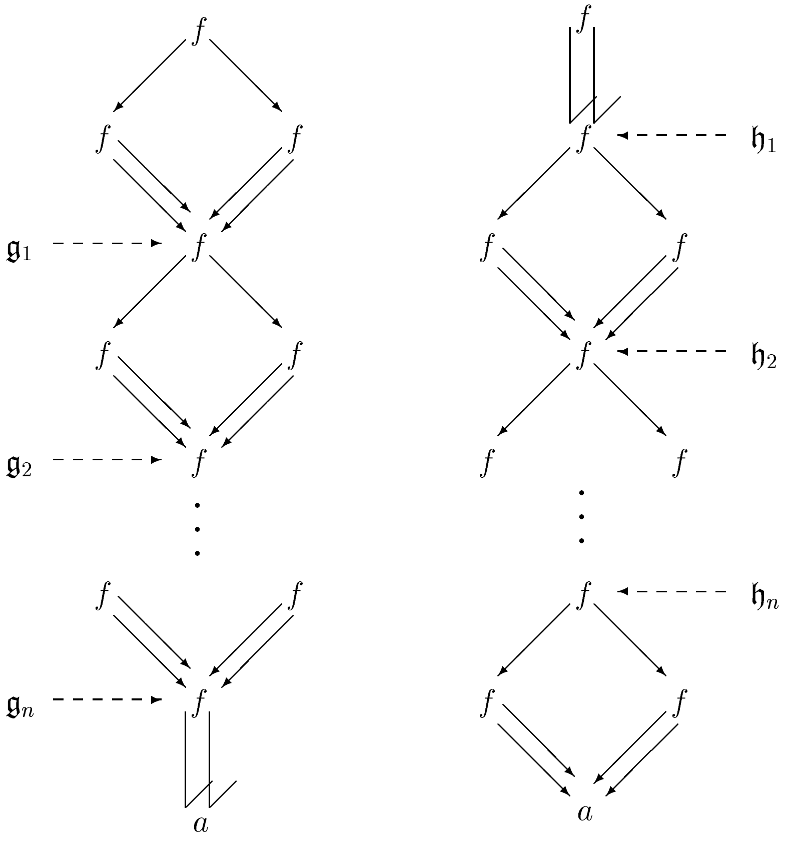


Figure 1.4: Symbol dags of the term classes from Example 1.3.2.

Example 1.4.3 (Critical expressions for identification) Consider two classes of definitional expressions with the structures

$f(f({}_{\mathfrak{g}_1}f(f({}_{\mathfrak{g}_2}f(\cdots f({}_{\mathfrak{g}_n}f({}_{\mathfrak{a}}a),\mathfrak{g}_n),f(\mathfrak{g}_n,\mathfrak{g}_n)\cdots),\mathfrak{g}_2,f(\mathfrak{g}_2,\mathfrak{g}_2)),\mathfrak{g}_1),f(\mathfrak{g}_1,\mathfrak{g}_1))$ and
$f({}_{\mathfrak{h}_1}f(f({}_{\mathfrak{h}_2}f(f(\cdots{}_{\mathfrak{h}_n}f(f({}_{\mathfrak{a}}a,\mathfrak{a}),f(\mathfrak{a},\mathfrak{a})),\mathfrak{h}_n\cdots),f(\cdots\mathfrak{h}_n,\mathfrak{h}_n\cdots)),\mathfrak{h}_2),f(\mathfrak{h}_2,\mathfrak{h}_2)),\mathfrak{h}_1).$

The dags corresponding to these expressions are depicted in Figure 1.4. The dashed arrows label the vertices in the dags which correspond to the *abbreviated* complex subexpressions.

It can be verified easily that when identifying the definitional expressions from Example 1.4.3, then the id-values always remain unchanged. This has as a consequence that the identification procedure implicitly expands both definitional expressions completely, so that the cost arising is not smaller than the cost for comparing the expansions themselves. The expansions, however, have an exponential size with respect to the input; therefore also their identification needs exponential time.

Note One might ask why we do not avoid the problem of dealing with different definitional expressions with one and the same expansion in an efficient way by exclusively working with an (up to definition renaming) unique normal form, like minimal strict dag expressions, for which less sophisticated methods may suffice. *Any* definitional expression could implicitly be transformed into this normal form. The reasons for dealing with general definitional expressions are the following two. On the one hand, it is interesting in itself to know that a minimal representation is not necessary to obtain polynomial run times for the identification (matching, and unification) procedures. On the other hand, the possibility of working polynomially with non-minimal dag expressions is very important for highly efficient unification methods using *Warren machine* technology [Warren, 1983], like [Letz et al., 1992], because the memorizing can be restricted to the *full unification* routine.

1.5 Instantiations of Logical Expressions

In the following, we shall introduce the concepts needed for describing instantiations of logical expressions, which is the most important operation performed on logical expressions. The developed notions culminate in the presentation of unification as instantiation operation. Unification marks one of the most successful advances of automated deduction, because it allows to make instantiation optimal with respect to generality. In this section, we shall work with logical expressions only, and extend the methods to the handling of definitional expressions in the subsequent section.

1.5.1 Substitutions and Matching

Let in the following denote $\mathcal{T}$ the set of terms and $\mathcal{V}$ the set of variables of a first-order language.

Definition 1.5.1 ((Variable) substitution) Let V be any finite subset of $\mathcal{V}$. A *(variable) substitution* is any mapping $\sigma : V \longrightarrow \mathcal{T}$, satisfying that for every $x \in \text{domain}(\sigma)$: $x \neq \sigma(x)$.[14]

Definition 1.5.2 (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.

Definition 1.5.3 (Instance, matching) If F is any (finite set of) expression(s) and σ is a substitution, then the *σ-instance of F*, written $F\sigma$, is the (set of) expression(s) obtained from F by simultaneously replacing every occurrence of each variable $x \in \text{domain}(\sigma)$ in F by the term $\sigma(x)$. If F and G are (finite sets of) expressions, then F is called an *instance of G* in case there is some substitution σ

[14] Alternatively, variable substitutions can be introduced as total mappings from $\mathcal{V}$ to $\mathcal{T}$ with almost all variables mapped to themselves. The advantage of that approach is that composition of substitutions becomes just functional composition, the disadvantage is that substitutions do not differ in their cardinality.

with $F = G\sigma$. We also say that G can be *matched with* F, and call σ a *matching substitution from* G *onto* F.

It is interesting to investigate the size increase caused by applying a substitution to a string. First, we need a representative size measure for substitutions. Apparently, the following will do.

Definition 1.5.4 (Size of a substitution) As the *size* of a substitution $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ we take

$$\sum_{i=1}^{n} \mathrm{length}(t_i).$$

Proposition 1.5.1 *If E is an expression and σ is a substitution, then*

$$\mathrm{length}(E\sigma) \leq \mathrm{length}(E) \times \mathrm{size}(\sigma).$$

Note This quadratic increase rate is the weak point of the standard manner of applying a variable substitution to an expression, and hence will be improved in the next section.

A standard algorithm for determining whether an expression can be matched with another is presented in Procedure 1.5.1.

Procedure 1.5.1 (Matching Algorithm)

```
{ define matching($E_1, E_2$)
    input two expressions $E_1, E_2$
    output a matching substitution from $E_1$ onto $E_2$ or false
    initialization of a global structure:
    a partial mapping $\boldsymbol{\sigma}$: $\mathcal{V} \longrightarrow \mathcal{T}$, initially empty
    if   match($E_1, E_2$) :
        let $\sigma$ be the substitution obtained from $\boldsymbol{\sigma}$ by removing all pairs $\langle x, x \rangle$,
        $\sigma$
    else false }

{ define match($E_1, E_2$)
    input two expressions $E_1, E_2$
    output boolean
    if   $E_1$ is a variable (string) $x$ and $E_2$ is a term :
        if   $\boldsymbol{\sigma}(x)$ is undefined : $\boldsymbol{\sigma}(x) := E_2$, true
        else $\boldsymbol{\sigma}(x) = E_2$
    else if   $E_1$ is a symbol (string) : $E_1 = E_2$
        else if   $E_2$ is a symbol (string) : false
            else let $S_1, S_2$ be the immediate subexpression sequences of
                $E_1, E_2$ and $o_1, o_2$ their dominating symbols, respectively,
                if   $o_1 = o_2$ : sequences('match', $S_1, S_2$)
                else false }
```

Description of the Matching Algorithm (Procedure 1.5.1)
Given two input expressions E_1 and E_2, the algorithm proceeds by incrementally generating a matching substitution, starting with the empty mapping. Whenever the procedure comes across a variable x at the first argument position, it checks whether x has already been given an instantiation, in which case it returns the truth value of the syntactic comparison of this instantiation with the other argument t; otherwise, if the variable is yet undefined, $\boldsymbol{\sigma}(x)$ is defined as t, and true is returned. The auxiliary procedure sequences (Procedure 1.4.6 on p. 26) is employed in the standard way.

The termination and the total correctness of the matching algorithm are obvious. Also it is evident that the complexity of this algorithm is linearly bounded by the input and that the procedure is deterministic, hence generates a unique matching substitution in the positive case. Furthermore, the computed substitution fulfills a certain minimality condition.

Proposition 1.5.2 *If σ is a matching substitution computed by the Matching Algorithm (Procedure 1.5.1) on two input expressions E_1, E_2, then σ is a subset of any matching substitution from E_1 onto E_2.*

Let us study the size[15] of the substitution resulting from a matching operation.

Proposition 1.5.3 *If σ is a substitution resulting from the successful matching of an expression E_1 with an expression E_2 according to the Matching Algorithm (Procedure 1.5.1), then*

$$\text{size}(\sigma) \leq \text{length}(E_2).$$

Definition 1.5.5 (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 $\sigma' \cup \tau'$, denoted by $\sigma\tau$, is called the *composition* of σ and τ.

Proposition 1.5.4 *Let σ, τ and θ be substitutions.*

(i) $\sigma\emptyset = \emptyset\sigma = \sigma$, *for the empty substitution* $\emptyset$.

(ii) If for all (finite sets of) expressions $F : F\sigma = F\tau$, *then* $\sigma = \tau$.

(iii) $(F\sigma)\tau = F(\sigma\tau)$, *for all (finite sets of) expressions* F.

(iv) $(\sigma\tau)\theta = \sigma(\tau\theta)$.

[15] Recall that the size of a substitution is defined as the sum of the lengths of the terms in the range of the substitution.

Proof (i) is immediate.
For (ii) assume, by contraposition, that $\sigma \neq \tau$. Then, without restriction of generality, there is a binding $x/t \in \sigma$ such that $x/t \notin \tau$. To demonstrate the existence of a (finite set of) expression(s) F for which $F\sigma \neq F\tau$, set $F = x$.
For the proof of (iii) let x be a variable in F. There are three cases. If $x \notin \text{domain}(\sigma) \cup \text{domain}(\tau)$, then $(x\sigma)\tau = x = x(\sigma\tau)$. If $x \in \text{domain}(\sigma)$, then $(x\sigma)\tau = x(\sigma\tau)$. If, lastly, $x \notin \text{domain}(\sigma)$ but $x \in \text{domain}(\tau)$, then $(x\sigma)\tau = x\tau = x(\sigma\tau)$. Since x was arbitrary and only variables are replaced in F, we have the result for F.
For (iv) let F be any (finite set of) expression(s). Then a repeated application of (iii) yields that $F((\sigma\tau)\theta) = (F(\sigma\tau))\theta = ((F\sigma)\tau)\theta = (F\sigma)(\tau\theta) = F(\sigma(\tau\theta))$, and, by using (ii), the proof is accomplished. □

Summarizing these results, we have that $\emptyset$ acts as a left and right identity for composition, (ii) expresses that a difference in substitutions involves a difference for some instances, by (iii) substitution application and composition permute, and (iv), the associativity of substitution composition, permits to omit parentheses when writing a composition of substitutions.

Definition 1.5.6 (Renaming substitution) Let F be a (finite set of) expression(s) and let V_F denote the set of variables occurring in F. A substitution σ is called a *renaming substitution* for F in case

1. σ is injective,
2. $\text{range}(\sigma) \subset \mathcal{V}$, and
3. $(V_F \setminus \text{domain}(\sigma)) \cap \text{range}(\sigma) = \emptyset$.

Definition 1.5.7 (Variant) Let F and G be (finite sets of) expressions. F and G are called *variants* of each other if there are substitutions σ and τ satisfying that $G = F\sigma$ and $F = G\tau$.

Proposition 1.5.5 *Let F and G be (finite sets of) expressions which are variants of each other. Then there are renaming substitutions σ for F and τ for G with $G = F\sigma$ and $F = G\tau$.*

Proof By assumption, there exist substitutions σ' and τ' with $G = F\sigma'$ and $F = G\tau'$. Let V_F and V_G be the sets of variables occurring in F and in G respectively. Then set $\sigma = \sigma' \restriction V_F$ and $\tau = \tau' \restriction V_G$.[16] We will show that σ and τ are such renaming substitutions. First, apparently, $G = F\sigma$ and $F = G\tau$. Since $F = F\sigma\tau$ and $G = G\tau\sigma$, both range(σ) and range(τ) must be subsets of $\mathcal{V}$. Consider any $x, y \in \text{domain}(\sigma)$ with $x \neq y$. Then $x\sigma\tau = x \neq y = y\sigma\tau$. Since τ is a mapping we have that $x\sigma \neq y\sigma$, which settles the injectivity of σ. By analogy, τ can be proved injective. Let, lastly, $x \in V_F \setminus \text{domain}(\sigma)$ and $z \in \text{range}(\sigma)$. Then

[16] With $f \restriction S$ we denote the *restriction* of a mapping f to a set S, i.e., $\{\langle e_1, e_2\rangle \in f \mid e_1 \in S\}$.

there exists $y \in \text{domain}(\sigma)$ with $z = y\sigma$ and $z \neq y$. Since $y\sigma\tau = y$, we get $z \neq z\tau$. On the other hand, $x \notin \text{domain}(\sigma)$, which yields $x \neq y$ and $x = x\sigma$. Noting that $x = x\sigma\tau$ yields $x = x\tau$. Therefore, $x \neq z$. Analogously, $V_G \setminus \text{domain}(\tau)$ and $\text{range}(\tau)$ can be shown disjoint, which completes the proof. □

For any given pair of substitutions, it is of fundamental importance whether one can be obtained from the other by composition.

Definition 1.5.8 (More general substitution) If σ and τ are substitutions and there is a substitutions θ such that $\tau = \sigma\theta$, then we say that σ is *more general* than τ.

1.5.2 Unification

We are mainly interested in substitutions which, when applied to a certain finite set of expressions, render all these expressions equal.

Definition 1.5.9 (Unifier) If S is a finite set of expressions and σ is a substitution such that $S\sigma$ is a singleton set, then σ is a *unifier* for S. If a unifier exists for a finite set of expressions S, then S is called *unifiable*.

The general notion of a unifier can be subclassified in certain useful ways.

Definition 1.5.10 (Restricted unifier) A unifier σ for a finite set of expressions S is called *restricted to S* if every variable in $\text{domain}(\sigma)$ occurs in S. If σ is a unifier for a finite set of expressions S and V_S is the set of variables occurring in S, then $\sigma \upharpoonright V_S$ is called the *restriction of σ to S*.

Definition 1.5.11 (Most general unifier) A unifier for a finite set of expressions S is called a *most general unifier*, *mgu*, if σ is more general than any unifier for the set S.

Most general unifiers have the nice property that any unifier for a set of expressions can be generated from a most general unifier by further composition. This qualifies mgu's as a useful instantiation vehicle in many inference systems.

Definition 1.5.12 (Minimal unifier) If a unifier σ for a finite set of expressions S has the property that for every unifier τ for S: $\text{card}(\sigma) \leq \text{card}(\tau)$, then we say that σ is a *minimal unifier for S*.

For a minimal unifier the number of substituted variables is minimal. It is apparent that for any finite unifiable set of expressions a minimal unifier always exists and that any minimal unifier for a finite set of expressions S is restricted to S, as opposed to most general unifiers, whose existence is not so obvious and which are not restricted to S. Also, any finite set of expressions has only finitely many minimal unifiers, again in contrast to most general unifiers. Another immediate consequence of the Unification Theorem (Theorem 1.5.12) demonstrated below is the following proposition.

Proposition 1.5.6 *Every minimal unifier is a most general unifier.*

For this reason, not most general unifiers but the more restrictive notion of minimal unifiers is the optimal tool for proof-theoretic and practical purposes, which has not been sufficiently recognized so far. The crucial question is, how, given a finite set of expressions, a minimal unifier can be obtained. To these ends, some additional terminology is needed.

Definition 1.5.13 (Address) The *address* of a node or subtree in an ordered tree is a sequence of positive integers defined inductively as follows.

1. The *address* of the root node (the tree itself) is the empty sequence $() = \emptyset$.
2. The *address* of the i-th successor node (subtree) of a node (subtree) with address $(k_1, \ldots, k_n)$, $n \geq 0$, is $(k_1, \ldots, k_n, i)$.

Definition 1.5.14 (Disagreement set) Let S be a finite non-empty set of expressions $E_1, \ldots, E_n$, and $T_1, \ldots, T_n$ their symbol trees, respectively. If S is a singleton set, i.e., $n = 1$, then the only *disagreement set* of S is the empty set. If S is no singleton set, then there exists an address $(k_1, \ldots, k_m)$, $m \geq 0$, such that among the nodes $N_1, \ldots, N_n$ with this address in the symbol trees $T_1, \ldots, T_n$ some are labelled with different symbols, and all i-th ancestors, $0 \leq i < m$, of the nodes in all symbol trees are labelled with the same symbols E_i, respectively. Any set S' of expressions $E'_1, \ldots, E'_n$ represented by the symbol subtrees with such an address is a *disagreement set* of S.

Example 1.5.1
A set of atoms of the structure $\{P(f(x), y),\ P(f(g(a)), x),\ P(f(y), g(z))\}$ has the two disagreement sets $\{x, g(a), y\}$ and $\{y, x, g(z)\}$.

Proposition 1.5.7 *Let σ be a unifier for a finite set of expressions S, and D_S a disagreement set of S.*

(i) σ unifies D_S.

(ii) Each member of D_S is a term.

(iii) If D_S is non-empty, then it contains a variable x with $x \neq x\sigma$.

(iv) D_S contains no pair of a variable x and a distinct term t such that x occurs in t.

Proof (i) and (ii) are obvious. For (iii), note that whenever D_S is non-empty, its cardinality must be > 1. Furthermore, D_S must contain variables, since otherwise, due to (i) and (ii)), all terms contained in D_S would have the same top-level function symbol, which would contradict the cardinality assumption or the definition of disagreement sets. Then, by (i) and the cardinality condition, one of the variables must be in the domain of σ, which proves (iii). Finally, it is clear that the existence of a non-proper binding x/t contradicts (i), hence (iv). □

Operationally, the examination whether a binding is proper is called the *occurs-check*. An extremely useful property for theoretical purposes is the following lemma.

Lemma 1.5.8 (Decomposition Lemma) *Let σ be a unifier for a finite set of expressions S with $\text{card}(S) > 1$, and let x/t be any binding composed from a disagreement set of S such that $x \neq x\sigma$ (which exists by Proposition 1.5.7(iv)). Let $\tau = \sigma \setminus \{x/x\sigma\}$. Then $\{x/t\}\tau = \sigma$.*

Proof First, since σ unifies any disagreement set of S, $x\sigma = t\sigma$. By Proposition 1.5.7(iv), 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 reflects 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 containing x and t, provided that x does not occur in t.

Definition 1.5.15 (Variable elimination and introduction) 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 *obtained from* S by a *variable elimination wrt* x/t. Conversely, we say that S can be *obtained from* $S\{x/t\}$ by a *variable introduction wrt* x/t.

Proposition 1.5.9 *Let S be any finite set of expressions and let V_S be the set of variables occurring in S.*

(i) If S is unifiable, so are all sets obtainable from S by a variable introduction or a variable elimination.

(ii) Only finitely many sets can be obtained from S by a variable elimination.

(iii) 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 $\text{card}(S') \leq \text{card}(S)$ and $V_{S'} = V_S \setminus \{x\}$.

(iv) The transitive closure of the relation

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

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

Proof For the proof of (i), note that the result for variable introductions follows immediately from their definition; for the case of variable eliminations, 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 1.5.8), we have $\{x/t\}\tau = \sigma$. Therefore, $S(\{x/t\}\tau) = (S\{x/t\})\tau = S'\tau$. Hence, τ unifies S'.
For (ii) 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 (iii), 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, $\text{card}(S') \leq \text{card}(S)$, and, since x/t is proper and t already occurs in S, we get $V_{S'} = V_S \setminus \{x\}$.
Lastly, (iv) is an obvious consequence of (iii). □

Now we turn to the computationally interesting notion of a *computed unifier* for a finite set of expressions, which is defined by simultaneous induction on the collections of all restricted unifiers and all finite sets of expressions, whereby the induction runs over the cardinality of the unifier.

Definition 1.5.16 (Computed unifier) (by simultaneous induction)

1. $\emptyset$ is a *computed unifier* for any singleton set of expressions.
2. If a substitution τ of cardinality n is a computed unifier for a finite set of expressions S' and S is a variable introduction of S' by some binding x/t, then the substitution $\sigma = \{x/t\}\tau$, which is of cardinality $n+1$, is a *computed unifier* for S.

Note That the notion of a computed unifier is indeed properly defined can be recognized as follows. By Proposition 1.5.9 (iii), the variable x does neither occur in the term t nor in the set S' nor in the expressions contained in the substitution τ, since by assumption τ is restricted to S'. Consequently, by the definition of substitution composition, $\sigma = \{x/t\}\tau = \{x/t\tau\} \cup \tau$. Therefore, $\text{card}(\sigma) = \text{card}(\tau) + 1$ and σ is restricted to S.

While the concepts of minimal and most general unifiers are mathematically comfortable, computed unifiers are computationally useful. On the one hand, if we read the inductive definition in a forward manner, it allows for the generation of pairs $\langle S, \sigma \rangle$ such that σ is a unifier for S. On the other hand, if we employ the definition in a backward manner, it specifies an algorithm for *really computing* a unifier for a given set of expressions. In Procedure 1.5.2 this algorithm is introduced in a more procedurally oriented fashion, which is a generalization of the procedure given by Robinson in [Robinson, 1965a].[17]

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

Procedure 1.5.2 (Set Unification Algorithm)

```
{ define unification($S$)
    input a finite set of expressions $S$
    output a unifier for $S$ or false
    initialization of a global structure : a substitution $\boldsymbol{\sigma}$, initially empty
    if   unify($S$) : $\boldsymbol{\sigma}$
    else false }

{ define unify($S$)
    input a finite set of expressions $S$
    output boolean
    if   $S$ is a singleton set : true
    else select a disagreement set $D$ of $S$
        if   $D$ contains a proper binding : choose one, say $x/t$,
            $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x/t\}$, unify($S\boldsymbol{\sigma}$)
        else false }
```

Note that the unification algorithm presented in Procedure 1.5.2 is a nondeterministic procedure. This is because there may be several different choices for a disagreement set and a binding. Apparently, 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 1.5.9 (i) and (iv) that each computed unifier is indeed a unifier and that the procedure terminates, respectively.

Another important property of computed unifiers is the following one.

Lemma 1.5.10 *If σ is a computed unifier for a finite set of expressions S, then no variable in* domain(σ) *occurs in the terms of* range(σ).

Proof The proof is by induction on the cardinalities of the computed unifiers. The induction base is evident: the computed unifier $\emptyset$ of any singleton set of expressions meets the disjointness property. For the induction step, assume the demanded property to hold for any computed unifier of cardinality n. Let σ be any computed unifier of cardinality $n+1$ $(n \geq 0)$ for a finite set of expressions S. By definition, S can be obtained from a set $S' = S\{x/t\}$ by a variable introduction wrt a proper binding $\{x/t\}$, and $\sigma = \{x/t\}\tau$ where τ is a computed unifier for S' with card$(\tau) = n$. As already noted, $\sigma = \{x/t\}\tau = \{x/t\tau\} \cup \tau$ and the variable x does neither occur in τ nor in t. Since, by the induction assumption, τ fulfills the disjointness property, the property is passed on to σ. □

By the definition of the composition of substitutions, from this lemma we get as an immediate corollary the idempotence of computed unifiers.

Corollary 1.5.11 *If σ is a computed unifier for a finite set of expressions S, then $\sigma = \sigma\sigma$.*

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 1.5.12 (Unification Theorem) *Let S be any unifiable finite set of expressions.*

(i) If σ is a minimal unifier for S, then σ is a computed unifier for S.

(ii) If σ is a computed unifier for S, then σ is a minimal unifier for S.

(iii) If σ is a computed unifier for S, then σ is an mgu for S.

Proof We will prove (i) to (iii) 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 expressions, and that $\emptyset$ is an mgu. 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 $> n$ ($n \geq 0$). 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 1.5.7(iv)). Let $S' = S\{x/t\}$ and set $\tau = \sigma \setminus \{x/x\sigma\}$, which is a unifier for S', by the Decomposition Lemma (Lemma 1.5.8).
For the proof of (i), 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. Since θ' is restricted to S', the Decomposition Lemma can be applied yielding that $\theta' = \theta \setminus \{x/x\theta\}$. And, from the chain $\text{card}(\theta') = \text{card}(\theta) - 1 \geq \text{card}(\sigma) - 1 = \text{card}(\tau)$ it follows that τ is a minimal unifier for S'. Since $\text{card}(\tau) \leq n$, 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 (ii) and (iii), 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 $x \in \text{domain}(\theta)$ 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 $x \in \text{domain}(\eta)$, 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

$$\text{card}(\theta) = \text{card}(\eta) = \text{card}(\eta') + 1 \geq \text{card}(\tau) + 1 = \text{card}(\sigma).$$

For (iii), 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. □

Corollary 1.5.13 *Minimal unifiers are idempotent and fulfil the variable-disjointness condition formulated in Lemma 1.5.10.*

Note Concerning terminology, notions are treated differently in the literature (see [Lassez et al., 1988] for a comparison). In [Robinson, 1965a], J. A. Robinson used a deterministic unification algorithm, by selecting one substitution from the computed unifiers for a finite set of expressions which he called *the* most general unifier. We have subscribed to the generalized versions, which relax the deterministic selection and, henceforth, the uniqueness of an mgu, as, e.g., in [Lloyd, 1984] and [Chang and Lee, 1973]. We even permit alternative disagreement sets, because this gives more flexibility for selecting *among* mgu's (this is exploited in the proof of Proposition 4.3.1 on page 170). Furthermore, we have introduced here the notion of a minimal unifier, which turned out to be very helpful. Note that our Unification Theorem also states that *each* minimal unifier indeed can be computed. Finally, the introduction of the *declarative* concept of a computed unifier—in contrast to working with the Unification Algorithm itself, how it is normally done—makes the proof of the Unification Theorem more elegant.

Just because of its mathematical perspicuity, as a direct implementation of the variable elimination reduction ordering, the Unification Algorithm presented in Procedure 1.5.2 contains a lot of obvious redundancies: in each variable elimination operation the procedure must run through the entire expressions by instantiating the substituted variable and by afterwards computing a new disagreement set. Therefore, nobody would program this algorithm exactly the way it is presented. Instead one would rather *incrementally* perform both the instantiation operation and the recomputation of a disagreement set. We shall give an optimized version of the Unification Algorithm which is doing exactly this. Also, we shall exploit in this algorithm the fact that each unification operation can be decomposed into *binary* unification operations, which successively always compare two-element sets of expressions. In order to establish the adequacy of such a decomposition approach, we prove the following lemma.

Lemma 1.5.14 (Unification decomposition) *If σ is any unifier for a set of expressions $S = S_1 \cup S_2$, τ an mgu for S_1, and θ an mgu for $S\tau$, then $\tau\theta$ is more general than σ.*

Proof On the one hand, since σ unifies S_1 and τ is an mgu for S_1, there is a substitution γ: $\sigma = \tau\gamma$. On the other hand, by assumption, θ is an mgu for $S\tau$, hence there is a substitution η: $\gamma = \theta\eta$. Therefore $\sigma = \tau\theta\eta$. □

An iterative application of Lemma 1.5.14 justifies that the solution of a set unification problem can be broken down into any possible combination of unification subproblems induced by the input set, and that any unifier for the complete set can be obtained by composing the most general unifiers resulting from solving the unification subproblems.[18]

In particular, any set unification problem can be solved by an iterative unification of two-element sets. An incremental binary unification algorithm, which is very close to an implementation, is presented in Procedure 1.5.3.

Procedure 1.5.3 (Binary Unification Algorithm)

```
{ define binary-unification($E_1, E_2$)
    input two expressions $E_1$ and $E_2$
    output a unifier for $\{E_1, E_2\}$ or false
    initialization of a global structure : a substitution $\boldsymbol{\sigma}$, initially empty
    if   binary-unify($E_1, E_2$) : $\boldsymbol{\sigma}$
    else false }

{ define binary-unify($E_1, E_2$)
    input two expressions $E_1$ and $E_2$
    output boolean
    if   $E_1$ is a variable (string) $x_1$ and $E_2$ is a term :
        if   $\boldsymbol{\sigma}(x_1)$ is undefined :
            if   $E_2$ is a variable (string) $x_2$ and $\boldsymbol{\sigma}(x_2)$ is undefined :
                if   $x_1 = x_2$ : true
                else either $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x_1/x_2\}$ or $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x_2/x_1\}$ , true
            else if   $x_1$ does occur in $E_2\boldsymbol{\sigma}$ : false
                else $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x_1/E_2\boldsymbol{\sigma}\}$ , true
        else binary-unify($x_1\boldsymbol{\sigma}$,$E_2$)
    else if   $E_2$ is a variable (string) $x_2$ and $E_1$ is a term :
            if   $\boldsymbol{\sigma}(x_2)$ is undefined and $x_2$ does not occur in $E_1\boldsymbol{\sigma}$ :
                $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x_2/E_1\boldsymbol{\sigma}\}$ , true
            else binary-unify($E_1, x_2\boldsymbol{\sigma}$)
        else if   $E_1$ is a symbol (string) : $E_1 = E_2$
            else if   $E_2$ is a symbol (string) : false
                else let $S_1, S_2$ be the immediate subexpression sequences of
                    $E_1, E_2$ and $o_1, o_2$ their dominating symbols, respectively,
                    if   $o_1 = o_2$ : sequences('binary-unify', $S_1, S_2$)
                    else false }
```

Description of the Binary Unification Algorithm (Procedure 1.5.3) Given two input expressions E_1 and E_2, the algorithm proceeds by incrementally generating a unifier, starting with the empty mapping. Whenever the procedure

[18] A related topic, the problem of finding a simultaneous unifier for finitely many *sets* of sets of expressions is treated in [Eder, 1985a]. Using the lattice property of the collection of idempotent substitutions, it is shown there that one can proceed by, first, determining mgu's for each single set of expressions and compute a simultaneous unifier by building the supremum of the mgu's in this lattice afterwards.

comes across two distinct variables which have not yet been instantiated by $\boldsymbol{\sigma}$, one of the variables is instantiated to the other, and the already generated substitution is composed with this binding. This is the only point of indeterminism in the whole procedure. If only one of the arguments is an unbound variable, the so-called occurs-check is performed—this corresponds to the test whether a proper binding exists in the Unification Algorithm for sets of expressions (Procedure 1.5.2); the occurs-check must fail in order to permit the continuation of the unification process.

Note In an actual implementation one even might go one step further. Instead of updating the complete unifier during the unification procedure one could only accumulate a set of *local bindings* in the process which themselves would not represent the unifier but from which the unifier could be constructed as the fixpoint of a transitive closure operation of variable instantiations. In detail, assume during the unification process an unbound variable x has to be unified with an expression E, then one could just augment the current set of bindings b by setting $b := b \cup \{x/E\}$. This modification would not affect the total correctness of the unification procedure provided that the following two adjustments be made. First, the occurs-check needs to be changed slightly in that one would have to look for occurrences of a variable in an expression *modulo* the recursive instantiations induced by the current set of bindings. Secondly, after a total success of the unification procedure, from the resulting set of local bindings $\{b_1, \ldots, b_n\}$ the unifier would have to be computed as the substitution composition $b_1 \cdots b_n$. Such an approach is particularly interesting in case an entire *sequence* of unification steps has to be performed—as it is the standard case in automated deduction calculi. Then, the intermediate unifiers need not be computed, instead every new unification step could be started with the already generated set of local bindings as input and the total unifier could be computed only once at the end of the sequence of inference steps.

1.5.3 The Complexity of Unification

Unification is the central ingredient applied in each inference step of the 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 with respect to the size of the initial expressions. Of course, this would also involve that $S\sigma$ contains expressions with an exponential size. The following class of examples demonstrates this fact.

Example 1.5.2 If P is an $(n+1)$-ary predicate symbol and f a binary function symbol, then, for every $n \in \mathbb{N}$, define S_n as the set containing the atomic formulae

$P(x_1, x_2, \ldots, x_n, x_n)$, 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. Venturini-Zilli [Venturini-Zilli, 1975] could reduce the complexity to quadratic time. A number of "almost" linear algorithms have been developed in [Huet, 1976], [Martelli and Montanari, 1976, Martelli and Montanari, 1982], and in [Paterson and Wegman, 1978], whose algorithm is *really* linear (see also [Jaffar, 1984]). Similar to Herbrand's early approach in [Herbrand, 1930], all of the mentioned efficient algorithms reduce the unification problem to the problem of solving a set of equations. Since all those procedures need sophisticated additional data structures (sets of *multi-equations*) and operations (merging of sets of multi-equations) and deviate from the basic idea of Robinson's unification algorithm (the binary version specified in Procedure 1.5.3 on p. 42), Corbin and Bidoit rehabilitated Robinson's algorithm by improving it with little additional data structures up to a quadratic complexity [Corbin and Bidoit, 1983]. Although this algorithm has a higher worst-case complexity than the linear ones it turns out to be more efficient in most practically occurring cases. Corbin and Bidoit used *minimal* dags as data structures for representing logical terms.

By employing the framework of definitional expressions, in the next section, we shall present a generalization of their algorithm which facilitates the working with arbitrary, i.e., not necessarily minimal, definitional expressions.

1.6 Instantiations of Definitional Expressions

In this section the matching and unification operations are generalized to the handling of definitional expressions. Also, the *application* of a variable substitution to an expression will be improved.

The necessity for both modifications can be explained with the matching operation. The matching of ordinary logical expressions always produces a unique matching substitution, if one exists. Furthermore, if there is a matching substitution σ from an expression E_1 onto an expression E_2, then $E_1\sigma = E_2$. In the case of definitional expressions—or potential definitional expressions that form a well-defined sequence, to be more general—matters change slightly. Let us illustrate with an example how things behave here.

Example 1.6.1 Consider the task of matching a definitional expression $D_1 = f(x, x)$ with a definitional expression $D_2 = f({}_{\mathfrak{f}}f(a, a), \mathfrak{f})$. Apparently, (the expansion of) D_1 can be matched with the expansion of D_2, but there is no variable substitution σ with $D_1\sigma = D_2$.

Example 1.6.1 demonstrates that for definitional expressions the matching operation needs to be revised. The only condition which definitely must be fulfilled if a string D_1 is to be matched with a string D_2 by a matching substitution σ is that the *expansions* of $D_1\sigma$ and D_2 be identical. The open question is how $D_1\sigma$ should look like. Referring to Example 1.6.1, there are two possibilies for $D_1 = f(x,x)$. Either $D_1\sigma = f(\mathfrak{f},\mathfrak{f})$ or $D_1\sigma = f(f(a,a),f(a,a))$. It is clear that, in order to obtain compact representations, one should vote for the first alternative.

Both alternatives leave the standard manner of *applying* a variable substitution to an expression untouched. But once definitional expressions are at hand, the interesting question occurs whether the *application* of variable substitutions to expressions may be improved, too. The following trivial example proves that an improvement is really necessary.

Example 1.6.2 Consider a variable x and a substitution $\sigma = \{x/f(x,x)\}$. If the application of substitutions is taken literally, then the term $x\sigma\cdots\sigma$ has a size which is exponential with respect to the number of substitution applications.

Accordingly, the *iterative* application of substitutions may result in an exponential behaviour. Since the application of substitutions is contained as a subroutine in the matching algorithm, the iterative performance of matching operations may lead to an exponential behaviour, too.

In order to remedy this weakness, the application of a variable substitution to an expression needs to be changed.

1.6.1 Definitional Substitutions

Let, in the sequel, $\mathcal{T}_\mathcal{D}$ be the collection of definitional terms of a definitional first-order language.

Definition 1.6.1 (Definitional (variable) substitution) Let $S = (D_1,\ldots,D_m)$ be a well-defined sequence of potential definitional expressions. A *definitional (variable) substitution in context* S is any finite mapping σ: $\mathcal{V} \longrightarrow \mathcal{T}_\mathcal{D}$ with domain$(\sigma) = \{x_1,\ldots,x_n\}$ such that:

1. the sequence $S' = (\sigma(x_1),\ldots,\sigma(x_n),D_1,\ldots,D_m)$ is well-defined, and
2. for every variable $x \in$ domain(σ): $x \neq$ expansion$(\sigma(x))$, where the expansion is with respect to the sequence S'.

Any member $\langle x,t\rangle$ of a definitional substitution is called a *definitional binding*, and is written x/t.

The notion of definitional substitutions is a natural generalization of ordinary substitutions, so that every ordinary substitution is a definitional substitution. However, a crucial difference can be made between the *ordinary application* and the *definitional application* of a substitution.

Definition 1.6.2 (Definitional application of a definitional substitution) Let σ be a definitional substitution in context $S = (D, D_1, \ldots, D_m)$. The *definitional application* of σ to D produces the following string, which we write $D\sigma$.[19] Let $\mathfrak{d}_{x_1}, \ldots, \mathfrak{d}_{x_m}$ be a list of distinct new term definition symbols, one for each variable in domain(σ). Simultaneously do, for every variable x in domain(σ):

1. if $\sigma(x)$ is a definition symbol $\mathfrak{d}$ or a definition with definiendum $\mathfrak{d}$, then replace every occurrence of x in D with $\mathfrak{d}$.
2. if $\sigma(x)$ is neither a definition nor a definition symbol and either $\sigma(x)$ is a symbol (string) or x occurs only once in D, then replace all occurrences (the single occurrence) of x in D with $\sigma(x)$.
3. if $\sigma(x)$ is a complex string D', no definition, and x occurs more than once in D, then replace the leftmost occurrence of x in D with the new definition ${}_{\mathfrak{d}_x}D'$, and substitute all other occurrences of x in D by the definition symbol $\mathfrak{d}_x$.

Example 1.6.3 Given a definitional substitution $\sigma = \{x/\mathfrak{f}, y/g(a,a), z/g(a,a)\}$ in context $S = ({}_{\mathfrak{f}}f(a,a))$, and a term $t = h(z, y, g(x, z))$, the definitional application of σ to t, is a definitional term of the structure $h({}_{\mathfrak{g}}g(a,a), g(a,a), g(\mathfrak{f}, \mathfrak{g}))$, which depends on the context S.

Proposition 1.6.1 *If $S = (D, D_1, \ldots, D_m)$ is a well-defined sequence and σ is a definitional substitution in context S, then $(D\sigma, D_1, \ldots, D_m)$ is a well-defined sequence.*

Note The third case of the definition above marks the crucial difference with the ordinary application of substitutions. Evidently, this entails that whenever a definitional substitution contains no complex terms in its range, then the definitional and the ordinary manner of applying a substitution coincide.

The length increase caused by applying a definitional substitution to a string can be estimated as follows.

Proposition 1.6.2 *If D is a potential definitional expression and σ is a definitional substitution*[20]*, then* $\text{length}(D\sigma) \leq \text{length}(D) + \text{size}(\sigma) - \text{card}(\sigma)$.

Proof Each complex term $x\sigma$ in range(σ) is inserted only once into D, either by completely removing the respective occurrence of the old variable x in D or, in Case 3 of Definition 1.6.2, by replacing the occurrence of x with a new definition symbol $\mathfrak{d}_x$ of the same size; all other replacements are size-preserving. □

[19] We use the same notation used for denoting the result of the ordinary application of a substitution. Whether the ordinary or the definitional form is meant either will be said explicitly or it will be apparent from the context.

[20] The size of a substitution is the sum of the lengths of the terms in the range of the substitution.

Note This linear increase rate significantly differs from the value for the ordinary application of a substitution, which is of quadratic order.

The definitional composition of definitional substitutions is defined in analogy to the composition of ordinary substitutions, as follows.

Definition 1.6.3 (Definitional composition of definitional substitutions) Assume σ and τ to be definitional substitutions. Let σ' be the definitional substitution obtained from the set $\{\langle x, t\tau\rangle \mid x/t \in \sigma\}$ (where $t\tau$ is the definitional application of τ to t) by removing all pairs for which $x = \text{expansion}(t\tau)$, and let τ' be that subset of τ which contains no binding x/t with $x \in \text{domain}(\sigma)$. The definitional substitution $\sigma' \cup \tau'$, which we abbreviate[21] with $\sigma\tau$, is called the *definitional composition* of σ and τ.

Referring to Example 1.6.2, with $\sigma = \{x/f(x,x)\}$, under the definitional application of substitutions, the term $x\underbrace{\sigma\cdots\sigma}_{n-\text{times}}$ has the structure

$$f(\mathfrak{f}_{n-1} f(\cdots \mathfrak{f}_2 f(\mathfrak{f}_1 f(x,x), \mathfrak{f}_1), \mathfrak{f}_2 \cdots), \mathfrak{f}_{n-1})$$

which is linear with respect to n and the sizes of x and σ.

Now, we are well-equipped to turn to the definitional versions of the matching and unification operations. Recall that $\mathcal{D}_T$ and $\mathcal{D}_F$ denote the sets of term and formula definition symbols, respectively, of the underlying definitional language.

1.6.2 Matching of Definitional Expressions

Procedure 1.6.1 (Definitional Matching Algorithm)

```
{ define definitional-matching($D_1, D_2$)
    input two strings $D_1, D_2$ such that $(D_1, D_2)$ is a well-defined sequence
    output a definitional substitution in context $(D_1, D_2)$ or false
    initialization of two global structures :
    a partial mapping $\boldsymbol{\sigma}$: $\mathcal{V} \longrightarrow \mathcal{T}_\mathcal{D}$, initially empty, and
    a partial mapping id: $\mathcal{D}_T \cup \mathcal{D}_F \longrightarrow \mathcal{D}_T \cup \mathcal{D}_F$, initially defined by
        $\text{id}(\mathfrak{d}) := \begin{cases} \mathfrak{d} & \text{for any definition symbol in } D_1 \text{ or } D_2 \\ \text{undefined} & \text{otherwise} \end{cases}$
    if   definitional-match($D_1, D_2$) :
         let $\sigma$ be the definitional substitution obtained from $\boldsymbol{\sigma}$ by removing
         all pairs $\langle x, t\rangle$ with $x = \text{expansion}(t)$,
         $\sigma$
    else false }
```

[21] Again, we use the standard terminology. Possible ambiguities will be cleared up explicitly or by the context.

```
{ define definitional-match($D_1', D_2'$)
    input two potential definitional expressions $D_1', D_2'$
    output boolean
    let $\langle D_1, D_2 \rangle =$ unfold($D_1', D_2'$),                                   (1)
    if   $D_1$ is a variable $x$ and $D_2$ is a definitional term :                          (2)
        if   $\boldsymbol{\sigma}(x)$ is undefined :                                         (3)
            if   $D_2'$ is a definition with definiendum $\mathfrak{d}_2$ : $\boldsymbol{\sigma}(x) := \mathfrak{d}_2$ , true   (4)
            else $\boldsymbol{\sigma}(x) := D_2'$, true                                      (5)
        else identify($\boldsymbol{\sigma}(x), D_2'$)                                        (6)
    else if   $D_1$ is a symbol (string) : $D_1 = D_2$                                       (7)
        else if   $D_2$ is a symbol (string) : false                                         (8)
            else let $S_1, S_2$ be the immediate subexpression sequences of                  (9)
                $D_1, D_2$ and $o_1, o_2$ their dominating symbols, respectively            (10)
                if   $o_1 = o_2$ : sequences('definitional-match', $S_1, S_2$)               (11)
                else false }                                                                 (12)
```

Description of the Matching Algorithm for definitional expressions (Procedure 1.6.1)

Given two strings D_1 and D_2 such that (D_1, D_2) is a well-defined set, the algorithm proceeds by incrementally generating a definitional matching substitution, starting with the empty mapping, as in the Matching Algorithm for ordinary expressions (Procedure 1.5.1 on p. 32). The only difference from there is that here a second global structure is carried along and the definition unfolding mechanism unfold (Procedure 1.4.5 on p. 26), also used in the Identification Algorithm (Procedure 1.4.4 on p. 26), is inserted at the beginning of the procedure definitional-match (line (1)). All other parts are analogous to the ordinary Matching Algorithm, with only two exceptions. First, if an unbound variable has to be instantiated to a definition, we do not take the definition itself, but its definiendum (this is in order to avoid double occurrences of definitions). On the other hand, instead of demanding that the instantiation of a variable at the first argument be *syntactically* equal to the second argument, here the equality of the *expansions* is tested, by calling the procedure identify as a subroutine (line (6)). Evidently, the Matching Algorithm for definitional expressions is just a natural combination of the Matching Algorithm for ordinary logical expressions with the Identification Algorithm for definitional expressions.

The termination and total correctness of this algorithm are evident. Its complexity behaviour can be estimated as follows.

Proposition 1.6.3 *There is a polynomial p (of order $\mathrm{O}(n^2)$) such that for any two potential strict dag expressions D_1, D_2 which form a well-defined sequence (D_1, D_2): if the procedure* definitional-matching *is called with both strings as input, then the procedure terminates within $p\,(\mathrm{length}(D_1) + \mathrm{length}(D_2))$ steps.*

Proof Let the input D_1, D_2 be as assumed. First of all, the cost for initialization is linearly bounded. We proceed by demonstrating that the run time of the procedure is constantly related with the run time of a very similar identification problem. For this, note that the Definitional Matching Algorithm can be obtained from the Identification Algorithm (Procedure 1.4.4 on p. 26) by pushing the lines (2) to (6) of the Definitional Matching Algorithm in between the lines (1) and (2) of the Identification Algorithm and by replacing the if in line (2) of the Identification Algorithm with an else if. This demonstrates that the Definitional Matching Algorithm behaves exactly as the Identification Algorithm, except when a variable x is the definiens of the string at the first argument position. Due to the strict dag format, every occurrence of any subexpression in D_2 which is neither a definition nor a definition symbol is abbreviated. Therefore, in case the variable x is unbound, the mapping $\boldsymbol{\sigma}$ is augmented with a pair $\langle x, \mathfrak{d}_x \rangle$, where $\mathfrak{d}_x$ is a definition symbol. Since the number of these instantiation steps and their cost is linearly bounded by the input, we may safely ignore them. In case the variable is bound, the instantiation of the variable is fetched and identified with the string at the second argument position. Let $\boldsymbol{\sigma}$ be the mapping generated at the sucessful or unsuccessful end of the procedure, and let D_1' be the string obtained from D_1 by substituting each occurrence of every variable x by the definition symbol $\mathfrak{d}_x = \boldsymbol{\sigma}(x)$. Now, the problematic part of the matching process, i.e., that part in which cases of unbound variables at the first argument position do not occur, can be viewed as simulating the Identification Algorithm applied to the strings D_1' and D_2. We only have to add, for each simulation of an identification operation identify$(\mathfrak{d}_x, D)$ with a definition symbol $\mathfrak{d}_x$ at the first argument position which replaces an occurrence of a variable x in the original input string D_1, the cost for an additional unfold$(\mathfrak{d}_x, D)$ plus the cost for fetching the value of x in $\boldsymbol{\sigma}$, which are computationally innocuous, due to the strict dag format; also, the id values are properly identified. Since, by assumption, the input strings are potential strict dag expressions, the string D_1' is a potential strict dag expression, too. By Proposition 1.4.4 on p. 28, the cost for Identification of D_1' and D_2 is quadratically bounded by $\mathrm{length}(D_1') + \mathrm{length}(D_2)$, and also by $\mathrm{length}(D_1) + \mathrm{length}(D_2)$, since the structure of $\boldsymbol{\sigma}$ guarantees that $\mathrm{length}(D_1') = \mathrm{length}(D_1)$. As demonstrated above, the simulation cost is constantly related with that cost. Therefore, the cost for definitional matching is quadratically bounded by the input. It remains to be noted that, for the final removal of pairs with identical expansions from the resulting mapping, the expansions itself need not be computed. □

Since any pair of definitional expressions can be transformed into strict dag format at linear cost, we get the corollary.

Corollary 1.6.4 *For any pair of potential definitional expressions which form a well-defined sequence it can be decided whether one can be matched with the other with cost quadratically bounded by the input.*

An important consequence for the iterative execution of definitional matching operations—for instance, in an inference system—is the subsequent proposition.

Proposition 1.6.5 *Suppose σ is the output of the Procedure 1.6.1 matching a potential strict dag expression D_1 successfully with a potential strict dag expression D_2, where $S = (D_1, D_2)$ is a well-defined sequence. If D_1' is the result of making $D_1\sigma$ independent of its context S, then* $\text{length}(D_1') \leq \text{length}(D_1)+\text{length}(D_2)$.

Proof Due to the dag format, the range of σ contains only definition symbols. Therefore, by Proposition 1.6.2 on p. 46, $\text{length}(D_1\sigma) = \text{length}(D_1)$. Since $(D_1\sigma, D_2)$ is a well-defined sequence, an application of Lemma 1.4.3 on p. 25 completes the proof. □

1.6.3 Unification of Definitional Expressions

Now we present an efficient procedure for unifying definitional expressions, which is a generalization of the algorithm in [Corbin and Bidoit, 1983]. This algorithm is constructed by a straightforward composition of Robinson's binary unification algorithm (Procedure 1.5.3 on p. 42) with the Identification Algorithm (Procedure 1.4.4 on p. 26), in a very similar manner the Definitional Matching Algorithm is generated.

Procedure 1.6.2 (Definitional Unification Algorithm)

```
{ define definitional-unification($D_1, D_2$)
    input two strings $D_1, D_2$ such that $(D_1, D_2)$ is a well-defined sequence
    output a definitional unifier for $D_1$ and $D_2$ or false
    initialization of two global structures :
    a definitional substitution $\boldsymbol{\sigma}$ in context $(D_1, D_2)$, initially empty, and
    a partial mapping id: $\mathcal{D}_T \cup \mathcal{D}_F \longrightarrow \mathcal{D}_T \cup \mathcal{D}_F$, initially defined by
        $\text{id}(\mathfrak{d}) := \begin{cases} \mathfrak{d} & \text{for any definition symbol in } D_1 \text{ or } D_2 \\ \text{undefined} & \text{otherwise} \end{cases}$
    if   definitional-unify($D_1, D_2$) : $\boldsymbol{\sigma}$
    else false }

{ define definitional-unify($D_1', D_2'$)
    input two potential definitional expressions $D_1', D_2'$
    output boolean
    let $\langle D_1, D_2\rangle$ = unfold($D_1', D_2'$),                                  (1)
    if   $D_1$ is a definition with definiendum $\mathfrak{d}_1$ : let $D_1'' = \mathfrak{d}_1$      (2)
    else let $D_1'' = D_1$,                                                  (3)
    if   $D_2$ is a definition with definiendum $\mathfrak{d}_2$ : let $D_2'' = \mathfrak{d}_2$      (4)
    else let $D_2'' = D_2$                                                   (5)
    if   $D_1$ is a variable $x_1$ and $D_2$ is a definitional term :            (6)
```

```
        if   $\boldsymbol{\sigma}(x_1)$ is undefined :                                                (7)
            if   $D_2$ is a variable $x_2$ and $\boldsymbol{\sigma}(x_2)$ is undefined :                    (8)
                if   $x_1 = x_2$ : true                                                      (9)
                else either $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x_1/D_2''\boldsymbol{\sigma}\}$ or $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x_2/D_1''\boldsymbol{\sigma}\}$ , true   (10)
            else if   occurring($x_1$,$D_2\boldsymbol{\sigma}$) : false                                   (11)
                else $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x_1/D_2''\boldsymbol{\sigma}\}$ , true                                    (12)
        else definitional-unify($x_1\boldsymbol{\sigma}$,$D_2'$)                                          (13)
    else if   $D_2$ is a variable $x_2$ and $D_1$ is a definitional term :                  (14)
            if   $\boldsymbol{\sigma}(x_2)$ is undefined :                                            (15)
                if   occurring($x_2$,$D_1\boldsymbol{\sigma}$) : false                                    (16)
                else $\boldsymbol{\sigma} := \boldsymbol{\sigma}\{x_2/D_1''\boldsymbol{\sigma}\}$ , true                                    (17)
            else definitional-unify($D_1'$, $x_2\boldsymbol{\sigma}$)                                     (18)
        else if   $D_1$ is a symbol : $D_1 = D_2$
            else if   $D_2$ is a symbol : false
                else let $S_1, S_2$ be the immediate subexpression sequences of
                    $D_1, D_2$ and $o_1, o_2$ their dominating symbols, respectively,
                    if   $o_1 = o_2$ : sequences('definitional-unify', $S_1, S_2$)
                    else false }
```

Description of the Unification Algorithm for definitional expressions (Procedure 1.6.2)
Given two strings D_1 and D_2 such that (D_1, D_2) is a well-defined sequence, the algorithm proceeds by incrementally generating a definitional unifier, starting with the empty mapping, as in the Unification Algorithm for ordinary expressions (Procedure 1.5.3). Again, the difference from there is that here a second global structure is carried along and the definition unfolding mechanism is inserted, as in the Definitional Matching Algorithm. The local parameters D_1'' and D_2'' are introduced to take care that no definition is taken as the instantiation of a variable. Furthermore, the occurs-check has been adapted to the handling of definitional expressions, as presented in Procedure 1.6.3. The gist of the occurs-check, which is responsible for its polynomial run time, is that any first time a definition symbol $\mathfrak{d}$ is checked, it is marked as *visited*, so that any further time $\mathfrak{d}$ is checked, no definition unfolding is performed and the procedure immediately returns false. All other parts of the occurs-check are standard and self-explanatory.

Procedure 1.6.3 (Occurs-check)

```
{ define occurring($x, D$)
    input a variable $x$ and a potential definitional expression $D$
    output boolean
    initialization of a global structure : a mapping visited, initially empty
    if   occurs($x, D$) : true
    else false }

{ define occurs($x, D$)
    input a variable $x$ and a potential definitional expression $D$
    output boolean
    if   $D$ is a definition $_{\mathfrak{d}}D'$ or a definition symbol $\mathfrak{d}$ :
         if   visited($\mathfrak{d}$) = true : false
         else visited($\mathfrak{d}$) := true : occurs($x, D'$)
    else if   $D$ is a symbol : $x = D$
         else let $S$ be the immediate subexpression sequence
              of $D$ and $o$ its dominating symbol, respectively,
              occurs_sequence($x, S$) }

{ define occurs_sequence($x, S$)
    input a variable and a finite sequence of strings
    output boolean
    if   $S = \emptyset$ : false
    else if   occurs($x$,first($S$)) : true
         else occurs_sequence($x$,rest($S$)) }
```

The termination and the total correctness of the Definitional Unification Algorithm can be realized easily. For its complexity behaviour, we can formulate the following estimate.

Proposition 1.6.6 *There is a polynomial p (of order $\mathrm{O}(n^2)$) such that for any two potential strict dag expressions D_1, D_2 which form a well-defined sequence (D_1, D_2): if the procedure* definitional-unification *is called with both strings as input, then any deterministic execution of the procedure terminates within $p\,(\mathrm{length}(D_1) + \mathrm{length}(D_2))$ steps.*

Proof Let the input be as assumed. First of all, the cost for initialization is linearly bounded. We employ the same technique used in the proof of the polynomial run time of the Definitional Matching Algorithm. First, note that the Definitional Unification Procedure (1.6.2) can be obtained from the Identification Algorithm (Procedure 1.4.4 on p. 26) by pushing the lines (2) to (18) of the Definitional Unification Procedure in between the lines (1) and (2) of the Identification Algorithm and by replacing the if in line (2) of the Identification Algorithm with an else else if. Consequently, the Definitional Unification Procedure behaves exactly as the Identification Algorithm, except when a variable x is the definiens of one of the arguments. Due to the strict dag format, every occurrence of any subexpression in D_2 which is neither a definition nor a definition symbol is abbreviated. Therefore, in case the variable x is unbound, the definitional substitution $\boldsymbol{\sigma}$ is composed with a definitional binding $x/\mathfrak{d}_x$, where $\mathfrak{d}_x$ is a definition symbol, as in the case of matching. Also, the run time of each occurs-check is quadratically bounded by every input, even if it is not in dag format. Therefore, all operations on unbound variables can be safely ignored. Consider an arbitrary deterministic execution of the unification procedure, and let $\boldsymbol{\sigma}$ be the finally generated substitution with cardinality n. We demonstrate that the selected deterministic execution of the

Definitional Unification Procedure simulates an Identification Procedure with input $D_1\boldsymbol{\sigma}$ and $D_2\boldsymbol{\sigma}$. In any *problematic* case, i.e., a case in which an unbound variable x is at one of the argument positions, and a string D at the other which is no unbound variable, the procedure simulates an identification operation of the corresponding substituted definition symbol $\mathfrak{d}_x$ and $D\boldsymbol{\sigma}$ with at most the $2n$-fold (in the worst case where D is a bound variable) of the following overheads: the fetching of the value of one variable, the performance of the operations from line (2) to (5), and an unfold on a definition symbol and another string; the cost of all those additional operations is linearly bounded by the input, due to the strict dag format, and only linearly many (the number of variables) problematic cases may occur. Also, the id values are identified properly. Since the structure of $\boldsymbol{\sigma}$ guarantees that $\text{length}(D_1\boldsymbol{\sigma}) = \text{length}(D_2\boldsymbol{\sigma})$, the run time of the Identification Algorithm must be quadratically bounded by the input. □

Since any pair of definitional expressions can be transformed into dag format at linear cost, we get the corollary.

Corollary 1.6.7 *For any pair of potential definitional expressions which form a well-defined sequence it can be decided whether they are unifiable with cost quadratically bounded by the input.*

The following size estimate can be stated which is essential for the iterative execution of unification operations, the standard modification mechanism in inference systems.

Proposition 1.6.8 *Suppose σ is the output of the Procedure 1.6.2 unifying a potential dag expression D_1 successfully with a potential dag expression D_2, where $S = (D_1, D_2)$ is a well-defined sequence. If D_1' and D_2' are the results of making $D_1\sigma$ and $D_1\sigma$ independent of the context S, respectively, then* $\text{length}(D_1') \leq \text{length}(D_1) + \text{length}(D_2)$, *and* $\text{length}(D_2') \leq \text{length}(D_1) + \text{length}(D_2)$.

Proof In analogy to the proof of Proposition 1.6.5. □

Note The relative independency of the mechanisms used for definitional expressions from the unification task illustrates that the necessity for improving the ordinary data structures of logical expressions is nothing intrinsic to the unification problem itself, as is often argued. The fact that polynomial unification cannot be achieved with ordinary logical expressions is just *one* indication of the weakness of the traditional data structures. The basic symptom, which has not yet been emphasized sufficiently, is that an *iterative* ordinary application of substitutions may also lead to an exponential behaviour, as illustrated in Example 1.6.2 on p. 45.

1.7 Sublanguages and Normal Forms

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 of the first-order language permit the application of more efficient solution techniques than available for the full first-order format. In this section, we shall present the most important sublanguages of the first-order language.

1.7.1 Formulae in Prenex and Skolem Form

Definition 1.7.1 (Prenex form) A first-order formulae Φ is said to be a *prenex formula* or in *prenex form* if Φ is a closed formula and has the structure $Q_1x_1 \cdots Q_nx_nF$, $n \geq 0$, where the Q_i, $1 \leq i \leq n$, are quantifiers, and F is quantifier-free. We call F the *matrix* of Φ.

Proposition 1.7.1 *For every first-order formula Φ there is a formula Ψ in prenex form which is logically equivalent to Φ.*

Proof We give a constructive method to transform any closed formula Φ into prenex form. Let Q be any quantifier, $\forall$ or $\exists$. For any closed formula which is not in prenex form one of the following two cases holds. Either, Φ has a subformula of the structure $\neg QxF$; then, by Proposition 1.2.1(o) and (p), and the Replacement Lemma (Lemma 1.2.2), the formula Ψ obtained from Φ by substituting all occurrences of $\neg QxF$ in Φ by $Q'x\neg F$ is logically equivalent to Φ where $Q' = \exists$ if $Q = \forall$, and $Q' = \forall$ if $Q = \exists$. Or, Φ has a subformula of the structure $(QxF \circ G)$ where $\circ$ is any binary connective; let x' be a variable not occurring in G, and $F' = F\{x/x'\}$; then, clearly $(QxF \circ G)$ and $Qx'(F' \circ G)$ are logically equivalent; since both formulae have the same sets of free variables, by the Replacement Lemma, the formula Ψ obtained from Φ by substituting all occurrences of $(QxF \circ G)$ in Φ by $Qx'(F' \circ G)$ is logically equivalent to Φ. Consequently, in either case one can let bubble up quantifiers, and after finitely many iterations prenex form is achieved. □

Note also that the run time of this procedure is polynomially bounded by the input, and the resulting prenex formula has the same size as the initial formula.

Definition 1.7.2 (Skolem form) A first-order formula Φ is said to be a *Skolem formula* or in *Skolem form* if Φ is a prenex formula of the form $\forall x_1 \cdots \forall x_nF$, 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 (but see the remarks at the end of this section).

Definition 1.7.3 (Skolemization) Given a prenex formula Φ of a first-order language $\mathcal{L}$ with the structure $\forall x_1 \cdots \forall x_n \exists y F$, $n \geq 0$. Suppose f is an n-ary function symbol in the signature of $\mathcal{L}$ not occurring in F. Then, let F' be the formula obtained from F by replacing every occurrence of y which is bound by the leftmost occurrence of the existential quantifier in Φ with f in case $n = 0$, and with the term $f(x_1, \ldots, x_n)$ if $n > 0$. The prenex formula $\forall x_1 \cdots \forall x_n F'$ is named a *Skolemization* of Φ.

When moving to a Skolemization of a prenex formula, the collection of models does not increase.

Proposition 1.7.2 *Given a prenex formula Φ of a first-order language $\mathcal{L}$ and a Skolemization Ψ of Φ, then $\Psi \models \Phi$.*

Proof Suppose Φ has the structure $\forall x_1 \cdots \forall x_n \exists y F$, $n \geq 0$, and Ψ has the structure $\forall x_1 \cdots \forall x_n F'$, with f being an n-ary function symbol not occurring in Φ and $F' = F\{y/f(x_1, \ldots, x_n\}$. Let $\mathcal{I}$ with universe $\mathcal{U}$ be a model for Ψ. By assumption, for every variable assignment $\mathcal{A}$ from the language $\mathcal{L}$ to $\mathcal{U}$, the formula assignment $\mathfrak{I}^{\mathcal{A}}(F') = \top$. Define the variable assignment $\mathcal{A}' = (\mathcal{A} \setminus \{\langle y, \mathcal{A}(y)\rangle\}) \cup \{\langle y, \mathcal{I}^{\mathcal{A}}(f(x_1, \ldots, x_n))\rangle\}$. $\mathfrak{I}^{\mathcal{A}'}(F) = \top$, and, by the definition of formula assignments, $\mathcal{A}$: $\mathfrak{I}^{\mathcal{A}}(\exists y F) = \top$. Since $\mathcal{A}$ was chosen arbitrarily, this holds for every variable assignment. Therefore, $\mathcal{I}$ is a model for Φ. □

When moving to a Skolemization of a prenex formula the collection of models may decrease. Consequently, for the transformation of prenex formulae into Skolem form, logical equivalence must be sacrificed, and merely the preservation of satisfiability can be guaranteed.

Proposition 1.7.3 *Given a prenex formula Φ of a first-order language $\mathcal{L}$ and a Skolemization Ψ of Φ. If Φ is satisfiable, then Ψ is satisfiable.*

In order to make the proof of this proposition easier, we introduce the technically useful notion of a *partial* variable assignment.

Definition 1.7.4 (Partial variable assignment) Let $\mathcal{V}$ be the set of variables in the signature of a first-order language $\mathcal{L}$. Any partial mapping A: $\mathcal{V} \longrightarrow \mathcal{U}$ is called a *partial variable assignment* from $\mathcal{L}$ to $\mathcal{U}$. The collection of all variable assignments from $\mathcal{L}$ to $\mathcal{U}$ which are functional extensions of A is written $\hat{A}$ and named the *extension of* A.

Lemma 1.7.4 *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. Given a partial variable assignment A from $\mathcal{L}$ to $\mathcal{U}$ with domain V, an interpretation $\mathcal{I}$ for $\langle \mathcal{L}, \mathcal{U} \rangle$, and any two variable assignments $\mathcal{A}_1, \mathcal{A}_2 \in \hat{A}$.*

(a) $\mathfrak{I}^{\mathcal{A}_1}(\Phi) = \mathfrak{I}^{\mathcal{A}_2}(\Phi)$.

(b) If Φ *has the structure* $\exists xF$*, let* $\mathcal{A}_1^x$ *denote the collection of all objects* u *from* $\mathcal{U}$ *for which the modification of* $\mathcal{A}_1$ *by setting the value of* x *to* u *results in a formula assignment which maps* F *to* $\top$*, and analogously* $\mathcal{A}_2^x$*. Then,* $\mathcal{A}_1^x = \mathcal{A}_2^x$*.*

Proof The proof of (a) is obvious from Definition 1.2.18 of formula assignments. To recognize (b), let $\mathcal{A}$ be any modification of $\mathcal{A}_1$ in the value of x only such that $\mathfrak{I}^{\mathcal{A}}(F) = \top$. Set $\mathcal{A}' = (\mathcal{A}_2 \setminus \{\langle x, \mathcal{A}_2(x)\rangle\}) \cup \{\langle x, \mathcal{A}(x)\rangle\}$. Then, both $\mathcal{A}$ and $\mathcal{A}'$ are contained in the extension $\hat{B}$ of a partial variable assignment B with domain $V \cup \{x\}$. Since $V \cup \{x\}$ is the set of free variables in F, by (a), $\mathfrak{I}^{\mathcal{A}'}(F) = \top$. By symmetry, the reverse holds, and we get that $\mathcal{A}_1^x = \mathcal{A}_2^x$. □

Now, we wish to furnish the missing proof that Skolemization preserves satisfiability.

Proof of Proposition 1.7.3 Suppose Φ has the structure $\forall x_1 \cdots \forall x_n \exists y F$, $n \geq 0$, and Ψ has the structure $\forall x_1 \cdots \forall x_n F'$, with f being an n-ary function symbol not occurring in Φ and $F' = F\{y/f(x_1, \ldots, x_n)\}$. Let $\mathcal{I}$ with universe $\mathcal{U}$ be a model for Φ. Then, for every variable assignment $\mathcal{A}$, $\mathfrak{I}^{\mathcal{A}}(\exists yF) = \top$. Let $\prec_{\mathcal{U}}$ be a well-ordering[22] on $\mathcal{U}$. Let P denote the collection of all partial variable assignments from $\mathcal{L}$ to $\mathcal{U}$ with domain $\{x_1, \ldots, x_n\}$. Clearly, P is a total and disjoint partition of the collection of all variable assignments from $\mathcal{L}$ to $\mathcal{U}$. By Lemma 1.7.4, for every member $\hat{A} \in P$, the collection of objects u from $\mathcal{U}$ for which the modification of *any* element $\mathcal{A} \in \hat{A}$ by setting the value of y to u results in a formula assignment which maps F to $\top$ is unique for *all* members of A. By assumption, this collection is non-empty for every member $\hat{A}$ of P. Let $A^{\mu y}$ denote the smallest element modulo $\prec_{\mathcal{U}}$ in the collection for $\hat{A}$. We define a total n-ary mapping f': $\mathcal{U}^n \longrightarrow \mathcal{U}$ by putting $f'(u_1, \ldots, u_n) = A^{\mu y}$ with $\hat{A}$ being the extension of the partial variable assignment $A = \{\langle x_1, u_1\rangle, \ldots, \langle x_n, u_n\rangle\}$. Now, define the interpretation $\mathcal{I}_\Psi = (\mathcal{I} \setminus \{\langle f, \mathcal{I}(f)\rangle\}) \cup \{\langle f, f'\rangle\}$. Since f does not occur in Φ, $\mathcal{I}_\Psi$ is a model for Φ. We prove that $\mathcal{I}_\Psi$ is a model for Ψ. For this, let $\mathcal{A}$ be an arbitrary variable assignment from $\mathcal{L}$ to $\mathcal{U}$. Clearly, $\mathfrak{I}_\Psi^{\mathcal{A}}(\exists yF) = \top$. Since P is a total partition of the collection of all variable assignments, $\mathcal{A}$ is contained in some element $\hat{A}$ of P. If u is the smallest element modulo $\prec_{\mathcal{U}}$ in the collection $A^{\mu y}$ defined as above and $\mathcal{A}' = (\mathcal{A} \setminus \{\langle y, \mathcal{A}(y)\rangle\}) \cup \{\langle y, u\rangle\}$, then $\mathfrak{I}_\Psi^{\mathcal{A}'}(F) = \top$. Since the term assignment of $\mathcal{I}_\Psi$ and $\mathcal{A}$ maps $f(x_1, \ldots, x_n)$ to u, $\mathfrak{I}_\Psi^{\mathcal{A}'}(F') = \top$. From the fact that y does not occur in F' it follows that $\mathfrak{I}_\Psi^{\mathcal{A}}(F') = \top$. As $\mathcal{A}$ was chosen arbitrarily, for every variable assignment, the respective formula assignment of $\mathcal{I}_\Psi$ maps F' to $\top$. This proves that $\mathcal{I}_\Psi$ is a model for Ψ. □

[22] A total relation $\prec$ on a collection of objects S is a *well-ordering on* S if every non-empty subcollection 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 [Krivine, 1971]).

Theorem 1.7.5 (Skolemization Theorem) *Given a prenex formula Φ of a first-order language $\mathcal{L}$ and a Skolemization Ψ of Φ. Φ is satisfiable if and only if Ψ is satisfiable.*

Proof Immediate from Propositions 1.7.2 and 1.7.3. □

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

Proposition 1.7.6 *Given a prenex formula Φ of a first-order language $\mathcal{L}$ and a Skolem formula Ψ obtained from Φ via a sequence of Skolemizations, then* $\text{length}(\Psi) < \text{length}(\Phi)^2$, *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 sizes of the inserted Skolem terms are bounded by the size of 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. □

Note Skolemization only works for classical logic (the classical logical validity), but not for intuitionistic validity or other logical relations. In those cases more sophisticated methods are needed to encode the quantifier nesting (consult [Prawitz, 1960, Bibel, 1987], and the generalizations of their technique to non-classical logic [Wallen, 1989] and [Ohlbach, 1991]).

1.7.2 Herbrand Interpretations

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

Proposition 1.7.7 *Given a set of closed formulae Γ and a closed formula F. $\Gamma \models F$ if and only if $\Gamma \cup \{\neg F\}$ is unsatisfiable.*

Accordingly, the problem of determining whether a closed formula is logically implied by a set of closed formulae 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 the pair $\langle \mathcal{L}, \mathcal{U} \rangle$ 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 1.7.5 (Herbrand universe) (inductive)
Let S be a set of Skolem formulae 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.

If S is a singleton set $\{\Phi\}$, the same terminology shall be used for its formula Φ.

Definition 1.7.6 (Herbrand interpretation) Given a set S of formulae 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 $f(t_1, \ldots, t_n)$.

If S is a singleton set $\{\Phi\}$, the same terminology shall be used for its formula Φ.

Proposition 1.7.8 *For any first-order formula Φ in Skolem form, if Φ 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 Φ, and let $\mathcal{U}$ denote the Herbrand universe of Φ. First, we define a total mapping h: $\mathcal{U} \longrightarrow \mathcal{U}'$, 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 Φ.

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 $\mathfrak{I}'^{\mathcal{A}'}(\Phi) = \top$ entails $\mathfrak{I}^{\mathcal{A}}(\Phi) = \top$. The induction base is evident from the definition of $\mathcal{I}$, item 3 above, and the induction step follows from Definition 1.2.18. Consequently, $\mathcal{I}$ is a model for Φ. □

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

Theorem 1.7.9 (Löwenheim-Skolem theorem) *Every satisfiable first-order formula Φ has a countable model.*

Proof Given any satisfiable first-order formula Φ, let Ψ be a first-order formula obtained from Φ by prenexing and Skolemization. By Propositions 1.7.1 and 1.7.3, Ψ must be satisfiable, too. Then, by Proposition 1.7.8, there exists a Herbrand model $\mathcal{I}$ for Ψ, which is countable since every Herbrand model is countable. By Propositions 1.7.1 and 1.7.2, $\mathcal{I}$ is a model for Φ. □

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

Definition 1.7.7 (Herbrand base) Given a set S of formulae 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 $1 \leq i \leq n$. If S is a singleton set $\{\Phi\}$, the same terminology shall be used for its formula Φ.

Notation 1.7.1 Since every Herbrand interpretation $\mathcal{I}$ of a set of formulae S can be represented by the set

$$\mathcal{H} = \{A \in \mathcal{B}_S \mid \mathfrak{I}(A) = \top\} \cup \{\neg A \mid A \in \mathcal{B}_S \text{ and } \mathfrak{I}(A) = -\}$$

from now on the literal set notation will be used for denoting Herbrand interpretations.

1.7.3 Complete and Compact Sets of Connectives

From the Definition 1.2.18 of formula assignments it is apparent that, with respects to the semantics of first-order formulae, certain logical connectives are definable by other connectives. This is expressed formally with the following notion.

Definition 1.7.8 (Complete set of connectives) A subset S of the set of connectives $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ is called *complete*[23] if for any first-order formula Φ there exists a first-order formula Ψ which is logically equivalent to Φ and in which only connectives from S occur.

Proposition 1.7.10 *All sets of connectives which are supersets of one of the following sets of connectives are complete:* $\{\neg, \vee\}$*,* $\{\neg, \wedge\}$*, and* $\{\neg, \rightarrow\}$*.*

Proof The case of $\{\neg, \vee\}$ is obvious from the Definition 1.2.18 of formula assignments. By Proposition 1.2.1(a) and item 5 of Definition 1.2.18, $(F \vee G) \equiv \neg\neg(\neg\neg F \vee \neg\neg G) \equiv \neg(\neg F \wedge \neg G)$, which reduces the completeness of $\{\neg, \wedge\}$ to the first case. By Proposition 1.2.1(a) and item 6 of Definition 1.2.18, $(F \vee G) \equiv (\neg\neg F \vee G) \equiv (\neg F \rightarrow G)$, which reduces the completeness of $\{\neg, \rightarrow\}$ to the first case, too. □

Note There are connectives (the Sheffer stroke $\mid$ and $\downarrow$) which alone form complete sets; we do not consider them here. The mentioned two-element sets of connectives are the only two-element sets of the considered connectives which are complete.

Completeness is one requirement on a set of connectives, another desired property is that a set of connectives is complete and additionally permits compact formulations.

Definition 1.7.9 (Compact set of connectives) A subset S of the set of connectives $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\}$ is called *compact* if there is a polynomial p such that for any first-order formula Φ there is an equivalent formula Ψ using merely connectives from S and $\text{length}(\Phi) > p\,(\text{length}(\Psi))$.

Unfortunately, not every complete set of connectives is compact. Consider the formula class presented in Example 1.7.1 for which no set of the considered connectives without $\leftrightarrow$ can provide an equivalent formulation which is polynomial in size.

Example 1.7.1 For every $n > 0$, define $F_n = A_1 \leftrightarrow A_2 \leftrightarrow \cdots A_{n-1} \leftrightarrow A_n$.

There are polynomial transformations which are merely satisfiability and unsatisfiability preserving [Reckhow, 1976]. Also, similar to the case of Skolemization, the transformed formula logically implies the source formula, so that most problems of automated deduction—unsatisfiability detection and model generation if possible—can be solved by considering the transformed formula.

[23] Also, using this set of connectives, any n-ary *boolean function* can be equivalently formulated as a propositional formula over n nullary predicate symbols (see [Shannon, 1938] or [Moret, 1982]).

Proposition 1.7.11 *All sets of connectives which are supersets of one of the following sets of connectives are compact:* $\{\neg, \vee, \leftrightarrow\}$, $\{\neg, \wedge, \leftrightarrow\}$, *or* $\{\neg, \rightarrow, \leftrightarrow\}$.

Proof Apparently, the paraphrasing of any occurrence of a connective from the set $\{\vee, \wedge, \rightarrow\}$ by any pair of connectives $\{\neg, \circ\}$, $\circ \in \{\vee, \wedge, \rightarrow\}$, gives rise to only a constant increase in size. The fact that any superset of the mentioned sets of connectives is compact is then an immediate consequence of Lemma 2.3.7, proven in Chapter 2. □

Every compact set of the considered connectives must contain $\leftrightarrow$. This can be verified by considering Example 1.7.1. As a consequence, no minimal complete set of the considered connectives is compact. One of the superiorities of the definitional first-order language over the ordinary first-order language is expressed in the following fundamental result.

Proposition 1.7.12 *For the definitional first-order language, every complete set of connectives is compact.*

Proof It suffices to consider the minimal complete sets of connectives $\{\neg, \vee\}$, $\{\neg, \wedge\}$, and $\{\neg, \rightarrow\}$, and the manner how the paraphrasing of the material equivalence sign can be performed. First, any occurrence of a material equivalence $\Phi = (F \leftrightarrow G)$ can be substituted by the formula $\Psi = ((_{\mathfrak{f}}F \rightarrow {_{\mathfrak{g}}G}) \wedge (\mathfrak{g} \rightarrow \mathfrak{f}))$ which is only by a constant larger than Φ. Then, for any target connective $\circ \in \{\vee, \wedge, \rightarrow\}$, the paraphrasing of the other connectives produces a formula in which for every replaced connective in Ψ, $\rightarrow$ and/or $\wedge$, at most the target connective $\circ$ plus a constant number of negation signs and brackets are obtained, so that the resulting formula is also merely by a constant larger than Φ. An application of Lemma 2.3.7 completes the proof. □

Note The important consequence to be drawn from this property of the definitional language is that there are linear and equivalence preserving translations between any two complete sets of connectives. In contrast, compare the considerable amount of work done by Reckhow in [Reckhow, 1976] to distinguish between different forms of translations (so-called *direct* and *indirect* translations), which becomes obsolete for the definitional format. One may object that this is because the definitional formalism introduces the material equivalence $\leftrightarrow$ in a hidden form. But this is not true. There is a fundamental distinction between permitting the formulation of material equivalences $F \leftrightarrow G$ and the possibility of abbreviating formulae by definitions $_{\mathfrak{f}}G$. The difference is that logical calculi have to contain additional inference rules for $\leftrightarrow$, and this way introduce additional sources of redundancy. From the perspective of automated deduction, which deals with the problem of *finding* proofs, additional inference rules may have the effect that the calculus gets worse; this is because the branching rate of the calculus increases with additional inference rules, so that the proof search may become more difficult. The working with definitional expressions, however, can be organized in such a way that it induces no additional redundancy, just like the availability of dags for the unification operation does not render unification more indeterministic. This can be obtained by distinguishing in the inference mechanism between definition symbols and ordinary expressions by applying a similar technique used in the identification, matching, and unification procedures for definitional expressions. This way, compactness becomes a matter of representation, and not a matter of logical operators.

1.7.4 Formulae in Clausal Form

After prenexing and Skolemizing a formula, it is a standard technique in automated deduction to transform the resulting formula into a normal form, called *clausal form.* In order to be able to define this normal form, it is useful to extend the first-order language.

Definition 1.7.10 (Generalized conjunction and disjunction) Let $(F_1, \ldots, F_n)$, $n \geq 0$, be a sequence of formulae. The concatenation $\rceil F_1 \cdots F_n \lceil$ is called the *generalized conjunction* of $(F_1, \ldots, F_n)$; if $n = 0$, the generalized conjunction $\rceil\lceil$ is named the *verum* and is abbreviated by writing $\top$. The concatenation $\rfloor F_1 \cdots F_n \lfloor$ is called the *generalized disjunction* of $(F_1, \ldots, F_n)$; if $n = 0$, the generalized disjunction $\rfloor\lfloor$ is named the *falsum* and is abbreviated by writing $-$.

Any first-order language $\mathcal{L}$ can be extended in an obvious way to a *generalized first-order language* $\mathcal{L}_G$ by permitting formulae in which generalized disjunctions and conjunctions may occur recursively as subformulae. The declarative semantics of expressions for generalized first-order languages is defined by extending the definition of formula assignment (Definition 1.2.18).

Definition 1.7.11 (Formula assignment for a generalized first-order language) Given an interpretation $\mathcal{I}$ for a first-order language $\mathcal{L}$ with universe $\mathcal{U}$, and a variable assignment $\mathcal{A}$ from $\mathcal{L}$ to $\mathcal{U}$, then the formula assignment for the generalization $\mathcal{L}_G$ of $\mathcal{L}$ is defined by simultaneous induction as in Definition 1.2.18 with the addition of the following two lines. Let $F_1, \ldots, F_n$, $n \geq 0$, be arbitrary generalized formulae.

10. $$\mathfrak{I}^{\mathcal{A}}(\rfloor F_1 \cdots F_n \lfloor) = \begin{cases} \top & \text{if there is an } F_i,\ 0 \leq i \leq n, \text{ and } \mathfrak{I}^{\mathcal{A}}(F_i) = \top \\ \bot & \text{otherwise.} \end{cases}$$

11. $$\mathfrak{I}^{\mathcal{A}}(\rceil F_1 \cdots F_n \lceil) = \mathfrak{I}^{\mathcal{A}}(\neg \rfloor \neg F_1 \cdots \neg F_n \lfloor).$$

Proposition 1.7.13 *Any generalized first-order formula* $\rfloor F_1 \cdots F_n \lfloor$, $n > 0$, *is logically equivalent to* $F_1 \vee \cdots \vee F_n$, *and any generalized first-order formula* $\rceil F_1 \cdots F_n \lceil$, $n > 0$, *is logically equivalent to* $F_1 \wedge \cdots \wedge F_n$.

Proof Immediate from Definition 1.7.11 on p. 62 and Proposition 1.2.1(g) and (h) (the associativity of $\wedge$ and $\vee$) on p. 12. □

Definition 1.7.12 (Literal) A *literal* is an atomic formula or the negation of an atomic formula.

Definition 1.7.13 (Clause formula) If $L_1, \ldots, L_n$, $n \geq 0$, are literals, then the generalized disjunction $\rfloor L_1 \cdots L_n \lfloor$ and any of its universal closures are called *clause formulae.*

Definition 1.7.14 (Conjunctive, disjunctive normal form) A formula is said to be in *conjunctive or clausal form* if it is a generalized conjunction of clause formulae. A formula is in *disjunctive normal form* if it is a generalized disjunction of existential closures of generalized conjunctions of literals.

The following is straightforward from Definition 1.2.18 and Proposition 1.2.1.

Proposition 1.7.14 *For any first-order formula Φ in Skolem form there exists a formula Ψ in clausal form with $\Phi \equiv \Psi$.*

Proof Let F be the matrix of a first-order formula Φ in Skolem form. We perform the following four equivalence preserving macro steps. First, by items 6 and 7 of the Definition 1.2.18 of formula assignment, successively, the connectives $\leftrightarrow$ and $\rightarrow$ are removed and replaced by their definientia. Secondly, the negation signs are pushed immediately before atomic formulae, using recursively Proposition 1.2.1(a) and de Morgan's laws (j) and (k). Thirdly, apply $\vee$-distributivity from left to right until no conjunction is dominated by a disjunction, Finally, move the quantifier prefix of F directly before the clause formulae (by iteratively applying Proposition 1.2.1(q), and delete redundant quantifiers and variables from the resulting clause formulae.

As an immediate consequence of Proposition 1.7.11 we obtain the following corollary.

Corollary 1.7.15 *There is no polynomial p such that for every first-order formula Φ in Skolem form there exists an equivalent clausal form formula Ψ with* $\text{length}(\Phi) > \text{length}(p\,(\Psi))$.

Again, the class of formulae $A_1 \leftrightarrow \cdots \leftrightarrow A_n$ from Example 1.7.1 furnishes a counter-example. But there are polynomial transformations if logical equivalence is sacrificed. Similar to Reckhow's transformations, these transformations [Eder, 1985b, Boy de la Tour, 1990] are satisfiability and unsatisfiability preserving, and the transformed formula logically implies the source formula.

Note It is an important open question, however, whether the representational advantages of definitional expressions can be made available to the standard mechanisms working on formulae in clausal form, of the type discussed in Chapter 3. The apparent problem is that those mechanisms cannot handle arbitrarily complex formulae, which would be necessary in order to exploit the full power of definitional expressions. The naïve approach, to simulate the abbreviating power of definitional expressions with the addition of ordinary clause formulae (see [Tseitin, 1970]), suffers from the mentioned weakness that it increases the branching rate of the calculus tremendously.

A specific sublanguage of clausal formulae which is fundamental for the field of *logic programming* is the Horn clause language.

Definition 1.7.15 (Horn clause formula) If $(L_1, \ldots, L_n)$, $n \geq 0$, is a sequence of literals with an atom at at most one position, then any universal closure of the generalized disjunction $\rfloor L_1 \cdots L_n \lfloor$ is called a *Horn clause formula.* A Horn clause formula is called *definite* if it derives from a sequence of literals with an atom at exactly one position.

Definition 1.7.16 (Horn clause form) A formula is said to be in *Horn clause form* if it is generalized conjunction of Horn clause formulae.

In general, there is no equivalence-preserving transformation from Skolem formulae to formulae in clausal form, even if polynomiality is sacrificed. Merely satisfiability and unsatisfiability can be preserved. Furthermore, the transformed formula does not logically imply the source formula.

Proposition 1.7.16 *For any first-order formula Φ in Skolem form there exists a formula Ψ in Horn clausal form such that Φ is satisfiable if and only if Ψ is satisfiable.*

There exist translation procedures which even have a polynomial run time [Letz, 1988]. Since these methods are typically not model-theoretic and require proof-theoretic techniques, we shall not discuss them here.

1.7.5 Ground and Propositional Formulae

Apart from the restriction of a first-order language by disallowing the use of certain connectives or quantifiers, one can consider formulae without variables or function symbols.

Definition 1.7.17 (Data-logic formulae) A first-order formula is said to be a *data-logic formula* if it is a Skolem formula in which no function symbols of arity > 0 occur.

Definition 1.7.18 (Ground formulae) A first-order formula is *ground* if no variables occur in the formula.

Definition 1.7.19 (Propositional formula) A first-order formula is a *propositional formula* if no variables, quantifiers or function symbols occur in the formula.

For all three sublanguages the satisfiability problem of a formula or a finite set of formulae is decidable. Since the class of propositional formulae is fundamental for complexity theory, it will be studied extensively in Chapter 3 of this work.

Chapter 2

Complexity Measures for Logic Calculi

This chapter developes the basic concepts needed for determining the *complexities* of logic calculi. In Section 1, we introduce the notions of *logic structures* and *logics*, and identify the different principal types of *logical problems.* In the second section, *logic calculi* are introduced, as the general mechanisms for *solving* logical problems; since, computationally, logic calculi can be viewed as transition relations, afterwards, the basic properties of transition relations are introduced. For a quantitative competitive assessment of different calculi the *lengths of proofs* are of crucial importance; in Section 3, three different formats are presented for measuring proof lengths, which are of increasing degree of abstraction, and it is investigated under which conditions the higher abstraction levels *adequately* represent the lowest level, by introducing the notions of *polynomial size-transparency* and *polynomial step-transparency.* In the automation of reasoning it is not sufficient to design powerful calculi which are (weakly) complete, the ultimate goal is to develop *strongly complete* calculi or *proof procedures*; in Section 4 proof procedures are introduced and it is investigated in which way one can come from complete calculi to proof procedures.

2.1 Logics and Logical Problems

2.1.1 Logic Structures

With the use of logical expressions a wealth of domains can be modelled and many problems in these domains can be described. Yet it has proven convenient not to be restricted to *isolated* logical expressions as representing elements but to have at one's disposal *compositions* of logical expressions.[1] This leads to the concept of what we shall call *logic structures.*

[1] In fact, one has never been satisfied with logical expressions alone, additionally, structures *composed* of logical expressions were used, typically, *sets* of logical expressions.

Definition 2.1.1 (General logic structure) A *general logic structure* S on a first-order language[2] $\mathcal{L}$ is any set-theoretic object composed of expressions from $\mathcal{L}$.

This definition is very broad and does not impose any restrictions on the nature and the size of the involved objects. For example, an interpretation of a logical formula—which in general is infinite—or even the collection of *all* interpretations of a formula—which for the first-order case normally is non-denumerable—are general logic structures. In principle, logic structures have the same status as logical expressions, since logical expressions themselves are strings over an alphabet, and strings are just ordinary set-theoretic constructs. To work with structures composed of *logical expressions* just means that logical expressions constitute a useful class of basic units from which further interesting structures can be achieved by composition.

With respect to mechanization, the condition of *finiteness* is an indispensible feature. This condition characterizes *proper* logic structures, in which we are particularly interested in this work.

Definition 2.1.2 ((Proper) logic structure) A *proper logic structure* or just *logic structure* S on a first-order language $\mathcal{L}$ is any finite[3] object composed of logical expressions from $\mathcal{L}$.

Logic calculi can cope with proper logic structures only. Proper logic structures impose principal representational restrictions on general logic structures. Indeed, it is impossible to represent all general logic structures as proper logic structures, as exemplified by the above example, namely, the collection of all interpretations for a formula of first-order logic.

2.1.2 Logical Relations and Logics

While logical expressions are the basic components of logic structures and hence at the microscopic end of the spectrum, there are particularly useful and uniform mathematical objects that are best-suited to represent the top elements in the hierarchy of logic structures. These are *relations* of logic structures.

Definition 2.1.3 (General logical relation) A *general logical relation on* a collection of logic structures $\mathcal{S}$ is any n-ary relation such that every tuple $\langle S_1, \ldots, S_n \rangle \in \mathcal{R}$ consists of general logic structures from $\mathcal{S}$.

As a matter of fact, general logical relations are just general logic structures. The advantage offered by the format of logical relations is that it is both uniform

[2] Although in our work only the first-order language (or sublanguages of it) are considered, the concepts developed in this chapter are not specific to first-order languages but apply to more expressive logical languages too.

[3] With *finite* objects we mean such objects which can be *explicitly* described by finitistic methods, for instance, injectively mapped to strings over a finite alphabet.

and general enough to represent many domains in a natural way. As will become apparent in a moment, relations on logic structures also play the key role in the formulation of logical problems.

Examples of relations on logic structures The classical example of a relation on logic structures is the binary relation of *logical consequence*, which contains tuples $\langle \Gamma, \Delta \rangle$ of *sets* of logical formulae such that Δ is a logical consequence of Γ. In the recent time, the *abduction* relation is gaining interest. Abduction may be viewed as a ternary logical relation consisting of triples $\langle \Gamma, \Delta, \Lambda \rangle$ of (sets of) logical formulae such that $\Gamma \cup \Lambda$ is consistent and Δ follows from $\Gamma \cup \Lambda$ but neither from Γ nor from Λ. As a third important example, the *theory revision* or *update* relation, as typically used in information systems, could be defined as a collection of triples $\langle \Gamma, \Delta, \Lambda \rangle$ of sets of logical formulae such that (Γ and Δ are inconsistent and) Λ is some minimal subset of the theory Γ satisfying that $\Gamma \setminus \Lambda$ and the update Δ are consistent.

Of particular importance is the study of those relations which are based upon *proper* logic structures.

Definition 2.1.4 (Proper logical relation) A general logical relation is said to be a *proper logical relation* or just a *logical relation* if its tuples are composed of proper logic structures.

It is important to note that proper logical relations need not be proper logic structures. While the *elements* of the relations, the tuples, have to be finite structures, the relations themselves may be—and typically are—infinite.

Proposition 2.1.1 *Every proper logic structure is countable.*

Proof By definition, every proper logic structure can be injectively mapped to a string over some given finite alphabet. Since the set of strings over any alphabet is countable, any proper logical relation must be countable, too. □

Lastly, we can come to the formal definition of what we will understand by a *logic*, both in the general and in the proper sense.

Definition 2.1.5 (General and proper logic) A *general logic* is a pair $L = \langle \mathcal{S}, \mathcal{R} \rangle$ where $\mathcal{S}$ is a collection of general logic structures and $\mathcal{R}$ is a logical relation of arbitrary arity on $\mathcal{S}$. Whenever $\mathcal{S}$ is a collection of proper logic structures, then L is called a *proper logic* or just a *logic*.

2.1.3 Logical Problems

Given a logic $\langle \mathcal{S}, \mathcal{R} \rangle$, then some fundamental types of *logical problems* can be formulated. The simplest logical problem is the *verification* problem. It consists in finding a universal and mechanical procedure which for any tuple $t \in \mathcal{R}$ can *verify* that $t \in \mathcal{R}$. The other basic types of logical problems need some definitions for both a general and precise formulation.

Definition 2.1.6 (Projection) A *projection function* π from a positive integer n to a non-negative integer k ($k \leq n$) is a bijective mapping from a subset of the positive integers $\{1, \ldots, n\}$ onto the set of positive integers $\leq k$ which is monotonic with respect to $<$, i.e., for all i, j in the domain of π: $i < j$ implies $\pi(i) < \pi(j)$. A projection function is called *proper* in case $n > k$. If π is a projection function from n to k and $\mathcal{R}$ is a logical relation of arity n, then the *π-projection induced by π on $\mathcal{R}$*, written $\pi_{\mathcal{R}}$, is a mapping with domain $\mathcal{R}$ and defined by

$$\pi_{\mathcal{R}}(\langle t_1, \ldots, t_n \rangle) = \begin{cases} \langle t_{\pi^{-1}(1)}, \ldots, t_{\pi^{-1}(k)} \rangle & \text{if } k > 0 \\ \emptyset & \text{if } k = 0. \end{cases}$$

To present an example, suppose a projection function π is the mapping $\{\langle 1, 1 \rangle, \langle 3, 2 \rangle, \langle 4, 3 \rangle\}$, then its π-projection on a relation $\mathcal{R}$ of arity 4 maps any quadruple $\langle a, b, c, d \rangle \in \mathcal{R}$ to the triple $\langle a, c, d \rangle$. In case a projection function π is from n to 0, then the corresponding π-projection $\pi_{\mathcal{R}}$ on any n-ary relation $\mathcal{R}$ maps its members constantly to $\emptyset$.

Definition 2.1.7 (Complement projection, complement tuple) If π is a projection function from n to k, then *its complement projection*, written $\overline{\pi}$, is the projection function from n to $n-k$ which has as its domain the complement of the domain of π in the set $\{1, \ldots, n\}$. Suppose $\pi_{\mathcal{R}}$ is a π-projection induced by a projection function π and a relation $\mathcal{R}$. For arbitrary tuples t, t', if $\pi_{\mathcal{R}}(t) = \pi_{\mathcal{R}}(t')$, then we call $\overline{\pi}_{\mathcal{R}}(t')$ a *complement tuple* of $\pi_{\mathcal{R}}(t)$ under π and $\mathcal{R}$.

Note While any projection function π has a unique complement projection $\overline{\pi}$, a tuple $\pi_{\mathcal{R}}(t)$ on a relation $\mathcal{R}$ may have more than one complement tuples.

The notion of complement tuples serves as the basis for the formulation of logical *computation relations*. Beforehand, we need the concept of *logical projections*.

Definition 2.1.8 (Logical projection) A *logical projection* is a triple $\langle \mathcal{S}, \mathcal{R}, \pi \rangle$ where $\mathcal{R}$ is an n-ary logical relation on a collection of logic structures $\mathcal{S}$ and π is any projection function from n to some natural number $k \leq n$. A logical projection is called *proper* if $k < n$.

Definition 2.1.9 (Computation relation) Given a logical projection $\langle \mathcal{S}, \mathcal{R}, \pi \rangle$, the binary relation $\mathcal{C}$ consisting of the set of pairs $\{\langle \pi_{\mathcal{R}}(t), \overline{\pi}_{\mathcal{R}}(t') \rangle \mid \pi_{\mathcal{R}}(t) = \pi_{\mathcal{R}}(t')\}$, i.e., the set of all pairs of tuples and their complement tuples, is called the *computation relation* of the logical projection. For any member $\langle i, o \rangle$ in a computation relation $\mathcal{C}$ we say that i is an *input of* o in $\mathcal{C}$ and that o is an *output of* i in $\mathcal{C}$. Furthermore, we call the sets $\{i \mid i$ has an output in $C\}$ and $\{o \mid o$ has an input in $C\}$ the *input set* and the *output set* of $\mathcal{C}$, respectively. A computation relation is called *proper* if it derives from a proper logical projection.

Example 2.1.1 Assume $\mathcal{S}$ is the set of formulae of a first-order language, and let $\mathcal{R} \subset \mathcal{S} \times \mathcal{S}$ be the binary relation

$$\{\langle \Phi, \Psi \rangle \mid \Phi \equiv \Psi \text{ and there is no } \Upsilon \in \mathcal{S} \text{ with } \Phi \equiv \Upsilon \text{ and } \text{size}(\Upsilon) < \text{size}(\Psi)\}.$$

The computation relation $\mathcal{C}$ of the logical projection $\langle \mathcal{S}, \mathcal{R}, \{\langle 1, 1 \rangle\} \rangle$ associates with every input formula a logically equivalent formula which is minimal in size, i.e., for this logical projection, the computation relation $\mathcal{C}$ is $\mathcal{R}$ itself.

Clearly, the output set of any improper computation relation either is empty, in case the relation itself is empty, or contains only the empty tuple. The same holds for the input set if the projection function is from n to 0.

Any specific logical problem can be expressed as a logical computation relation $\mathcal{C}$. Its solution consists of *computing* for any member i in the input set of $\mathcal{C}$ outputs of i in $\mathcal{C}$. Due to the fact that for any n-ary logical relation up to 2^n different computation relations may be formulated, corresponding to the 2^n existing logical projections, logical computation relations offer a flexible tool for expressing various logical problems for a given logical relation.

Two fundamental types of *computation problems* may be distinguished. On the one hand, there is the task of finding for any element i in the input set of $\mathcal{C}$ *at least one* output of i in $\mathcal{C}$, which we will call an *existential* computation problem. On the other hand, one can pose the problem of computing for any element i in the input set of $\mathcal{C}$ *all* outputs of i in $\mathcal{C}$; in this case we will speak of a *universal* computation problem or an *enumeration* problem. In the general case, in which there are infinitely many output values for i, an enumeration problem can only be solved by a perpetual process which never terminates.

In the terminology of computation problems, a verification problem for a logical relation $\mathcal{R}$ turns out to be the special case of an improper computation problem, i.e., $\pi_{\mathcal{R}} = \mathcal{R}$ and $\overline{\pi}_{\mathcal{R}}$ contains at most the empty set. Since in this case the computation relation is a (constant) function, existential and universal computation problems coincide.

The notions of logical computation relations and computation problems provide both a rich and elegant terminology for describing logical problems. Let us illustrate this at some concrete computation problems that can be formulated for typical logical relations.

Examples of computation problems The *verification* problem for the binary relation of logical consequence is the classical task of automated deduction. The problem of *enumerating* the logical consequences of a formula is a proper *universal computation* task. For problems like constraint satisfaction, model generation, or query answering in logic programming, one normally is satisfied if *one* output is computed, so these are typical *existential computation* problems.

In practice, during the solution of one particular problem type any of the three types of problems may occur as subproblems. For subproblems which are universal computation problems, due to possible non-termination—in case of infinitely many output values—it may then be necessary to use interleaving techniques.

Example of a complex computation problem Complex computation problems can be formulated for the ternary abduction relation mentioned above, which consists of triples $\langle \Gamma, \Delta, \Lambda \rangle$ of sets of logical formulae such that $\Gamma \cup \Lambda$ is consistent and Δ follows from $\Gamma \cup \Lambda$ but neither from Γ nor from Λ. The typical computation relation for abduction is the one which takes pairs of the type $\langle \Gamma, \Delta \rangle$ as inputs and computes output values, *abducibles*, of the type Λ. The problem of computing abducibles contains as subproblems the verification of logical consequence and consistency.

2.1.4 Specializations of Logics

A logic explicitly distinguishes between the underlying collection of logic structures and the logical relation defined on these structures. This separation turns out to be helpful when *comparing* logics, in particular, in case one logic is a *specialization* of the other. Apparently, there are two different ways of specializing a logic. On the one hand, there are restrictions on the admissible logic structures, and on the other, there are restrictions on the logical relation.

Definition 2.1.10 ((Structure) restriction) If $\langle \mathcal{S}, \mathcal{R} \rangle$ and $\langle \mathcal{S}', \mathcal{R}' \rangle$ are logics where $\mathcal{S}' \subseteq \mathcal{S}$ and $\mathcal{R}' = \{t \in \mathcal{R} \mid t \text{ is a tuple on } \mathcal{S}'\}$, then $\langle \mathcal{S}', \mathcal{R}' \rangle$ is called a *(structure) restriction* of $\langle \mathcal{S}, \mathcal{R} \rangle$.

Definition 2.1.11 (Sublogic) If $\langle \mathcal{S}, \mathcal{R} \rangle$ is a logic and $\mathcal{R}' \subseteq \mathcal{R}$, then $\langle \mathcal{S}, \mathcal{R}' \rangle$ is called a *sublogic* of $\langle \mathcal{S}, \mathcal{R} \rangle$.

The two different ways of specializing a logic also have implications on the resulting logical problems. While, typically, the structural recognition of whether a logic structure belongs to a given collection of logic structures is decidable with low cost, proving that a tuple belongs to a logical relation is difficult in most cases, often undecidable. The possibility of keeping this separation motivates a further subclassification of logic structures.

Definition 2.1.12 (Polynomial difficulty) A collection of logic structures $\mathcal{S}$ is said to be of *polynomial difficulty* if there is a polynomial p and a decision procedure P for membership in $\mathcal{S}$ such that, for arbitrary logic structures S, the run time of P on input S is less than $p\,(\text{size}(S))$ where $\text{size}(S)$ is the string size of an appropriate string encoding of S. A logic $\langle \mathcal{S}, \mathcal{R} \rangle$ is said to be of *polynomial structure-difficulty* if $\mathcal{S}$ is of polynomial difficulty.

Any method for solving a logical problem for a logic $\langle \mathcal{S}, \mathcal{R} \rangle$ automatically carries over to any structure restriction $\langle \mathcal{S}', \mathcal{R}' \rangle$ of the logic if $\mathcal{S}'$ is of polynomial difficulty.

2.2 Logic Calculi and Transition Relations

Once a logical problem has been formulated, the question is whether there exist *effective* methods for solving arbitrary instances of this problem. The materializations of such effective mechanisms for logical relations are in the form of *logic calculi*.

2.2.1 Inference Rules and Deductions

Viewed from the highest representational level, a logic calculus is given as a *finite* set of structural rules which specify *deductive* or *inferential* operations. Traditionally, these *deduction* or *inference rules* are presented as collections of figures of the general shape

$$\frac{S_1 \ \cdots \ S_n}{S}$$

where $S_1, \ldots, S_n$ and S are *schemata* describing the permitted structures in the input tuple and the output of the rule, respectively. The paradigmatic interpretation of an inference operation according to such a figure is meta-level *matching* and *deduction*: given an already generated set $\mathcal{S}$ of proper logic structures,

1. select a tuple $\langle S_1', \ldots, S_n' \rangle$ of logic structures from $\mathcal{S}$ such that there exists a substitution σ for schema variables with $\langle S_1\sigma, \ldots, S_n\sigma \rangle = \langle S_1', \ldots, S_n' \rangle$,
2. afterwards, select a logic structure S' such that there is a substitution τ for schema variables with $S\sigma\tau = S'$.

S' is the output of the deduction step. Sometimes additional conditions need to be met to admit the performance of the deduction step. Typically, these conditions cannot be expressed using the schematic form, therefore they are formulated alongside.

The following two examples of inference rules which are taken from the *axiomatic Frege/Hilbert calculi* [Frege, 1879, Hilbert and Bernays, 1934] are concrete instances of such inference rules formulated in modern symbolism, without additional conditions. The gothic letters stand for arbitrary first-order formulae.

Example 2.2.1 (Detachment or Modus ponens rule)

$$\frac{\mathfrak{A} \qquad \mathfrak{A} \rightarrow \mathfrak{B}}{\mathfrak{B}}$$

Example 2.2.2 (First axiom rule[4] of the Frege/Hilbert system)

$$\frac{}{\mathfrak{A} \rightarrow (\mathfrak{B} \rightarrow \mathfrak{A})}$$

Note In the modus ponens rule the second substitution τ is empty, whereas in the axiom rule the first substitution σ is empty. In Gentzen's *sequent system* [Gentzen, 1935] there are inference rules in which both substitutions are non-empty.

Inference rules describe the elementary steps for building *deductions* and *proofs*, which are special types of deductions. There are different paradigms for defining deductions. One frequently used definition is to view deductions as finite sequences $(S_1, \ldots, S_n)$ of logic structures where each S_i, $1 \leq i \leq n$, can be deduced by applying an inference rule to structures with an index strictly less than i; examples of calculi in which deductions normally are understood this way are the Frege/Hilbert systems mentioned above and the *resolution calculi* [Robinson, 1965a]. Another popular interpretation is to define deductions as trees labelled with logic structures where each parent node is obtained by applying an inference rule to its successor nodes; sequent deductions were originally presented this way. In Section 3.3, *tableau deductions* [Beth, 1955, Beth, 1959, Smullyan, 1968] will be defined as trees which satisfy certain graph properties.

There is no limitation to further ways of defining deductions. However, all deductions seem to share one essential property in order to be accepted as such, namely, the cost for deciding whether a given logic structure is a deduction of a certain type must be *adequately* represented in the size of the logic structure. A reasonable weak formalization of the term 'adequately' is to read it as 'polynomially'. In other words, any collection of logic structures defining deductions of a certain type must be of polynomial difficulty.[5]

2.2.2 Deduction Processes

For investigations into the computational complexity of logic calculi, it is important to realize that one can distinguish between the *deduction* as a declarative

[4] In the literature, often a distinction is being made between inference rules of the modus pones type and so-called *axiom schemata* presented in this example. An axiom schema is sometimes not read as an inference rule, but as specifying the set of all instances of logic structures (in the example just formulae) which are instances of the schema. Computationally, such a distinction does not make sense. We treat axiom schemata simply as inference rules without any structural conditions on the input set.

[5] It is for this reason, that complementary spanning matings (Definition 3.3.12 on p. 121) cannot be accepted as deductions.

object and the *deduction process*. Deductions as static objects of the type mentioned above tend to be non-operational, in the sense that they do not prescribe the precise methodology according to which they have to be constructed. A deduction *process* can be viewed as one particular way of building up a deduction object.[6]

There is no agreement in the logic community about whether a logical system merely has to describe deductions as static objects or whether the system should also determine the operational generation paradigm of deductions. This is an important subject since different logical systems may produce the same deduction objects but completely differ in the recommended methodology *how* to construct the deduction objects. From the viewpoint of automated deduction, which is concerned with the problem of *finding* proofs, the deduction *process* is essential. Also, strictly speaking, the deduction process is the more fundamental notion and the deduction object is just a—even though extremely useful—by-product of the deduction process. This evaluation can be justified by recalling under which conditions a given object is accepted as a deduction of a type $\mathcal{S}$, namely, if there exists a *procedure* which decides in polynomial time whether the object has type $\mathcal{S}$. Consequently, the declarative reading of deductions need to be supplemented with an additional *operational* methodology.

Since, in its essence, the concept of a logic calculus is an *operational* concept, there has to be a clear idea of mechanical processing, a notion of moving from one state of affairs to another, and the possibility for doing this iteratively. Therefore, a very natural and general specification model is to interpret logic calculi as defining binary *transition relations* between proper logic structures, which play the role of the *states* in the transition relations.

2.2.3 General Notions of Transition Relations

First, we have to review the standard vocabulary for transition or *reduction* relations, which is taken from the area of rewriting systems. We begin with a series of notational abbreviations.

Notation 2.2.1 For a given transition relation $\vdash$, we let denote

$\vdash^i$	the i-fold composition of $\vdash$;
$\vdash^+$	the transitive closure of $\vdash$;
$\vdash^*$	the transitive-reflexive closure of $\vdash$;
$\dashv\vdash$	the symmetric closure of $\vdash$.

Definition 2.2.1 (Derivation, predecessor, successor, ancestor, descendant, accessibility) Given a transition relation $\vdash$, then any sequence S of objects such that for every two successive elements e_i, e_{i+1} in S: $e_i \vdash e_{i+1}$ is called a *derivation in*

[6] In the Chapters 3 and 4, we shall frequently make use of the distinction between deductions as objects and the different processes for generating deductions.

$\vdash$. If $e \vdash e'$, we call e a *predecessor* of e' and e' a *successor* of e *in* $\vdash$. If $e \vdash^{+} e'$, we call e an *ancestor* of e' and e' a *descendant* of e *in* $\vdash$. If $e \vdash^{*} e'$, we say that e' is *accessible from* e.

Notation 2.2.2 If two objects e_1 and e_2 are accessible from a common object in a transition relation $\vdash$, we write $e_1 \curlywedge e_2$, and if from two objects e_1 and e_2 a common object is accessible, we write $e_1 \curlyvee e_2$. The set of objects accessible from an object e in $\vdash$, $\{e' \mid e \vdash^{*} e'\}$, is written ${}_{e}\vdash^{*}$.

Definition 2.2.2 (Height) Let $\vdash$ be a transition relation. The *height* Λ of an element e in $\vdash$ is defined by

$$\Lambda(e) = \begin{cases} \max(\{i \mid \text{there is an } e' \text{ with } e \vdash^{i} e'\}) & \text{if it exists} \\ \infty & \text{otherwise.} \end{cases}$$

Definition 2.2.3 We say that a transition relation $\vdash$ is

1. *acyclic* if $\vdash^{+}$ is irreflexive;
2. *noetherian* if there is no infinite derivation in $\vdash$;
3. *bounded* if for all objects e: $\Lambda(e) \neq \infty$;
4. *locally confluent* if for all e_1, e_2: whenever there exists an e with $e \vdash e_1$ and $e \vdash e_2$, then $e_1 \curlyvee e_2$.
5. *(globally) confluent* if for all e_1, e_2: $e_1 \curlywedge e_2$ entails $e_1 \curlyvee e_2$.
6. *locally finite* if for all e: the set of successors of e in $\vdash$ is finite;
7. *(globally) finite* if for all e: ${}_{e}\vdash^{*}$ is finite.

Definition 2.2.4 (Normal form) An object e from the field of a transition relation $\vdash$ is *in normal form* or *irreducible in* $\vdash$ if it has no successor in $\vdash$. An object e' is said to be a *normal form of* e if e' is irreducible and $e \vdash^{*} e'$.

Definition 2.2.5 (Maximal derivation) A derivation is called *maximal in* a transition relation $\vdash$ if either it is infinite or its last member is irreducible in $\vdash$.

Let us state some more or less evident dependencies between the introduced notions (for proofs of the non-obvious results consult, for example, [Huet, 1980]).

Proposition 2.2.1

(i) Every bounded relation is noetherian, and every noetherian relation is acyclic.

(ii) Any locally finite and noetherian relation is bounded and globally finite.

(iii) Any acyclic and globally finite relation is bounded.

(iv) *If a relation is confluent, then for all* e_1, e_2*:* $e_1 \dashv\vdash^{*} e_2$ *if and only if* $e_1 \curlyvee e_2$ *("Church-Rosser" property).*

(v) *If a relation is confluent, then the normal form of any element, if it exists, is unique.*

(vi) *If every element from the field of a relation has a unique normal form, then the relation is confluent.*

(vii) *Any noetherian and locally confluent relation is confluent.*

A further basic notion is the distance between two elements in a transition relation.

Definition 2.2.6 (Distance) Let $\vdash$ be a transition relation. The *distance* δ of an element e from an element e' in $\vdash$ is defined by

$$\delta(e, e', \vdash) = \begin{cases} \min(\{i \mid e \overset{i}{\vdash} e'\}) & \text{if it exists} \\ \infty & \text{otherwise.} \end{cases}$$

2.3 Indeterministic Complexities

Given a logical computation problem, as defined in Section 2.1 (p. 69), and different logic calculi which can solve the instances of this problem, the question is which is the best among these calculi. Essentially, the competitiveness of a logic calculus is determined by two complementary factors; on the one hand, there is its ability to provide compact proofs, and on the other, there is the effort needed for finding such proofs, i.e., the search space induced by the indeterminism inherent in the calculus. In this section, we shall systematically address the problem how to measure the first of these two capabilities of a calculus, which could be called its *indeterministic power*. The indeterministic power of a calculus is determined by the complexities of the shortest proofs for a given logical computation problem. This raises the fundamental question how the complexities of proofs and deductions in a calculus should be measured. This subject is general enough to be investigated on the level of arbitrary transition relations.

2.3.1 Three Natural Measures for Derivations

For evaluating the complexity of a derivation $e_0, \ldots, e_n$ in a transition relation $\vdash$, three different measures are the obvious alternatives, which correspond to three different degrees of precision. The finest measure charges the minimal *computing cost* needed in a *basic machine model* to come from the initial state e_0 to the terminal state e_n via the given intermediate states in the derivation. The computing cost of rewriting a state e_i to a state e_{i+1} may be, for example, the minimal number of configurations of a *nondeterministic Turing machine* (or the machine operations of the indeterministic version of any alternative *realistic* machine model)[7] to transform e_i into e_{i+1}. Conceptually, the chosen basic machine model can be viewed as another (more elementary) transition relation, written $\rightarrow$. Then, the *elementary computing cost* of the derivation $D = e_0, \ldots, e_n$ can be defined as

$$\text{cost}(D) = \sum_{i=0}^{n-1} \delta(e_i, e_{i+1}, \rightarrow)$$

where δ denotes the distance between two elements in a transition relation (Definition 2.2.6 on p. 75).

Taking the elementary computing cost of a derivation as the measure of its complexity has certain disadvantages. First, for the standard realistic machine models, the measure is too detailed to be interesting as a quantity of comparison on a higher level of abstraction. Second, its value may vary strongly, depending on the chosen realistic machine model—even though only up to polynomials. Lastly, it may be very difficult to *actually obtain* the realistic computing cost, because the mapping down of high-level transition steps into basic machine operations is normally not carried out explicitly, instead one is satisfied with *knowing about the possibility* of such a transformation and its computational invariances.

An advance is offered by *abstracting* from the elementary computing cost and restricting oneself to a higher level of representation, by only considering the *realistic (string) size* of a derivation $D = e_0, \ldots, e_n$:

$$\#(D) = \sum_{i=0}^{n} \#(e_i).$$

Note We shall introduce all notions and results for the case of *realistic* string sizes. A generalization of the concepts and the propositions presented below to *unrealistic* string sizes is straigtforward, as long as the unrealistic sizes are polynomially related with realistic ones.

The highest abstraction level even disregards the *size* of a derivation $D = e_0, \ldots, e_n$ and considers only the *number of rewrite steps* in the top-level transition relation $\vdash$, in terms of logic calculi, the number of inference steps:

$$\text{steps}(D) = n.$$

Eventually, it is this measure that is being striven for. It has been used successfully for analyzing the indeterministic power of many propositional calculi, for example, in [Reckhow, 1976], [Haken, 1985], and various other papers. The abstraction performed by these authors is an abstraction *modulo polynomials*; they make plausible that the elementary computing cost is polynomially bounded by the number of inferences. Such an abstraction is very natural in that it takes into account the problem area of *NP* vs *coNP*, on the one hand, and additionally

[7] See the remarks on realistic machine models in Section 1.1.

leaves aside uninteresting subpolynomial differences which result from the choice of the realistic machine model, on the other.

One of the main objectives of this work is to explore the possibilities of abstraction modulo polynomials in a systematic manner, and to apply it to the investigation of arbitrary logic calculi and transition relations.

2.3.2 Polynomial Size- and Step-Transparency

The following two notions are fundamental for a general theory of the abstraction modulo polynomials. First, we consider the abstraction step from the elementary computing cost to the *size* of a derivation, and state under which condition such an abstraction is permissible.

Definition 2.3.1 (Polynomial size-transparency) A transition relation $\vdash$ is called *polynomially size-transparent* if there is a polynomial p such that for every derivation $D = e_0, \ldots, e_n$ in $\vdash$:

$$\text{cost}(D) < p\,(\#(D)).$$

If a transition relation $\vdash$ is polynomially size-transparent, then the size of any derivation gives a representative complexity measure of its elementary computing cost, as long as we are interested in complexities modulo polynomials. Polynomial size-transparency generalizes a basic concept introduced by Cook and Reckhow in [Cook and Reckhow, 1974]. They define a *(complete) proof system* as a (surjective) in polynomial time computable function from the set of strings to the set of valid formulae. Apparently, any proof system is polynomially size-transparent.

In order to define a *general* criterion which guarantees that we can even abstract from the *size* of a derivation, it is necessary to use polynomials in two arguments.

Definition 2.3.2 (Polynomial (step-)transparency) A transition relation $\vdash$ is called *polynomially step-transparent* or just *polynomially transparent* if there is a polynomial p in two arguments such that for every derivation $D = e_0, \ldots, e_n$ in $\vdash$:

$$\text{cost}(D) < p\,(\#(e_0), n).$$

It is apparent that polynomial transparency is a highly desirable property. If a transition relation (logic calculus) is polynomially transparent, then the number of rewrite steps (inference steps) of any derivation is a representative measure of the complexity of the derivation. In a transition relation (logic calculus) which lacks polynomial transparency the number of rewrite steps (inference steps) furnishes no reliable information about the actual complexity of the derivation. For such systems, it is impossible to measure complexities on an abstract level. Also, the comparison with other transition relations (logic calculi) may become extremely difficult. The benefit of stressing the importance of polynomial transparency is twofold, not only does it facilitate the abstract classification of different systems, it also may give advice how to improve the systems, as shown in Section 4.2 for the case of resolution. There, we shall present the principal solution methodologies when faced with the polynomial intransparency of a transition relation.

Furthermore, the concept of polynomial transparency leads to a natural generalization of the notion of a realistic machine model. By a *generalized realistic machine model* we can understand any computation model which, as a transition relation, is polynomially transparent and has the expressive power of Turing machines.

Note It is clear that indeed a polynomial in two arguments is needed for the definition of polynomial transparency. Demanding that $\text{cost}(D) < p\,(n)$ does not result in a useful notion. As an example, consider a calculus which solely can check whether a logical formula has the structure $F \vee \neg F$. According to the intended reading of inference steps, we wish to say that the calculus can verify its input in a single inference step. However, there is no complexity function (and hence no polynomial) which bounds the elementary computing cost for verifying formulae of arbitrary size that have the shape $F \vee \neg F$.

Proposition 2.3.1 *If a transition relation $\vdash$ is polynomially transparent, then $\vdash$ is polynomially size-transparent.*

Proof By assumption, there is a polynomial in two arguments p such that for every derivation D in $\vdash$: $\text{cost}(D) < p\,(\#(e_0), \text{steps}(D))$. Apparently, $\#(D) \geq \#(e_0)$ and $\#(D) \geq \text{steps}(D)$, since any state has a size ≥ 1. Hence, $\text{cost}(D) < p\,(\#(D), \#(D))$, which can be made into a polynomial with one argument. □

Since most transition relations considered here are polynomially size-transparent, the following weaker variant of polynomial transparency proves useful.

Definition 2.3.3 (Polynomial transparency wrt size) A transition relation $\vdash$ is called *polynomially transparent wrt to (derivation) size* if there is a polynomial p in two arguments such that for every derivation $D = e_0, \ldots, e_n$ in $\vdash$:

$$\#(D) < p\,(\#(e_0), n).$$

Proposition 2.3.2 *If a transition relation $\vdash$ is polynomially transparent wrt size and $\vdash$ is polynomially size-transparent, then $\vdash$ is polynomially transparent.*

The indeterministic power of a transition relation as a problem solving mechanism is intended to be the minimal cost needed for solving a given computation problem. Accordingly, we have to define when a computation problem has been *solved* by a transition process in a transition relation. This can be done by introducing for transition relations the notion of a *successful derivation* or *proof.* There are different possibilities for defining successful derivations, depending on whether existential or a universal computation problems are concerned. To avoid unnecessary complications, we shall deal with verification problems only, for which existentiality and universality coincide. Then, *proofs* can be defined by associating with a given transition relation $\vdash$ a distinguished state Ω, named the *success state*, which is assumed to be irreducible in $\vdash$. We shall call such transition relations *proof relations*. Any finite derivation in a proof relation $\vdash$ with an initial state e and terminal state Ω is said to be a *proof of* e in $\vdash$; we also say that e is *provable in* $\vdash$.

Now, we can define the properties of soundness and (weak) completeness of a proof relation with respect to a given computation problem.

Definition 2.3.4 (Sound and complete proof relation) Any pair $\Sigma = \langle \Sigma^+, \Sigma^- \rangle$ consisting of disjoint sets of states Σ^+ and Σ^- is called an *input pair*, Σ^+ is termed the *positive part* and Σ^- the *negative part* of Σ. A proof relation $\vdash$ is said to be *sound* for an input pair Σ if no element of its negative part is provable in $\vdash$. A proof relation is called *(weakly) complete* for an input pair Σ if every element of its positive part is provable in $\vdash$.

Definition 2.3.5 ((Indeterministic) polynomial boundedness) A proof relation $\vdash$ is called *(indeterministically) polynomially bounded* for an input pair Σ if there is a polynomial p such that for any element e in the positive part of Σ there exists a proof D of e in $\vdash$ with:

$$\mathrm{cost}(D) < p\,(\#(e)).$$

It is more convenient, to measure the indeterministic power of a transition relation in more abstract terms.

Definition 2.3.6 (Polynomial size-boundedness) A proof relation $\vdash$ is called *polynomially size-bounded* for an input pair Σ if there is a polynomial p such that for any element e in the positive part of Σ there exists a proof D of e in $\vdash$ with:

$$\#(e) < p\,(\#(D)).$$

Definition 2.3.7 (Polynomial step-boundedness) A proof relation $\vdash$ is called *polynomially step-bounded* for an input pair Σ if there is a polynomial p such that for any element e in the positive part of Σ there exists a proof D of e in $\vdash$ with:

$$\#(e) > p\,(\#(e), \mathrm{steps}(D)).$$

The polynomial size- or step-transparency of a proof relation permit to evaluate its indeterministic power as a problem solving mechanism for a verification problem in terms of the sizes of proofs or the sizes of inputs and proof steps, respectively. This is expressed in the following obvious propositions.

Proposition 2.3.3 *Given a proof relation $\vdash$ which is polynomially size-transparent. If $\vdash$ is polynomially size-bounded for an input pair Σ, then $\vdash$ is polynomially bounded for Σ.*

Proposition 2.3.4 *Given a proof relation $\vdash$ which is polynomially transparent. If $\vdash$ is polynomially step-bounded for an input pair Σ, then $\vdash$ is polynomially bounded for Σ.*

2.3.3 Sufficient Conditions for Polynomial Transparency

After the importance of polynomial transparency has sufficiently been emphasized, the question emerges how it can be determined whether a given transition relation has this property. Polynomial transparency is a characteristic defined on derivations of arbitrary lengths. It would be comfortable if the polynomial transparency of a transition relation could be derived from more elementary properties of the transition relation. We shall present a very useful sufficient condition for polynomial transparency which only takes into account the *step-behaviour* of a transition relation. For this purpose, we have to consider different forms of step-reliability.

Definition 2.3.8 (Polynomial time step-reliability) A transition relation $\vdash$ is called *polynomial time step-reliable* if there is a polynomial p such that for any one-step derivation $D = (e, e')$ in $\vdash$:

$$\text{cost}(D) < p\,(\#(e)).$$

Proposition 2.3.5 *If a transition relation $\vdash$ is polynomial time step-reliable, then $\vdash$ is polynomially size-transparent.*

Note The development of data structures and algorithms for *polynomial unification* can be viewed as the attempt to achieve the polynomial time step-reliability of deduction systems using unification.

Definition 2.3.9 (Polynomial size step-reliability) A transition relation $\vdash$ is called *polynomial size step-reliable* if there is a polynomial p such that for any pair $\langle e, e' \rangle \in \vdash$:

$$\#(e') < p\,(\#(e)).$$

The following obvious proposition demonstrates that transition relations which are locally infinite—and most of the traditional logic calculi are locally infinite according to the naïve reading—are very problematic from a computational point of view.

Proposition 2.3.6 *If a transition relation $\vdash$ is not locally finite, then, for any complexity function f, there exists a pair of states $\langle e, e' \rangle \in \vdash$ with*

$$\#(e') > f(\#(e)).$$

Note It is clear that locally infinite transition relations can never be polynomially transparent.

Unfortunately, polynomial time and polynomial size step-reliability of a transition relation do *not* guarantee its polynomial transparency. As a counter-example consider the transition relation $\vdash$ defined in Example 2.3.1.

Example 2.3.1 Let $\vdash = \{\langle F, (F \wedge F)\rangle \mid F \in \mathcal{L}\}$ where $\mathcal{L}$ is the language of propositional logic.

Obviously, $\vdash$ is polynomial time and polynomial size step-bounded, but it is not polynomially transparent, since after n successive rewrite steps a formula of exponential size is generated. This example is in perfect analogy to Example 1.6.2 (p. 45) where the ordinary n-fold application of a substitution $\sigma = \{x/f(x,x)\}$ to a variable x generated a term $x\sigma\cdots\sigma$, exponential in size with respect to n and the input. While (here and there) polynomial time step-reliability poses no problems, the condition of polynomial size step-reliability must be tightened.[8] A sufficient general condition is provided with the following notion.

Definition 2.3.10 (Logarithmic polynomial size step-reliability) A transition relation $\vdash$ is called *logarithmic polynomial size step-reliable*, or just *logp size step-reliable*, if there is an integer $b > 1$ and a polynomial p such that for every pair $\langle e, e'\rangle \in \vdash$:

$$\#(e') < (\log_b p\,(\#(e))) + \#(e).$$

The following lemma is fundamental for the theory of abstraction modulo polynomials.

Lemma 2.3.7 *If a transition relation $\vdash$ is polynomial time step-reliable and logp size step-reliable, then $\vdash$ is polynomially transparent.*

Proof Let $\vdash$ be as assumed, and suppose the value of the polynomial p for an argument α be

$$p\,(\alpha) = \sum_{r=1}^{s} k_r \alpha^{h_r}.$$

Consider an arbitrary derivation $D = e_0, \ldots, e_n$ in $\vdash$. The following upper bound can be obtained for the size of each e_i, $1 \leq i \leq n$. First, by simply replacing each $\#(e_j)$, $1 \leq j \leq i$, with its upper bound in terms of $\#(e_{j-1})$, we get that

$$\#(e_i) < \left(\sum_{j=1}^{i-1} \log_b p\,(\#(e_j))\right) + \log_b \#(e_0) + \#(e_0).$$

[8] In the case of substitution application, this led us to the development of the *definitional* application of a substitution.

Then, for any state e_j, the following upper bound can be obtained:

$$\#(e_j) < \#(e_{j-1}) + log_b \sum_{r=1}^{s} k_r \#(e_{j-1})^{h_r} \leq \#(e_{j-1}) + \sum_{r=1}^{s} log_b(k_r \#(e_{j-1})^{h_r}) =$$

$$\#(e_{j-1}) + \sum_{r=1}^{s} log_b k_r + \sum_{r=1}^{s} (h_r log_b \#(e_{j-1})) < c\,\#(e_{j-1})$$

for some constant c. Consequently, for any member $\log_b p\,(\#(e_j))$ of the big sum above, the following estimate holds:

$$\log_b p\,(\#(e_j)) < \log_b \left(c^{j-1}\#(e_0)\right) = \left(\log_b c^{j-1}\right) + \log_b \#(e_0) =$$

$$= \frac{\log_c c^{j-1}}{\log_c b} + \log_b \#(e_0) = \frac{j-1}{\log_c b} + \log_b \#(e_0).$$

For the entire big sum, this yields the bound:

$$\sum_{j=1}^{i-1} p\,(\log_b \#(e_j)) < \frac{i(i-1)}{2\log_c b} + (i-1)\log_b \#(e_0).$$

Therefore, for any member e_i in the derivation D:

$$\#(e_i) < \frac{1}{2\log_c b} i^2 + (i+1)\#(e_0).$$

Finally, for the whole derivation D, we get that

$$\#(D) < \sum_{i=0}^{n} \left(\frac{1}{2\log_c b} i^2 + (i+1)\#(e_0)\right)$$

which is a polynomial in $\#(e_0)$ and n, and hence demonstrates that $\vdash$ is polynomially transparent wrt size. Since, by assumption, $\vdash$ is polynomial time step-reliable, and therefore polynomially size-transparent, by Proposition 2.3.2 (p. 78) the transition relation $\vdash$ is polynomially transparent. □

Note The logp size step-reliability condition generalizes various special instances. The simplest one is the *constant size increase* condition $\#(e') < c + \#(e)$, which is that special instance of logp size where p is a constant polynomial of the form b^c.

Since the generalization of the above lemma to *unrealistic* but polynomially related size measures proves useful in practice, we shall carry out this generalization explicitly.

Corollary 2.3.8 *If a transition relation $\vdash$ is polynomial time step-reliable and logp size step-reliable where the size measure is polynomially related with a realistic size measure, then $\vdash$ is polynomially transparent.*

Proof Let $\vdash$ be a transition relation such that the chosen size measure is polynomially related with a realistic size measure # for all members in the field of $\vdash$, that is, there are polynomials p_1, p_2 such that for any object e in the field of $\vdash$:

$$\#(e) < p_1(\text{size}(e)) \quad \text{and} \quad \text{size}(e) < p_2(\#(e)).$$

In analogy to the proof of Lemma 2.3.7, there exists a polynomial p such that for any derivation $D = e_0, \ldots, e_n$ in $\vdash$:

$$\text{size}(D) < p\,(\text{size}(e_0), n).$$

Since

$$\#(D) < p_1(\text{size}(D)) < p\,(\text{size}(e_0), n) < p\,(p_2(\#(e_0)), n),$$

$\vdash$ is logp size step-reliable for the realistic size measure #. Then, by Lemma 2.3.7, $\vdash$ is polynomially transparent. □

2.3.4 Weaker Forms of Size- and Step-Transparency

There are transition relations for which polynomial size- or step-transparency cannot be guaranteed for arbitrary derivations, so that not in any case the size or the input size and the steps of a derivation give a representative measure of its complexity. But, one may argue, whenever a transition relation is applied as a mechanism of solving a computation problem, its indeterministic power is solely determined by those derivations which are *shortest proofs* of the inputs in the computation relation. Accordingly, one can weaken the notions of polynomial size- and step-transparency in such a way that only those derivations are being taken into account which are shortest proofs. The question is how to define 'short', in terms of elementary computing cost, in terms of derivation size, or number of steps. Also, the shortest proof, in anyone of these models, may violate the conditions of polynomial size- or step-transparency, but the second shortest may fit. In order to facilitate the formulation of reasonably tolerant generalizations of polynomial size- and step-transparency, we define minimal proofs with respect to polynomials.

Definition 2.3.11 [p-(size,step-)minimal proof) Given a proof relation $\vdash$ and a polynomial p.

(a) A proof D of a state e in $\vdash$ is said to be *minimal with respect to p*, or just *p-minimal*, in $\vdash$ if for any proof D' of e in $\vdash$:

$$\text{cost}(D) < p\,(\text{cost}(D')).$$

(b) A proof D of a state e in $\vdash$ is said to be *size-minimal with respect to p*, or just *p-size-minimal*, in $\vdash$ if for any proof D' of e in $\vdash$:

$$\#(D) < p\,(\#(D')).$$

(c) A proof D of a state e in $\vdash$ is said to be *step-minimal with respect to* p, or just p*-step-minimal*, in $\vdash$ if for any proof D' of e in $\vdash$:

$$\mathrm{steps}(D) < p\,(\#(e), \mathrm{steps}(D')).$$

Now, polynomial difference in complexity poses no problems, not the absolutely shortest proof must be taken, any proof will do which p-simulates the shortest one. Using p-size- and p-step-minimal proofs the notions of polynomial size- and step-transparency can be weakened as follows.

Definition 2.3.12 (Weak polynomial size-transparency) A proof relation $\vdash$ is called *weakly polynomially size-transparent* for an input pair Σ if there are polynomials p and p' such that for every element e in the positive part of Σ there exists a p-size-minimal proof D of e in $\vdash$ with

$$\mathrm{cost}(D) < p'(\#(D)).$$

Definition 2.3.13 (Weak polynomial (step-)transparency) A proof relation $\vdash$ is called *weakly polynomially (step-)transparent* or just *weakly polynomially transparent* if there are polynomials p and p' such that for every element e in the positive part of Σ there exists a p-step-minimal proof D of e in $\vdash$ with

$$\mathrm{cost}(D) < p'(\#(e_0), \mathrm{steps}(D)).$$

Note One could even be more liberal and only demand the existence of p-minimal proofs in both definitions above. We think that the resulting notions would become too weak, for the following reason. With the notions of weak polynomial size- and step-transparency we intend to express that the respective chosen abstraction level indeed provides a representative complexity measure for the indeterministic power of a transition relation, even though not for the *absolutely* shortest proofs, so at least for *one* of the short proofs. But the class of short proofs should be *defined* in terms of the respective abstraction level, this way demonstrating the usefulness of the abstraction level.

2.4 Proof Procedures

While proof relations which are (weakly) complete only ensure the *existence* of proofs for any state in the positive part Σ^+ of a given input pair $\Sigma = \langle \Sigma^+, \Sigma^- \rangle$ (see Definition 2.3.4 on p. 79), in automated deduction, one is interested in *really finding* a proof. For such purposes one needs proof relations which meet the stronger requirement of strong completeness.

2.4.1 Strong Completeness

Definition 2.4.1 (Strong completeness) A proof relation $\vdash$ is called *strongly complete* for an input pair Σ if, for any element e in the positive part of Σ, every maximal derivation in $\vdash$ with initial state e is finite and terminates in the success state Ω.

Definition 2.4.2 (Proof procedure) A proof relation $\vdash$ which is sound and strongly complete for an input pair Σ is named a *proof procedure* for Σ.

Note Most theorem proving programs are implementations of deterministic proof procedures or deterministic implementations of proof procedures. In general, proof procedures *need* not be deterministic, and indeed, most of them are nondeterministic. It is an important research topic in the field of automated deduction to extract from a given nondeterministic proof procedure an optimally behaving deterministic subsystem.

The property of strong completeness puts very strict requirements on proof relations. Thus, every proof procedure must be acyclic and, consequently, asymmetric and irreflexive. For the *design* of proof procedures it is instructive that the property of strong completeness can be broken up into the two notions of *proof-confluence* and *semi-noetherianness.*

Definition 2.4.3 (Semi-confluence, proof-confluence) Given a proof relation $\vdash$ and an input pair Σ. Let ${}_{\Sigma}\vdash^{*}$ denote the set of all states from the field of $\vdash$ which are accessible from objects in the positive part of Σ. If the field restriction[9] of $\vdash$ to ${}_{\Sigma}\vdash^{*}$ is confluent, then $\vdash$ is called *semi-confluent* for Σ. A proof relation $\vdash$ is said to be *proof-confluent* for an input pair Σ if any state which is accessible from an element in the positive part of Σ is provable in $\vdash$.

Proposition 2.4.1 *A proof relation $\vdash$ is proof-confluent for an input pair Σ if and only if $\vdash$ is complete and semi-confluent for Σ.*

Proof The 'only if'-part is trivial. For the proof of the 'if'-part, assume $\vdash$ to be complete and semi-confluent for an input pair Σ. Let e be an arbitrary state accessible from some element e' in the positive part of Σ. By the completeness assumption, $e' \vdash^{*} \Omega$, and by the semi-confluence assumption, $e \curlyvee \Omega$. According to the definition of proof relations, the success state Ω must be irreducible, therefore $e \vdash^{*} \Omega$. □

Definition 2.4.4 (Semi-noetherianness) A proof relation $\vdash$ is *semi-noetherian* for an input pair Σ if there are no infinite derivations starting from states in the positive part of Σ.

[9] The *field restriction* of a relation R *to* a set S is the collection of tuples from R with elements in S.

Proposition 2.4.2 *A proof relation $\vdash$ is strongly complete for an input pair Σ if and only if $\vdash$ is proof-confluent and semi-noetherian for Σ.*

Proof The 'only if'-part is trivial. For the proof of the 'if'-part, let D be any maximal derivation from an element in the positive part of Σ. By semi-noetherianness, D is finite and has a last element e. By the property of proof-confluence, $e \overset{*}{\vdash} \Omega$. From the maximality of D follows that e is irreducible, therefore e must be the success state Ω. □

Definition 2.4.5 (Noetherianness) A proof relation $\vdash$ is *noetherian* for an input pair Σ if there are no infinite derivations starting from states in the positive or negative part of Σ.

Definition 2.4.6 (Decision procedure) A proof relation $\vdash$ which is sound, noetherian, and strongly complete for an input pair Σ is a *decision procedure* for Σ.

2.4.2 From Completeness to Strong Completeness

Although the ultimate goal in the automation of reasoning is the design of proof procedures, it is often very difficult to construct a proof procedure all at once. Instead, it is reasonable to start off from a sound and complete proof relation and, in a second step, to modify the relation and its internal data structures in such a way that strong completeness is obtained. Normally, this is achieved by putting an additional control structure on top of the relation. One can distinguish two principle methodologies for this approach, the object-level and the meta-level approach. The object-level approach works by *state saturation* whereas the meta-level approach works by *state enumeration.*

State Saturation The state saturation methodology presupposes a proof relation to be sound, complete, and proof-confluent, and achieves strong completeness by making the relation semi-noetherian. There is no standard technique for obtaining semi-noetherianness, it strongly depends on the structure of the states of the proof relation.

The methodology of state saturation can be illustrated at best with logic calculi of a generative nature, in which the states of the proof relation are sets of logical formulae which principally are accumulated (as in resolution systems). Typically, in such proof relations, one can directly step to the success state if a formula of a "success" type (in resolution: the empty clause) is contained in the current state. For this particular type of proof relations, semi-noetherianness can be achieved in two steps. First, construct a *saturation relation* $\vdash_{\mathcal{S}}$ from the initial proof relation $\vdash$, by modifying $\vdash$ in such a way that, for every formula e in the positive part of the input pair Σ, *every* formula in $_e\vdash^*$ (i.e., the set of states accessible from e) is contained in a state of every maximal derivation in $\vdash_{\mathcal{S}}$ with initial state e. Due to the completeness of $\vdash$, this *fairness* condition guarantees that in every such maximal derivation a state occurs which contains a formula of the success type. The second step simply consists in replacing every pair $\langle e, e' \rangle \in \vdash_{\mathcal{S}}$ where e contains a success formula with the pair $\langle e, \Omega \rangle$. Apparently, the resulting proof relation is semi-noetherian.

A *general* method for obtaining semi-noetherianness for proof relations which are sound, complete, and proof-confluent, is to minimize the distances from the success state.

Proposition 2.4.3 *Given a proof relation $\vdash$ which is sound, complete, and proof-confluent for an input pair Σ. If, for every state e accessible from the positive part of Σ, there exists a number $k \in \mathbb{N}$ such that for every state e' with $e \overset{i}{\vdash} e'$, $i \geq k$: $\delta(e', \Omega, \vdash) < \delta(e, \Omega, \vdash)$, then $\vdash$ is semi-noetherian for Σ.*

State Enumeration The state enumeration methodology merely presupposes the soundness and completeness of a proof relation $\vdash$ for an input pair $\langle \Sigma^+, \Sigma^- \rangle$, and hence has more cases of application. Using this approach, strong completeness is obtained on the meta-level by *enumerating* all possible states accessible from a given state. This can be done in two steps. First, construct an *enumeration relation* $\vdash_{\mathcal{E}}$ which has the property that, for every state $e \in \Sigma^+$: *every* state accessible from e except the success state occurs in *every* maximal derivation in $\vdash_{\mathcal{E}}$ with initial state e. The second step simply consists in replacing every pair $\langle e, e' \rangle \in \vdash_{\mathcal{E}}$ where $e \vdash \Omega$ with the pair $\langle e, \Omega \rangle$. Apparently, the resulting proof relation is strongly complete.

Chapter 3

Propositional Calculi

Since first-order calculi are to a large extent determined by their propositional or ground fragments, we devote a whole chapter to the presentation of logic calculi for propositional and ground formulae. The first section contains some general remarks on the central role of propositional logic in complexity theory; also we shortly argue why the traditional calculi of the generative type are not suited for the purposes of automated deduction. In the second section, resolution and semantic tree systems are introduced, which both utilize a condensed variant of the cut rule from sequent systems, resolution in a forward, and semantic trees in a backward manner. In Section 3, tableau and connection calculi are introduced, which in their pure versions are cut-free systems. A straightforward combination of both systems leads to the so-called *connection tableaux*, which are treated in the fourth section. Due to their lack of proof-confluence, connection tableaux are not optimally suited for the propositional case, but they offer an excellent framework for the development of successful first-order calculi. In Section 5, we present a method to overcome the weakness of connection tableaux concerning indeterministic power by adding the *folding-up* rule, which is a *controlled* incorporation of lemmata and the cut rule, and an improvement of factorization used in connection calculi.

3.1 The Importance of Propositional Logic

Deciding the logical validity of a formula of propositional logic is one of the central problems in time complexity theory. This is, on the one hand, because very many other important problems are in essence of the same difficulty. On the other hand, propositional logic is contained as a central sublanguage in almost all logic-based languages and systems and, therefore, gives a lower complexity bound on the handling of those more expressive logical formalisms.

3.1.1 Propositional Logic and Complexity Theory

In 1971, Cook defined the *NP-class*, as the collection of all languages accepted by a *non-deterministic algorithm* in time polynomially bounded by the size of the input. He also showed the language of the satisfiable propositional formulae to be an adequate representative of this class, by proving that the recognition problem of any language in the NP-class can be reduced to the propositional satisfiability problem, at polynomial cost [Cook, 1971]. This manifests the so-called *NP-completeness* of the satisfiability problem, a property it shares with hundreds of other well-known problems[1].

The complement problem of proving the satisfiability is to demonstrate the *unsatisfiability* of a propositional formula, which is equivalent to showing the *validity* of the negation of the formula. This problem belongs to the most difficult problems in the *coNP-class*, which is defined to contain exactly the *complements* of the languages in the NP-class.

The whole area of time complexity is full of open questions. First, it is not known whether the NP-class or the coNP-class differ from the *P-class*, which is the collection of all languages accepted by a *deterministic* algorithm in polynomial time. Secondly, it is unknown whether NP and coNP are different—since P is closed under complements, such a result would entail that P $\neq$ NP and P $\neq$ coNP. The *satisfiability* of a propositional formula can be "solved" in polynomial time by a non-deterministic algorithm[2]. One *merely* has to guess the right truth valuation, which then can be checked in polynomial time with respect to the size of the input. For the complement problem, that is, proving the *unsatisfiability* of a propositional formula, it is not known whether there exist non-deterministic algorithms which have a polynomial run time. Or, in terms of logic calculi, which constitute the most natural formulations of non-deterministic algorithms, it is not known whether there exists a sound logic calculus such that every valid propositional formula has a proof in the system which is polynomially bounded by the size of the formula. Therefore, according to our intuitions, the recognition of satisfiability seems to be "easier" than the recognition of unsatisfiability or validity, which means that one is rather inclined to believe that the NP-class is contained in the coNP-class than the converse. The strange thing about this intuition is that it leads to what might be called the *paradox of time complexity.*

Proposition 3.1.1 (The paradox of time complexity)

$$NP \subseteq coNP \ \textit{ if and only if } \ coNP \subseteq NP.$$

Proof Suppose NP $\subseteq$ coNP. Let $\mathcal{L}$ be any language in coNP. By definition, its complement language, written $\mathcal{L}^{-1}$, is in NP and, by assumption, $\mathcal{L}^{-1}$ is in

[1] For a nice survey, see the book of Garey and Johnson [Garey and Johnson, 1979].

[2] We have put the term 'solved' in quotation marks, because a non-deterministic algorithm is a mathematical notion and there may not exist a corresponding actual computing devise, as opposed to deterministic algorithms. It is for this reason that the term 'non-deterministic algorithm' can be very misleading.

coNP. Therefore, by the definition of complement, $\mathcal{L}^{-1^{-1}} = \mathcal{L}$ is in NP. The other direction holds by analogy. □

So, either NP = coNP or none of them is properly contained in the other.[3]

3.1.2 Generative Calculi

Historically the first formalized logical rule system was developed by Frege in [Frege, 1879], which was modified and elaborated in [Hilbert and Bernays, 1934]. Since, traditionally, the Frege/Hilbert systems are subclassified into *axiom schemata* and *proper* inference rules, these rule systems are called *axiomatic calculi*[4]. Another very influential work in logic is Gentzen's dissertation [Gentzen, 1935] where consecutively two alternative characterizations of logical consequence were developed, the *natural deduction system* and the *sequent system*. The natural deduction system is an attempt to formalize the mathematical way of presenting arguments, by making assumptions, drawing conclusions from assumptions, and discharging assumptions. The elegance of natural deduction is the way logical symbols are treated, by having both introduction and elimination rules for the symbols, which is not the case in Frege/Hilbert systems. The sequent system is a proof system combining two interesting properties. On the one hand, unlike natural deduction, the system is *logicistic*, that is, all derived formulae are logically valid by themselves and do not depend on assumption formulae; on the other hand, the sequent system adopts from natural deduction the symmetric classification of inference rules into introduction and elimination rules for the logical symbols.

The naïve transitional interpretations of all three systems, by which, starting from the empty set of formulae, successively new formulae are generated, suffer from two fundamental weaknesses which render the procedures unsuitable as bases for solving logical computation problems. The first weakness—in fact, this is the crucial one—derives from the manner the systems tackle a verification problem. The verification problem is transformed into a proper computation problem, the computation problem is solved by guessing an output, and finally, it is verified whether the output is indeed the desired one. Since the structure of the formula to be proven or refuted is solely used as an exit information, the procedures lack *goal-orientedness*.[5] The second weakness is due to the fact that all systems contain rules which induce an infinite branching rate of the respective

[3] The consequence to be drawn from this observation is that as a heuristic conceptual guide line for our problem solving intuitions, the notion of a non-deterministic algorithm alone seems not to be sufficient, because it presents a distorted picture of this area of complexity theory. Additionally, one should develop a complementary *positive* characterization of the coNP-class, or even device completely new concepts which better illuminate our intuitions.

[4] From a computational point of view this distinction is not very instructive (see the remarks in Footnote 4 of Chapter 2).

[5] To call upon an analogy from the domain of sort algorithms, the procedures have the efficiency status of *permutation sort*.

calculus, i.e., the corresponding transition relations are not locally finite. Consequently, for any positive integer n, there are infinitely many deductions with n inference steps, and, by Proposition 2.3.6 on p. 80, any deduction may arbitrarily increase in size within a single inference step. Both disadvantages render the construction of proof procedures from the calculi very difficult.[6]

3.2 Resolution Systems and Semantic Trees

Most efforts in automated deduction concentrate on demonstrating the unsatisfiability of formulae in clause normal form. The restriction to this normal form permits the application of particularly efficient proof techniques. In this section, two families of calculi are presented, resolution systems and semantic tree procedures. Both families operate with a single inference rule, namely, a condensed variant of the *cut rule* from the sequent system. 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.

3.2.1 Resolution

Resolution was introduced by J. A. Robinson in [Robinson, 1965a] as a calculus for first-order formulae in clause normal form. Resolution systems are typically defined as manipulating sets of literals. The associativity, commutativity and idempotency of the logical disjunction operator with respect to the denotations assigned by an interpretation admits a particularly simple representation of a clause formula $c = \forall x_1 \cdots \forall x_n (\rfloor L_1, \ldots, L_n \lfloor)$, namely, by the *set* of literals $\{L_1, \ldots, L_n\}$ occurring as disjuncts in the matrix of c.

Definition 3.2.1 (Clause) A *clause* is a finite set of literals. A clause is *tautological* if it contains a literal and its complement.[7] A *unit clause* is a clause containing exactly one literal.

The semantic assignment function on logical formulae can be extended to also give meaning to clauses.

Definition 3.2.2 (Clause assignment) Given an interpretation $\mathcal{I}$ for a first-order language $\mathcal{L}$ with universe $\mathcal{U}$, then the following line is added to the definition of formula assignment (Definitions 1.2.18 and 1.7.11). For any clause $c = \{L_1, \ldots, L_n\}$, $n \geq 0$, with $x_1, \ldots, x_n$ being the variables occurring in the literals of c:

[6] Note, however, that the second disadvantage can be completely overcome, at least for the propositional case, by using truth value variables, which permit to reduce the branching rate of any critical rule from ∞ to 1 (see [Letz, 1993a]).

[7] Our definition of a clause slightly differs from the ones given in [Robinson, 1965a] or [Davis and Putnam, 1960], which demand that clauses be non-tautological.

11. $\mathfrak{I}(c) = \mathfrak{I}(\forall x_1 \cdots \forall x_n \lrcorner L_1, \ldots, L_n \llcorner)$.

Since every interpretation assigns $\top$ to every tautological clause, we have the following proposition.

Proposition 3.2.1 (Tautology deletion) *If a set of clauses S contains a tautological clause c, then $S \equiv (S \setminus \{c\})$.*

Resolution can be formulated very naturally as consisting of a unique inference rule. Here we present the propositional or ground fragment of resolution.[8]

Definition 3.2.3 (Ground resolution rule) Let L be a literal and c_1 and c_2 clauses with $L \notin c_1$ and $\sim\!L \notin c_2$. The *ground resolution rule* has the shape:

$$\frac{\{L\} \cup c_1 \qquad\qquad \{\sim\!L\} \cup c_2}{c_1 \cup c_2}\,.$$

The clause $c_1 \cup c_2$ is called a *ground resolvent of* $\{L\} \cup c_1$ *and* $\{\sim\!L\} \cup c_2$ *over* L, and $\{L\} \cup c_1$ and $\{\sim\!L\} \cup c_2$ are termed *parent clauses* of the resolvent. Since every pair c_1, c_2 of ground clauses has at most one non-tautological ground resolvent, this resolvent, if it exists, will be called *the* ground resolvent of c_1 and c_2, and written $\mathfrak{R}(c_1, c_2)$.

We first introduce the *static* deduction objects generated by using the resolution rule.

Definition 3.2.4 (Ground resolution proof) A *ground resolution deduction* or *proof of* a clause c_n *from* a set of clauses S is a finite sequence $D = (c_1, \ldots, c_n)$ of clauses such that each clause c_k, $1 \leq k \leq n$, is the (non-tautological) ground resolvent of two parent clauses where for each parent clause c: either $c \in S$ or $c = c_i$ and $i < k$. A ground resolution proof of the empty clause from a set S is called a ground resolution *refutation of* S. A ground resolution proof of a clause c_n from a set S of ground clauses is *compact* if D has no proper subset whose sequence normalization is a ground resolution proof of c_n from S.

Example 3.2.1 Given a set of ground clauses

$$S = \{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}\}.$$

The sequence of clauses

$$(\{p\}, \{q\}, \{\neg p\}, \emptyset)$$

is a compact ground resolution refutation of S.

[8] Propositional resolution is the dual of Quine's *consensus* [van Orman Quine, 1955].

Definition 3.2.5 (Ground resolution dag) A *ground resolution dag* is a pair $T = \langle t, \lambda \rangle$ consisting of a directed acyclic graph t which is rooted, finite, and binary branching, and a function λ labelling its nodes with clauses and its edges with literals in such a way that

1. the clause $c_1 \cup c_2$ at each non-leaf node N is the (non-tautological) ground resolvent of the clauses $\{L\} \cup c_1$ and $\{\sim L\} \cup c_2$ at its successor nodes N_1 and N_2,
2. and the edges e_1 and e_2 leading from N to N_1 and N_2 are labelled with the literals $\sim L$ and L, respectively.

The clause at the root of a ground resolution dag T is called the *bottom clause* of T. Let S be the set of clauses at the leaves of a ground resolution dag T. We say that T is a ground resolution dag *for* S. A ground resolution dag for S with empty bottom clause is called a *ground resolution refutation dag for* S. If the dag t of a ground resolution dag T is a tree, T is named a *ground resolution tree.*

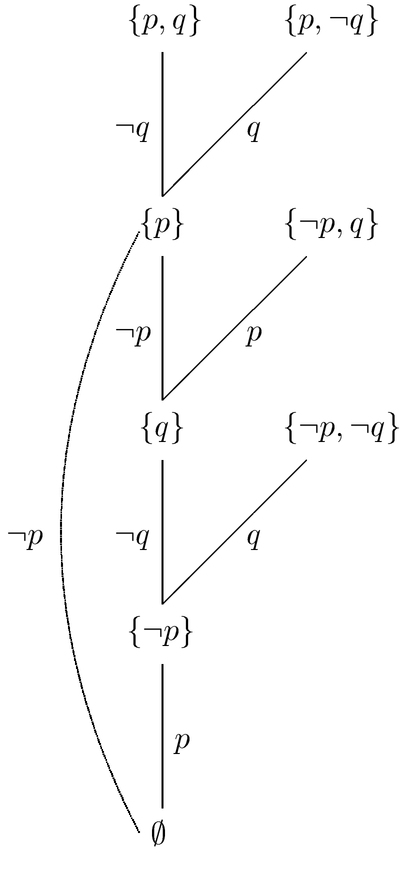


Figure 3.1: Resolution dag for $\{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}\}$.

Convention In order to express the forward-oriented working methodology of resolution calculi, we shall display resolution dags as ordinary upward trees.

An example of a ground resolution refutation dag is depicted in Figure 3.1. The relation between resolution proofs and resolution dags is apparent. To every ground resolution proof $(c_1, \ldots, c_n)$ from a set of clauses S there can be constructed a ground resolution dag for S with bottom clause c_n according to the following indeterministic procedure.

Procedure 3.2.1 (Transformation from resolution proofs to resolution dags) Given a ground resolution proof $D = (c_1, \ldots, c_n)$ from a set S of ground clauses. Starting with a one-node dag labelled with c_n, iterate the following procedure. As long as the current dag has clauses at leaf nodes which are not in S, choose such a leaf node with clause c_k, $1 \leq k \leq n$, attach two new successor nodes, label them with two parent clauses of c_k where for each parent clause c: either $c \in S$ or $c = c_i$ and $i < k$, and mark the edges with the respective complementary literals.

Conversely, from any resolution dag a corresponding resolution proof can be constructed.

Definition 3.2.6 (Resolution inference steps) The *number of resolution inference steps* of a resolution proof D, written $\text{steps}(D)$, is $\text{length}(D)$, and the *number of resolution inference steps* of a resolution dag T, written $\text{steps}(T)$, is the number of non-leaf nodes of T.

Proposition 3.2.2 *Given a ground resolution proof D and a ground resolution dag T obtained from D by a deterministic execution of Procedure 3.2.1, then* $\text{steps}(T) \leq \text{steps}(D)$*; and if D is compact, then* $\text{steps}(T) = \text{steps}(D)$.

Proposition 3.2.3 *To every compact ground resolution refutation there exists exactly one ground resolution dag which can be obtained by applying Procedure 3.2.1.*

Hence, for compact resolution proofs, Procedure 3.2.1 is a mapping, but this mapping is not injective in the general case. Consequently, the dag representation is more indeterministic than the sequence representation. The resolution dag of Figure 3.1 is the result of applying the transformation procedure to the resolution proof given in Example 3.2.1.

Proposition 3.2.4 (Soundness of the ground resolution rule) *Any ground resolvent is logically implied by the set of its parent clauses.*

Proof Let $\mathcal{H}$ be a Herbrand model[9] for a set of parent clauses $\{\{L\} \cup c_1, \{\sim L\} \cup c_2\}$ of a ground resolvent c, i.e., there are literals $L_1 \in \{L\} \cup c_1$ and $L_2 \in \{\sim L\} \cup c_2$ with $L_1 \in \mathcal{H}$ and $L_2 \in \mathcal{H}$. This entails that $L_1 \neq \sim L_2$. Consequently, either $L_1 \neq L$ or $L_2 \neq \sim L$, so that either L_1 or L_2 is contained in the resolvent c. Therefore, the formula assignment of $\mathcal{H}$ maps c to $\top$. □

[9] Recall that we use the literal set notation for denoting Herbrand interpretations (Notation 1.7.1).

Corollary 3.2.5 (Soundness of ground resolution) *If there is a ground resolution proof of a clause c from a set of clauses S, then $S \models c$.*

Proof Immediate from Proposition 3.2.4 and the transitivity of $\models$. □

Resolution is a refutational proof method. It proceeds by demonstrating that from a given initial unsatisfiable set of clauses the empty clause can be a deduced, which is false under every interpretation and hence explicitly testifies the unsatisfiability of the input set. This approach is sufficient for proving theoremhood, because any verification problem of logical validity or logical implication is reducible to an unsatisfiability problem. Ground resolution is *refutation-complete* for ground clause formulae, i.e., for every unsatisfiable set S of ground clauses, there exists a ground resolution refutation of S. Interestingly, resolution is not *deduction-complete*, i.e., not every clause logically implied by a set of clauses can be deduced by resolution.[10] From the viewpoint of automation, however, this weakness can be seen as an advantage, because this way the number of possible proofs may be strongly restricted. We wish to postpone the completeness proof of ground resolution to Subsection 3.2.5. There we shall demonstrate that even a refinement of ground resolution—i.e., a system in which not every resolution step is permitted—has this property, namely, the *Davis/Putnam calculus*.

3.2.2 Resolution Deductions vs Resolution Procedures

The sequence representation of resolution proof objects induces a particularly natural operational reading. The ground resolution *calculus* proceeds by reasoning in a forward manner just like a sequent calculus or an axiomatic calculus of the Frege/Hilbert style.

Definition 3.2.7 (Ground resolution calculus) The *ground resolution calculus* can be defined as the following transition relation

$$\mathcal{R} = \{\langle S, S \cup \{c\}\rangle \mid c = \mathfrak{R}(c_1, c_2) \text{ for some } c_1, c_2 \in S\}$$

where S ranges over finite sets of ground clauses.

Resolution presents a striking example for illustrating the distinction between the declarative and the procedural interpretation of a deduction. While a deduction of the former type is simply a sequence D of clauses where each element of D is derived from clauses in the input set or earlier elements of D, the deduction process consists of a sequence of increasing clause sets. If the deduction process is based on unrestricted resolution—which is free of reduction rules like *subsumption deletion* considered below—, then any state of the deduction process can be

[10] Only the following holds. For any ground clause c logically implied by a set of ground clauses S, there exists a ground resolution proof of a clause c' from S with $c' \models c$.

made into a deduction of the declarative type. This property holds for all calculi which are *accumulative*. In general, however, the states of a deduction process need not represent declarative deduction objects, even if no reduction rules are applied. The comparison also exhibits a certain weakness of measuring the size of a deduction as the sum of the sizes of the states in the deduction process, since *untouched* parts of the states are counted multiply, so that the static deduction object and the sum of the states in the deduction process may differ in size, though only up to a polynomial (viz., quadratic) difference. A finer model would count solely the touched parts of the non-initial states. It is important to emphasize, however, that in the transition relational framework it is almost impossible to make a *computationally reliable* distinction between touched and untouched parts of a state, since, for instance, sets cannot be directly represented on a computer. Questions of this type demand a more implementation-oriented model like the proof module framework. We shall not concentrate on such fine differences in this work. On the contrary, we shall exploit the polynomial relatedness of the deduction models and move back and forth between the models, depending on which one is the best-suited for a certain purpose.

3.2.3 The Indeterministic Power of Ground Resolution

As is the case for deductions in the traditional calculi mentioned in the previous section, ground resolution proofs (or dags) are of polynomial difficulty (see Definition 2.1.12 on p. 71), that is, for any given structure $\mathcal{S}$, it can be decided whether $\mathcal{S}$ is a ground resolution proof with cost polynomially bounded by the size of $\mathcal{S}$. Or, in terms of properties of transition relations, the ground resolution calculus is polynomially size-transparent. But ground resolution has the advantage over the traditional calculi that the cost for verifying a ground resolution proof (dag) from a set of clauses S is bounded by a polynomial of the size of S and the resolution inference steps of the proof (dag).

Proposition 3.2.6 *The ground resolution calculus $\mathcal{R}$ (Definition 3.2.7) is polynomially transparent.*

Proof Apparently, the transition relation $\mathcal{R}$ is polynomial time step-reliable. Since, for every given input S, the maximal length of clauses deducible by ground resolution from S is bounded by size(S), $\mathcal{R}$ is logp size step-reliable. Therefore, by Lemma 2.3.7, $\mathcal{R}$ is polynomially transparent. □

Concerning indeterministic power, however, propositional resolution is strictly weaker than the propositional fragments of the traditional systems, which are all in the same equivalence class of polynomial simulation. This is formally expressed in the following two Propositions 3.2.7 and 3.2.9. We use the static deduction model which is better suited for comparing the indeterministic powers of the systems.

Proposition 3.2.7 *There is no polynomial p such that for any validity proof T of a propositional formula F in the propositional Frege/Hilbert, natural deduction, or sequent system there exists a ground resolution refutation D of a set of clauses S which is an appropriate*[11] *translation of $\sim F$ such that* $\text{size}(D) < p\,(\text{size}(T))$.

The proof of this proposition is too difficult to be carried out here from scratch. We establish it by combining two results by Haken and Urquhart. In [Haken, 1985] the following proposition was demonstrated.

Proposition 3.2.8 (Intractability of resolution) *The ground resolution calculus is not polynomially bounded for sets of unsatisfiable ground clauses.*

Haken proved that there is an infinite class of unsatisfiable clause sets, the so-called *pigeon-hole class* which are specified in Example 3.2.2, and the smallest ground resolution refutation for any element of this class is of exponential size. In [Urquhart, 1987] it was demonstrated that there is a polynomial p such that, for any element S of this class, if $\sim F$ is an appropriate translation of S, then there exists a sequent proof T with end sequent $\twoheadrightarrow F$ with $\text{size}(T) < p\,(\text{size}(F))$. Both results together imply Proposition 3.2.7.

Example 3.2.2 The *pigeon-hole principle* asserts that n pigeons do not fit into $n-1$ holes, or, more formally, there is no total and injective mapping p with a domain of n and a range of $n-1$ elements. For any natural number $n > 1$, this principle can be formulated as an unsatisfiable clause set S_n consisting of the union of the following sets of propositional clauses where p_i^k denotes a nullary atom which asserts that p maps pigeon k to hole i:

$$\bigcup\{\neg p_i^k, \neg p_i^l\} \qquad 1 \le i < n, 1 \le k < l \le n \qquad \text{(injectivity of } p\text{)}$$

$$\bigcup\{p_1^k, \ldots, p_{n-1}^k\} \qquad 1 \le k \le n \qquad \text{(totality of } p\text{)}$$

The traditional proof systems, however, can polynomially simulate resolution. We consider a slightly more general proposition.

Proposition 3.2.9 *There is a polynomial p such that for any propositional resolution refutation of a set of clauses S there exists a validity proof T of a formula $\sim F$ in the propositional Frege/Hilbert, natural deduction, or sequent system with F being an appropriate translation of S such that* $\text{size}(T) < p\,(\text{size}(D))$.

A proof can be found in [Reckhow, 1976] (see also [Letz, 1993b] for a polynomial simulation of resolution in the intuitionistic tree sequent system). Another important difference of resolution from the traditional proof systems is the following.

[11] The notion of appropriateness is investigated in Reckhow's dissertation [Reckhow, 1976]. Briefly, the translation procedure should have a polynomial run time and preserve satisfiability and unsatisfiability. An example of an adequate translation is the so-called *definitional clause normal form transformation* mentioned in Chapter 1.

Proposition 3.2.10 *For unsatisfiable sets of ground clauses, tree ground resolution cannot polynomially simulate ground resolution.*

The result can be easily proven by using a definitional form transformation S_n of formulae of the structure $F_n \wedge \neg F_n$ where F_n is a formula of the shape $F_n = A_1 \leftrightarrow A_2 \leftrightarrow \cdots A_{n-1} \leftrightarrow A_n$ presented in Example 1.7.1. The S_n have polynomial resolution refutation *dags* but only exponential resolution refutation *trees* (see [Reckhow, 1976]).

Note The reason why dags may help to improve the efficiency of resolution is because they facilitate the multiple use of derived clauses as parent clauses, in a recursive manner, whereas in the tree format for every use of a clause as a parent clause, its entire *derivation* must be repeated.

3.2.4 The Resolution Proof Relation

The ground resolution calculus can be made into a strongly complete proof relation for verifying the unsatisfiability of finite sets of ground clauses by the following simple modification of the transition relation $\mathcal{R}$ from Definition 3.2.7.

Definition 3.2.8 (Ground resolution proof relation) The *ground resolution proof relation* is the following transition relation $\mathcal{R}' =$

$$\{\langle S, S \cup \{c\}\rangle \mid c = \mathfrak{R}(c_1, c_2) \text{ for some } c_1, c_2 \in S \text{ and } c \notin S\} \;\cup\; \{\langle S, \Omega\rangle \mid \emptyset \in S\}$$

where S ranges over finite sets of ground clauses and Ω is the success state of $\mathcal{R}'$.

Proposition 3.2.11 *Let Σ^+ be the set of all finite sets of unsatisfiable ground clauses, and Σ^- the set of all finite sets of satisfiable ground clauses. The ground resolution proof relation $\mathcal{R}'$ is a decision procedure for the input pair $\langle \Sigma^+, \Sigma^- \rangle$.*

Proof We have to prove soundness, noetherianness, and strong completeness of $\mathcal{R}'$ for the given input pair. The soundness of $\mathcal{R}'$ follows from Proposition 3.2.5. Since from any finite number of ground literals only finitely many ground clauses can be composed, from a given finite set of ground clauses only finitely many states are accessible in $\mathcal{R}'$. Now $\mathcal{R}'$ is accumulative and irreflexive, hence acyclic. Then, Proposition 2.2.1 (iii) yields the boundedness, and (i) the noetherianness of $\mathcal{R}'$. Finally, The strong completeness of $\mathcal{R}'$ can be recognized as follows. From the accumulativity of $\mathcal{R}'$ follows its semi-confluence and from Corollary 3.2.16 below its completeness, which together, by Proposition 2.4.2, entail the strong completeness of $\mathcal{R}'$. □

Unfortunately, the ground resolution proof relation is not suited as a propositional decision procedure, because a shortest derivation from an input state may extremely differ from a longest derivation, and typically most derivations are much longer than a shortest one. This striking discrepancy is the motivation for the development of refinements of the calculus.

3.2.5 The Davis/Putnam Calculus

The *Davis/Putnam calculus* is a refinement and modularization of ground resolution and consitutes the kernel of the *Davis/Putnam procedure* which was introduced in [Davis and Putnam, 1960] *before* Robinson's resolution article [Robinson, 1965a] was published. The working with this system has two advantages. First, it admits a particularly elegant completeness proof. Second, the system is one of the most *successful* decision procedures for propositional and ground formulae, in contrast to the original ground resolution calculus.

Note There is an unpleasant systematic incorrectness in the literature. In most textbooks on automated deduction, the name 'Davis/Putnam procedure' is assigned to a proof procedure presented two years later by Davis, Logemann, and Loveland in [Davis et al., 1962]. Although the latter procedure is constructed from the original one by a simple modification, this modification concerns the kernel of the Davis/Putnam procedure, so that both procedures significantly differ from each other, even with respect to indeterministic power. Conceptually, the latter procedure is a variant of semantic trees, which are discussed below.

The Davis/Putnam calculus works by the *replacement* of clauses with other clauses.

Definition 3.2.9 (Clause replacement) Let L be a literal in a clause c of a set of ground clauses S. Assume further $c_1, \ldots, c_n$ are the clauses in S containing the literal $\sim L$ and not the literal L. The set R of all non-tautological resolvents of c and c_i over L, $1 \leq i \leq n$, is called the *clause replacement of* c *by* L in S.

Lemma 3.2.12 *Let L be a literal in a clause c of a set of ground clauses S, and assume R is the clause replacement of c by L in S. If S is unsatisfiable, then $(S \setminus \{c\}) \cup R$ is unsatisfiable.*

Proof Let L be a literal in a clause c of an unsatisfiable set of ground clauses S. Consider the set of Herbrand interpretations $\mathcal{H}_c$ which exclusively falsify the clause c. If $\mathcal{H}_c$ is empty, $S \setminus \{c\}$ must be unsatisfiable, and we are done trivially. Otherwise, consider an arbitrary Herbrand interpretation $\mathcal{H}$ in $\mathcal{H}_c$. Evidently, the interpretation $\mathcal{H}' = (\mathcal{H} \setminus \{\sim L\}) \cup \{L\}$ is a model for c. Hence $\mathcal{H}'$ must falsify another non-tautological clause d in S. Since $\mathcal{H}$ and $\mathcal{H}'$ differ only in the elements $\sim L$ and L respectively, the assumptions that $\mathcal{H}$ is a model for d and $\mathcal{H}$ not entail that d must contain the literal $\sim L$. Now, the Herbrand interpretation $\mathcal{H}$ falsifies each of the clauses $c \setminus \{L\}$ and $d \setminus \{\sim L\}$, hence also their union, which, being a non-tautological ground resolvent of c and d over L, is an element of the clause replacement R of c by L in S. Since $\mathcal{H}$ was chosen arbitrary, every interpretation in $\mathcal{H}_c$ falsifies an element of R. Therefore, the set $(S \setminus \{c\}) \cup R$ must be unsatisfiable. □

The proof of the strong completeness of the Davis/Putnam calculus is based on the following obvious fact.

Lemma 3.2.13 *Let L be a literal in a clause c in a set of ground clauses S, and R the clause replacement of c by L in S. Then, the number of clauses in $(S \setminus \{c\}) \cup R$ containing the literal L is by one less than the number of clauses in S containing the literal L.* □

This lemma shows how to get rid of *all* clauses containing a certain literal.

Definition 3.2.10 (L-clauses replacement) Let S be a set of ground clauses and S^L the set of clauses in S containing the literal L. The union of all clause replacements of any clause in S^L by L in S is called the *replacement* of *L-clauses in S*.

Lemma 3.2.14 (Literal elimination) *Let S be a set of ground clauses, S^L the set of clauses in S containing the literal L, and R^L the replacement of L-clauses in S. If S is unsatisfiable, then $(S \setminus S^L) \cup R^L$ is unsatisfiable.*

Proof It suffices to recognize that a literal elimination step with a literal L produces the same result as an iterative substitution of L-clauses with their respective clause replacements. □

Although, in general, the number of occurrences of other literals as well as the size of the formula may increase, the literal elimination step is the kernel of one of the most natural resolution-based decision procedures for ground formulae.

Definition 3.2.11 (Davis/Putnam calculus) The *Davis/Putnam calculus* can be defined as the following transition relation

$$\mathcal{R} = \{\langle S, (S \setminus S^L) \cup R^L \rangle \mid L \text{ is contained in some clause of } S\}$$

where S ranges over finite sets of ground clauses, S^L is the set of clauses in S containing the literal L, R^L is the replacement of L-clauses in S, and $\{\emptyset\}$ is the success state of $\mathcal{R}$.

Note The original way of presenting a transition step in the Davis/Putnam calculus [Davis and Putnam, 1960] is slightly more indirect. First, the given set of clauses S is transformed into the logically equivalent set S' of formulae

$$\{A \vee L, B \vee {\sim}L\} \cup R$$

where A is the conjunction of clause formulae $\rfloor L_1, \ldots, L_n \lfloor$ not containing L and with a clause $\{L_1, \ldots, L_n, L\} \in S$, B is the conjunction of clause formulae $\rfloor L_1, \ldots, L_n \lfloor$ not containing ${\sim}L$ and with a clause $\{L_1, \ldots, L_n, {\sim}L\} \in S$, and R is the set of clauses from S neither containing L nor ${\sim}L$. Apparently, S' is unsatisfiable if and only if $S'' = \{A \vee B\} \cup R$ is unsatisfiable. Afterwards, S'' is translated into clausal form by applying the standard transformation given in the proof of Proposition 1.7.14 on p. 63.[12] The resulting set of clauses is exactly the subsequent state in the Davis/Putnam calculus given above.

Now we have collected the material for an easy proof of the completeness of ground resolution.

Proposition 3.2.15 (Decision property of the Davis/Putnam calculus) *Let Σ^+ be the set of all finite sets of unsatisfiable ground clauses, and Σ^- the set of all finite sets of satisfiable ground clauses. The Davis/Putnam calculus $\mathcal{R}$ is a decision procedure for the input pair $\langle \Sigma^+, \Sigma^- \rangle$.*

Proof In each iteration the number of distinct literals contained in clauses of the formula decreases by 1, while satisfiability and unsatisfiability are preserved. Consequently, after finitely many transition steps a set S' of clauses is reached which is void of literals. If the initial set S was satisfiable, due to the soundness of ground resolution, $S' = \{\}$. If, on the other hand, S was unsatisfiable, by the Literal Elimination Lemma 3.2.14, $S' = \{\emptyset\}$. □

Corollary 3.2.16 (Completeness of ground resolution) *For any unsatisfiable set S of ground clauses there is a ground resolution refutation of S.*

Proof Given an input set S, the sequence of resolvents generated in any maximal Davis/Putnam derivation, in the order of their generation, is a ground resolution refutation of S. □

The transition relation of the Davis/Putnam calculus has some interesting properties.

Proposition 3.2.17 *The Davis/Putnam proof relation $\mathcal{R}$ is polynomially step-bounded, but not polynomially bounded and not polynomially transparent.*

Proof If n is the number of distinct literals contained in clauses of an unsatisfiable input set S, then clearly the success state is reached within n steps from the initial state S. Since the ground resolution calculus is not polynomially bounded (Proposition 3.2.8), and the indeterministic power of the Davis/Putnam calculus is not greater than that of resolution, $\mathcal{R}$ is not polynomially size-bounded, hence not polynomially bounded. Then, the polynomial intransparency of $\mathcal{R}$ follows from Proposition 2.3.4. □

Note The Davis/Putnam calculus is a good example of a proof relation in which the notion of what has to be counted as a single inference step is not acceptable.

[12] Note that the use of a definitional transformation procedure is not permitted here, because it may introduce new propositional atoms, with the result that the Davis/Putnam procedure may never terminate.

Apparently, if we would succeed in making the proof relation polynomially transparent *and* preserving the distances between input states and normal forms of the transition relation, then we would have solved the P/NP problem.

For the presentation of the Davis/Putnam *procedure*, some additional terminology is needed.

Definition 3.2.12 (Literal occurrence) If a literal L is contained in a clause c, then the pair $\langle L, c \rangle$, written L_c, is called a *literal occurrence of* L *in* S.

Definition 3.2.13 (Ground purity) Let L be a literal in a clause c of a set of clauses S. The literal occurrence L_c is called

1. *strongly ground pure in* S if the literal $\sim L$ is not contained in a clause of S,
2. *ground pure in* S if the literal $\sim L$ is not contained in another clause of S,
3. *weakly ground pure in* S if the clause replacement of c by L in S is empty.

Note All three versions of purity have been used in the literature. The standard version, which has been introduced in [Robinson, 1965a], is the second one.

Proposition 3.2.18 (Ground purity deletion) *Let* L *be a literal in a clause* c *of an unsatisfiable set of clauses* S*. If* L_c *is strongly ground pure, ground pure, or weakly ground pure in* S*, then* $S \setminus \{c\}$ *is unsatisfiable.*

Proof It suffices to prove the latter case, which is an immediate consequence of the Clause Replacement Lemma 3.2.12 on p. 100. □

It is clear that from the perspective of optimal reduction the third version of purity is the best.

Definition 3.2.14 (Ground subsumption) Given two ground clauses c_1 and c_2. We say that c_1 *(properly) ground subsumes* c_2 if c_1 is a (proper) subset of c_2.

Properly subsumed clauses may be deleted, due to the following fact.

Proposition 3.2.19 (Ground subsumption deletion) *If a clause* c *is properly ground subsumed by a clause in a set* S *of ground clauses, then* $S \equiv (S \setminus \{c\})$.

Proof Clearly every model for S is also a model for $S \setminus \{c\}$. For the converse, note that, by assumption, there is a clause $c' \in S$ with $c' \subset c$. We show that $c' \models c$. Let $\mathcal{I}$ be an arbitrary model for c'. Its assignment $\mathfrak{I}$ maps some literal $L \in c'$ to $\top$. Since $L \in c$, $\mathfrak{I}(c) = \top$, and hence, $\mathcal{I}$ is a model for c. □

The Davis/Putnam procedure with subsumption can be defined as the following complex transition relation.

Definition 3.2.15 (Davis/Putnam procedure with subsumption) Let S be any finite set of ground clauses, S^L the set of clauses in S containing L, and R^L the replacement of L-clauses in S. The *Davis/Putnam procedure with subsumption* $\mathcal{R}'$ is the union of the following binary relations:

1. $\{\langle S, S' \rangle \mid S \neq S'$ and S' is the subset of clauses in S which are not properly ground subsumed in $S\}$,
2. $\{\langle S, S' \rangle \mid S \neq S'$ and S' is the subset of clauses in S which contain no literal occurrences which are ground pure in $S\}$,
3. $\{\langle S, (S \setminus S^L) \cup R^L \rangle \mid S$ contains neither properly subsumed clauses nor ground pure literal occurrences, and L is any literal in a clause of minimal length in $S\}$.

The success state of the proof relation is $\{\emptyset\}$.

Note In the original Davis/Putnam procedure [Davis and Putnam, 1960] only those properly subsumed clauses are deleted with are subsumed by unit clauses. Apparently, there is no reasonable motivation for such a restriction.

Proposition 3.2.20 (Decision property of the Davis/Putnam procedure) *Let Σ^+ be the set of all finite sets of unsatisfiable ground clauses, and Σ^- the set of all finite sets of satisfiable ground clauses. The Davis/Putnam procedure (with subsumption) is a decision procedure for the input pair $\langle \Sigma^+, \Sigma^- \rangle$.*

Proof In analogy to the proof of Proposition 3.2.15 on p. 102. □

3.2.6 Other Resolution Refinements

There are various other refinements of resolution. We consider here *linear ground resolution* and *regular ground resolution. Linear resolution* was introduced simultaneously by Loveland [Loveland, 1969] and Luckham [Luckham, 1970].

Definition 3.2.16 (Linear ground resolution proof) A ground resolution proof $D = (c_1, \ldots, c_n)$ from a set S of ground clauses is called *linear* if, for each $c_i, 1 < i \leq n$, one of the parent clauses of c_i is c_{i-1}.

Definition 3.2.17 (Linear ground resolution dag) A ground resolution dag is called *linear* if all inner nodes of the dag lie on the same branch.

The resolution proof given in Example 3.2.1 on p. 93 and the resolution dag depicted in Figure 3.1 on p. 94 are both linear. The most natural way of defining the linear ground resolution *calculus* is by using pairs consisting of the input set and the current linear deduction.

Definition 3.2.18 (Linear ground resolution calculus) The *linear ground resolution calculus* can be defined as the transition relation

$$\{\langle\langle S, (c_1, \ldots, c_m)\rangle, \langle S, (c_1, \ldots, c_m, c_{m+1})\rangle\rangle \mid \text{ for } m \geq 1\text{: } c_m \text{ is a parent of } c_{m+1}\}$$

where the states in the transition relation are pairs $\langle S, D\rangle$ consisting of a set S of ground clauses and a ground resolution proof D from S.

The linearity refinement is mainly interesting for *first-order* resolution. Linear *ground* resolution is unsuited as a calculus for ground formulae, because of the following obvious property.

Proposition 3.2.21 *The linear ground resolution calculus is not proof-confluent for input pairs $\langle \Sigma^+, \Sigma^-\rangle$ with Σ^+ being the set of all pairs $\langle S, ()\rangle$ with S being a finite unsatisfiable set of ground formulae.*

This has as a consequence that strong completeness can only be achieved by means of deduction *enumeration*. Such an approach turns out to be not optimal for the propositional case, since here a calculus is typically used as a decision procedure, whereas in the first-order case one can only demand semi-noetherianness.

Regular resolution [Tseitin, 1970] is at best defined using the dag framework.

Definition 3.2.19 (Regular branch) A branch $b = (e_1, \ldots, e_n)$ in a ground resolution dag is called *regular* if no two edges in b are labelled with the same literal.

Definition 3.2.20 (Regular ground resolution dag) A ground resolution dag is called *regular* if every branch is regular, and *semi-regular* if for every leaf node N there exists a regular branch terminating in N.

The importance of regular resolution derives from the following fact.

Proposition 3.2.22 *For unsatisfiable sets of ground clauses, regular ground resolution can polynomially simulate the Davis/Putnam calculus and the Davis/Putnam procedure (with subsumption).*

Proof Given a resolution proof D constructed from a deduction in one of the Davis/Putnam systems, than any resolution dag resulting from applying Procedure 3.2.1 to D is regular. □

In [Tseitin, 1970] the intractability of regular ground resolution was proven. Moreover, Tseitin showed that ground tree resolution cannot polynomially simulate regular ground resolution. A recent interesting result concerning the relation between regular and unrestricted resolution is the following.

Proposition 3.2.23 *For unsatisfiable sets of ground clauses, regular ground resolution (and hence any-one of the Davis/Putnam systems) cannot polynomially simulate ground resolution.*

This result was proven by Goerdt in [Goerdt, 1989]. He used a class of formulae which are modifications of the pigeon-hole class and have polynomial proofs in unrestricted resolution but only exponential proofs in regular resolution.

3.2.7 Semantic Trees

The most primitive approach to determining the satisfiability status of a propositional formula is the truth table method. Starting off from the model theory of logic, all interpretations for the atoms in the formula are listed as lines in a table, which afterwards are examined, one after the other. If n is the number of atoms occurring in the formula, an evaluated truth table 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. If the truth value $\top$ does not occur in the last column of the evaluated table, the formula is unsatisfiable, and vice versa. While, for each interpretation, the truth value of the formula can be computed from the value of its atoms in polynomial time, the obvious problem is the number of lines, which is exponential with respect to the number of atoms occurring in the formula. Consequently, if the ratio between the lengths of formulae in an infinite collection and their numbers of atoms is less than 2^n—which is the case for almost all interesting formula classes—then the truth table method has exponential complexity with respect to the formula sizes in the class. Truth tables are not suited as a basis for propositional calculi, for two reasons. First, since all truth tables for a formula have equal size, there is no difference between worst-case and best-case behaviour, and hence no potential for heuristic support. Second, since the size and the structure of the formula has no influence on the size of the truth table, the efficiency of the method cannot be improved by manipulations and accumulations of the input formula, which are the techniques for rendering proof systems more powerful.

A natural improvement of truth tables—and one of the most promising frameworks for propositional calculi—are semantic trees. Semantic trees were applied in [Robinson, 1968, Kowalski and Hayes, 1969], 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. 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 functional 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.

Semantic trees can be introduced as manipulating ground clauses or ground clause formulae. To keep closest proximity to resolution systems, we choose the version for clauses. First, we define the deduction *objects*.

Definition 3.2.21 (Semantic tree) A *semantic tree for* a set of ground 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 leading out from the same vertex is labelled with an atom p occurring in S and its negation $\neg p$, respectively.
2. Any leaf node N may be labelled with a clause from S, provided that all literals in the clause 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 clause from S.

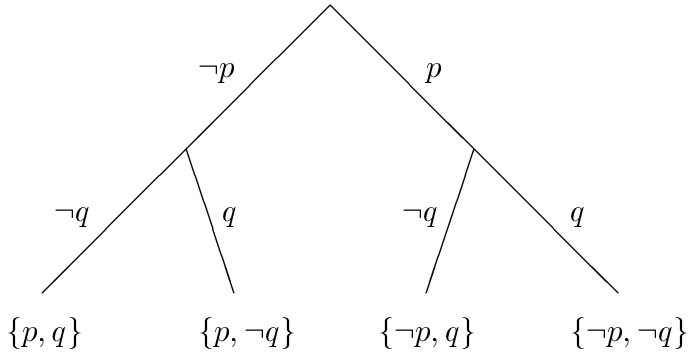


Figure 3.2: Closed semantic tree for $\{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}\}$.

An example of a closed semantic tree is depicted in Figure 3.2; we display semantic trees as downward trees, in order to conform with the construction methodology of semantic tree *calculi*.

There is the following close relationship between semantic trees and resolution.

Proposition 3.2.24 *Every ground resolution tree for a set of clauses S is a semantic tree for S, provided the labellings of the internal nodes are disregarded.*

Conversely, any closed semantic tree can be made into a resolution refutation without increase in size, according to the following procedure.

Transformation from semantic trees to resolution trees Let T be a closed semantic tree for a set of ground clauses S. Construct a ground resolution refutation of S, by performing the following procedure on S.

> First, select any unlabelled node N of the current tree whose sucessors are both labelled with ground clauses c_1 and c_2. If the non-tautological ground resolvent c of c_1 and c_2 exists, label N with c. Otherwise, prune the tree by connecting the edge incident to N to one of its successors N_1 or N_2. Then, iterate the procedure with the resulting tree.

It is clear that the procedure generates a ground tree resolution refutation of S with a size equal or smaller than the initial semantic tree. Consequently, concerning indeterministic power, semantic trees and tree resolution are equivalent proof systems.

Proposition 3.2.25 *For sets of ground clauses, semantic trees and tree resolution polynomially simulate each other.*

Since ground tree resolution cannot polynomially simulate unrestricted ground resolution (Proposition 3.2.10 on p. 99), we can immediately infer the following corollary.

Corollary 3.2.26 *For sets of ground clauses, semantic trees cannot polynomially simulate ground resolution.*

Definition 3.2.22 A semantic tree is called *regular* if no atom occurs more than once on a branch.[13]

Again, the notational relation with resolution is preserved.

Proposition 3.2.27 *Every regular ground resolution tree is a regular semantic tree, provided the leaf vertices are disregarded.*

The regularity restriction on semantic trees is very reasonable, which is motivated by the following obvious proposition.

Proposition 3.2.28 *Any smallest closed semantic tree for a set of clauses S is regular.*

Consequently, in contrast to resolution where regularity is a proper restriction when concerning lengths of shortest proofs, imposing regularity on semantic trees has no disadvantages. Apparently, the resulting calculus is a decision procedure for sets of ground clauses.

Note As recent experiments have shown [Buro and Kleine Büning, 1992], the most successful logic-based computer programs for deciding the satisfiability status of ground formulae are variants of regular semantic trees.

An interesting simulation possibility concerning *linear* ground resolution is expressed in the following proposition.

[13] Normally, the regularity restriction is already included in the semantic tree definition. For systematic and terminological reasons, we have left the condition outside.

Proposition 3.2.29 *For sets of ground clauses, linear resolution can linearly simulate (regular) tree resolution and (regular) semantic trees.*

In order to proof this result we make use of the following two lemmata.

Notation 3.2.1 If T is a dag with its nodes labelled with clauses, then we shall denote with $\mathcal{S}_T$ the set of clauses appearing at the nodes of T, and with $\mathcal{L}_T$ the set of clauses appearing at the leaf nodes of T.

Lemma 3.2.30 *A regular ground resolution tree T of the shape specified in Figure 3.3 with $K \notin c_1$ satisfies the following two properties.*

(i) No complement of a literal in $\{{\sim}K\} \cup c_1$ is contained in a clause of $\mathcal{S}_T$.

(ii) Some of the clauses in $\mathcal{L}_T$ contain the literal ${\sim}K$.

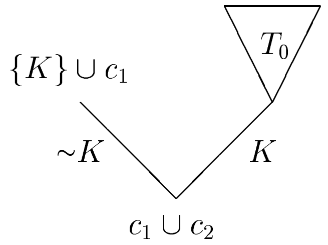


Figure 3.3: Resolution tree T with leaf and neighbouring subtree T_0.

Proof The first fact is due to the regularity of T, the second because T is a *resolution* tree—and not a general semantic tree. □

Lemma 3.2.31 *Suppose T to be a regular ground resolution tree of the shape specified in Figure 3.3 with $K \notin c_1$. Let, furthermore, r^+ and r^- be clauses such that no complement of a literal in $r^+ \cup r^-$ is contained in a clause of $\mathcal{S}_T$. Set $c' = r^+ \cup ((\{K\} \cup c_1) \setminus r^-)$. Then, there is a subset r' of r^- and a linear ground resolution dag $T^\star$ with bottom clause $c = r^+ \cup ((c_1 \cup c_2) \setminus r')$ for the set of ground clauses $\mathcal{L}_{T_0} \cup \{c'\}$ with* $\text{steps}(T^\star) \leq 2 \times \text{steps}(T)$.

Proof The proof is by induction on the depth of the subtree T_0. The induction base $\text{depth}(T_0) = 1$ is trivial. For the induction step, assume the result to hold for any regular ground resolution tree with $\text{depth}(T_0) < n$, for its subtree T_0. Let T be such a tree with $\text{depth}(T_0) = n$, and assume r^+ and r^- as described. By Lemma 3.2.30, there exists a leaf node N in T_0 with neighbouring subtree T_1 such that N is labelled by a clause $\{L\} \cup d_1$ with ${\sim}K \in d_1$ and $L \neq {\sim}K$; also, no complement of a literal in $\{K\} \cup c_1$ is contained in a clause of $\mathcal{S}_{T_0}$ (consult the left tree in Figure 3.4 as an illustration). Set $d' = r^+ \cup \{L\} \cup c_1 \cup (d_1 \setminus \{{\sim}K\} \setminus r^-)$. By the induction assumption, there is a subset r_1' of $\{{\sim}K\} \cup r'$ and a linear ground resolution dag T' with bottom clause $d = r^+ \cup c_1 \cup ((d_1 \cup d_2) \setminus r_1')$ for the set $S_{T_1} \cup d'$ with $\text{steps}(T') \leq 2 \times (\text{steps}(T_1) + 1)$. If m is the distance in T between the node labelled with $d_1 \cup d_2$ and the node labelled with $\{\sim K\} \cup c_2$, then m further applications of the induction assumption yield the existence of a subset r'_m of r'_1 and a linear ground resolution dag T'' with bottom clause $d' = r^+ \cup c_1 \cup (c_2 \setminus r'_m)$ for the set of clauses $(S_{T_0} \setminus \{\{L\} \cup d_1\}) \cup d'$ with steps$(T'') \leq 2 \times$ steps(T_0). Modify the leaf of T'' labelled with d' by attaching two parent nodes labelled with the clauses $c' = r^+ \cup \{K\} \cup (c_1 \setminus r^-)$ and and $\{L\} \cup d_1$; note that d' is their ground resolvent. Two cases need to be distinguished. Either, $\sim K \notin d'$ and we are done. Or, $\sim K \in d'$; in this case an additional (*ancestor*) resolution step with d' and far parent clause c' yields the desired linear resolution dag $T^\star$. In either case steps$(T^\star) \leq 2 \times$ steps(T). □

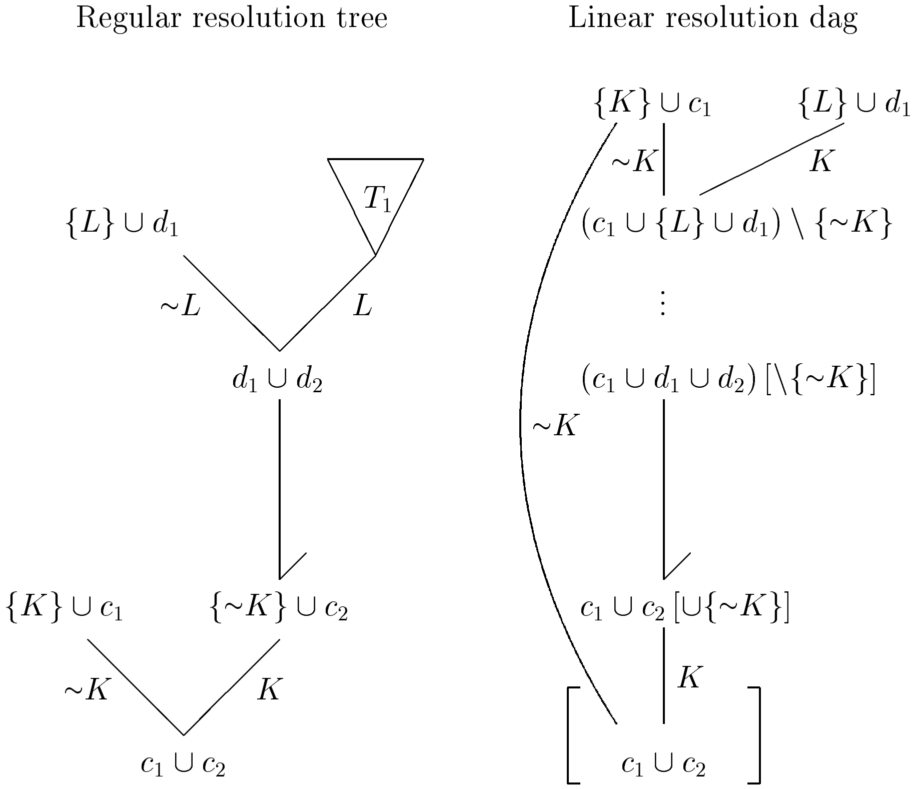


Figure 3.4: Simulation of regular tree resolution by linear resolution.

Proof of Proposition 3.2.29 It suffices to prove that linear ground resolution can linearly simulate regular ground resolution trees. Let, therefore, T be a closed regular ground resolution refutation tree for a set of ground clauses S. From T one can construct a linear ground resolution refutation dag $T^\star$ for S as follows. Choose any clause $\{K\} \cup c_1$ at a leaf node of a branch with length m in T as start clause. Then, m successive application of Lemma 3.2.31, always putting $r^+ = r^-$, guarantee the existence of the desired linear refutation $T^\star$ with steps$(T^\star) \leq 2 \times$ steps(T). A single iteration of the process is schematically displayed in Figure 3.4. □

3.3 Tableau and Connection Calculi

In this section two families of logic calculi are discussed, tableau and connection calculi. These families are closely related, in the following two respects. First, both methodologies are working in a backward (goal-oriented) manner, like semantic tree procedures. Secondly, the basic systems of both families are cut-free. Tableau and connection calculi are not optimally suited as decision procedures for ground formulae, their real applicability is on the first-order level. Accordingly, some of the notions developed in this section gain their actual importance when first-order formulae are considered.

3.3.1 The Tableau System

The *tableau calculus* was introduced by Beth in [Beth, 1955, Beth, 1959] and elaborated by Hintikka in [Hintikka, 1955] and Smullyan in [Smullyan, 1968]. Similar to resolution, the tableau calculus is a refutational system, that is, the method demonstrates the validity of a formula F by proving the inconsistency of its negation $\neg F$. In contrast to resolution, however, the tableau calculus can be used to show the inconsistency of ordinary first-order formulae. We shall take Smullyan's *analytic tableaux* as standard reference system. The analytic tableau method proceeds by constructing a tree with its nodes labelled by *subformulae*—in a sense defined immediately—occurring in the input formula, therefore the epithet 'analytic'. The standard tableau calculus is restricted to ordinary logical formulae. Since such a restriction is unnecessary, we introduce a system which is a variant of the standard tableau calculus, extended to the handling of *finite sets* of *general* formulae, including general conjunctions and disjunctions. A further notational difference is that in the standard method the root node of the tableau is labelled with the respective input formula, whereas we prefer to keep the input alongside the tableau. This way of designing the calculus has some advantages concerning presentation and the formulation of complexity issues.

Conjunctive		Disjunctive	
α	α-subformulae sequence	β	β-subformulae sequence
$\neg\neg F$	(F)		
$F \wedge G$	(F, G)	$\neg(F \wedge G)$	$(\neg F, \neg G)$
$\neg(F \vee G)$	$(\neg F, \neg G)$	$F \vee G$	(F, G)
$\neg(F \rightarrow G)$	$(F, \neg G)$	$F \rightarrow G$	$(\neg F, G)$
$F \leftrightarrow G$	$(F \rightarrow G, G \rightarrow F)$	$\neg(F \leftrightarrow G)$	$(\neg(F \rightarrow G), \neg(G \rightarrow F))$
$\neg -$	$(\top)$	$\neg\top$	$(-)$
$\rceil F_1, \ldots, F_n \lceil$	$(F_1, \ldots, F_n)$	$\neg\rceil F_1, \ldots, F_n \lceil$	$(\neg F_1, \ldots, \neg F_n)$
$\neg\rfloor F_1, \ldots, F_n \lfloor$	$(\neg F_1, \ldots, \neg F_n)$	$\rfloor F_1, \ldots, F_n \lfloor$	$(F_1, \ldots, F_n)$

Figure 3.5: Syntactic types and α-, β-subformulae of ground formulae.

The ground tableau method is based on the fact that all ground formulae which are no literals or nullary connectives can be partitioned into two syntactic types, a *conjunctive type*, called the *α-type*, and a *disjunctive type*, named the *β-type*; to any formula F of any type (α or β) a certain sequence of formulae different from F can be assigned, called the α- or *β-subformulae sequence* of F depending on the type of F, as defined in Figure 3.5 (with assuming $n \geq 1$).

Proposition 3.3.1 *A formula of the conjunctive type is logically equivalent to the* conjunction *of its α-subformulae, whereas a formula of the disjunctive type is logically equivalent to the* disjunction *of its β-subformulae.*

Although an α- or β-subformula of a given complex formula F is not always a *proper* subformulae of F, in the standard sense defined in Chapter 1, the kind of “subformula” relation defined here shares the following important property with the standard immediate subformula relation.

Proposition 3.3.2 *The transitive closure $\prec$ of the union of the α- and β-subformula relations is well-founded on the collection of ground formulae, i.e., there are no infinite decomposition sequences. Moreover, the minimal elements in the relation $\prec$ are literals or nullary connectives.*

We begin with the definition of the proof objects generated by the tableau calculus.

Definition 3.3.1 (Unary formula) A non-literal ground formula is called *unary* if its α- or β-subformulae sequence is unary.

Definition 3.3.2 (Tableau) A *tableau T for* a finite set S of general ground formulae is a pair $\langle t, \lambda \rangle$ consisting of an ordered tree t and a labelling function λ on its non-root nodes such that for any non-leaf node N:

1. if N has exactly one successor node N_1 labelled with a formula F_1, then
   - either $F_1 \in S$,
   - or F_1 is an α-subformula or the β-subformula of a unary β-formula F which is contained in S or appearing on the branch from the root of T up to N,
2. if N has successor nodes $N_1, \ldots, N_n$, $n > 1$, labelled with formulae $F_1, \ldots, F_n$, respectively, then $(F_1, \ldots, F_n)$ is the β-subformulae sequence of a formula F which is contained in S or appearing on the branch from the root of T up to N.

Definition 3.3.3 (Closed tableau) If on a branch in a tableau appears $-$ or a formula and its negation, then the branch and its leaf node are called *closed*; otherwise the branch and its leaf node are termed *open*. A tableau is called *closed* if every branch is closed; otherwise the tableau is said to be *open*. If on a branch in a tableau appears $-$ or an atom and its negation, then the branch is called *atomically closed*; a tableau is called *atomically closed* if every branch is atomically closed. Two nodes N_1 and N_2 with labels F_1 and F_2 in a tableau are called *complementary* or *connected* if $F_1 = \neg F_2$ or $\neg F_1 = F_2$.

In Figure 3.6 a closed tableau for a set of clause formulae is depicted. Like semantic trees, tableaux are displayed as downward trees; normally, we do not display the label of the root node.

Clause formulae

c_1 : ⌋p⌊

c_2 : ⌋$r, \neg p, q$⌊

c_3 : ⌋$s, \neg q$⌊

c_4 : ⌋$\neg q, \neg s$⌊

c_5 : ⌋$\neg q, \neg r$⌊

c_6 : ⌋$\neg r, q$⌊

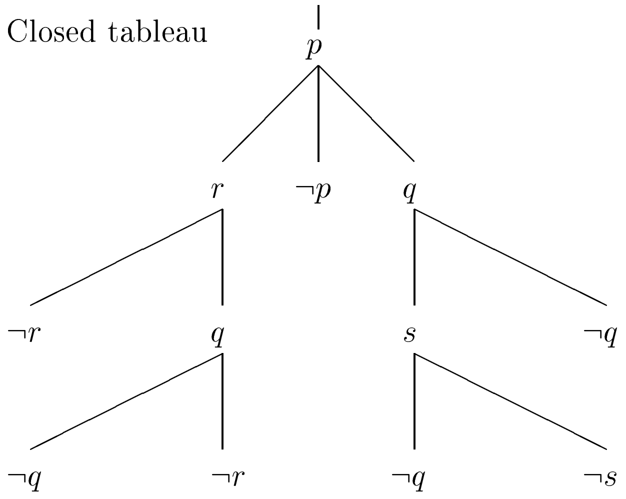


Figure 3.6: Closed tableau for a set of clause formulae.

One interpretation of a tableau is to take any branch as the conjunction of the formulae appearing on it, and the tableau itself as the disjunction of its branches. Under this reading, a closed tableau represents the logical falsum. But, unlike an unsatisfiable set of ground formulae, a closed tableau represents an explicit form of unsatisfiability, in the sense that it can be verified in polynomial time, by just checking each branch of the tableau. In other terms, the set of closed tableaux is of polynomial difficulty (according to Definition 2.1.12).

Proposition 3.3.3 (Tableau soundness and completeness) *A set S of general ground formulae is unsatisfiable if and only if there exists a closed tableau for S.*

The soundness is immediate from Proposition 3.3.1, for a completeness proof of tableaux for general formulae, we refer to [Smullyan, 1968], the completeness for clause formulae will be proved below (Theorem 3.4.6).

Similar to semantic trees, the regularity condition can be defined for tableaux.

Definition 3.3.4 (Regular tableau) A tableau is regular if no two nodes on a branch are labelled with the same formula.

3.3.2 The Tableau Calculus

Like resolution dags or semantic trees do not prescribe the precise order according to which they have to be generated or worked off, there are different possibilities how tableaux can be constructed in sequences of inference steps. The tableau *calculus* introduced now describes the standard *top-down* methodology of building up tableaux. In advance, we introduce the notion of marked tableaux.

Definition 3.3.5 (Marked tableau) A *marked tableau* is a pair $\langle T, \mu \rangle$ consisting of a tableau T and a partial labelling function μ from the set of leaf nodes of T into the set of nodes of the tableau. A (branch with) leaf node N of a marked tableau is called *marked* if $N \in \text{domain}(\mu)$. A marked tableau is named *marked as closed* if all of its leaves are marked.

The ground tableau calculus consists of the following two inference rules.

Procedure 3.3.1 (Tableau expansion) Given a set S of general formulae as input and a marked tableau for S, choose a leaf node N which is not marked, and:

1. either select a formula $F \in S$, expand the tableau at the node N with a new node, and label it with F,
2. or select a formula F from S or appearing on the branch from the root up to N, and
   - if F is of type α, then expand the tableau at the node N with a new node, and label it with an α-subformula of F,
   - otherwise F is of type β with the β-subformulae sequence $(F_1, \ldots, F_n)$, in this case attach n new successor nodes $N_1, \ldots, N_n$ to N, and label them with $F_1, \ldots, F_n$ respectively.

Regular tableau expansion is defined in the same way except that the expanded tableau need to be regular.

Procedure 3.3.2 (Tableau reduction) Given a marked tableau T, choose an unmarked leaf node N with literal L, select a dominating node N' with literal $\sim L$, and mark N with N'.

With the marking of a leaf node it is noted explicitly that the respective branch has been checked off as being closed and need not be further expanded.

Definition 3.3.6 (Tableau calculus) The *(ground) tableau calculus* can be defined as the transition relation

$$\{\langle\langle S,T\rangle,\langle S,T'\rangle\rangle \mid T' \text{ is obtained from } T \text{ by an expansion or reduction step}\}$$

$$\cup\ \{\langle\langle T_C\rangle,\Omega\rangle \mid T_C \text{ is a tableau which is marked as closed}\}$$

where S ranges over finite sets of ground formulae, T and T' are marked tableaux for S, and Ω is the success state of the transition relation. The *regular* version results from replacing expansion with regular expansion.

It is apparent that any tableau can be obtained by applying the top-down rules of the tableau calculus, and conversely.

Proposition 3.3.4 *Given an input pair $\Sigma = \langle\Sigma^+,\Sigma^-\rangle$ where Σ^+ is the set of finite sets of unsatisfiable ground formulae. The (regular) tableau calculus is*

(i) finitely branching,

(ii) polynomially transparent, and

(iii) proof-confluent for Σ.

(iv) Additionally, the regular version is a decision procedure for Σ.

Proof While the finite branching rate (i) is evident, the polynomial transparency (ii) follows from the polynomial time and logp size step-reliability of the tableau calculus. Proof-confluence (iii) follows from completeness and from the fact that *any* tableau can be completed to a closed one if the input set is unsatisfiable. The decision property (iv) for the regular case follows from (iii), the soundness, and the noetherianness of the regular tableau calculus; the latter is a consequence of the regularity condition and the well-foundedness of the transitive closure of the α- and β-subformula relations (Proposition 3.3.2). □

3.3.3 The Indeterministic Power of Tableaux

There is a close relation between tableau and sequent systems. Tableau calculi are often viewed as backward variants of sequent calculi. Like for resolution, the tableau system cannot polynomially simulate the propositional sequent system *with* cut. In fact, the tableau system cannot even polynomially simulate truth tables (and hence, semantic trees).

Proposition 3.3.5 *There is no polynomial p such that for any given finite unsatisfiable set $S = \{F_1,\ldots,F_n\}$ of propositional or ground formula there exists a closed tableaux T for S with* $\text{size}(T) < p\,(n)$ *where n is the number of lines of a complete truth table for $F_1 \wedge \cdots \wedge F_n$; and conversely.*

Proof Consider the class of formula sets given in Example 3.3.1. A truth table for the equivalent of an S_n has 2^n lines. Any minimal closed tableau for an S_n has the tree structure shown in Figure 3.7, for $n = 3$. Therefore, taking the number of unclosed nodes in such a tableau T_n as a lower bound of its size, we get that

$$\text{size}(T_n) > \sum_{i=1}^{n} \prod_{j=i}^{n} j \geq n!$$

while the size of S_n—we take the number of atom occurrences in a propositional formula as the size of the formula—is $n \times 2^n$. The converse result is trivial. □

Example 3.3.1 For any set $\{A_1, \ldots, A_n\}$ of distinct propositional atoms, let S_n denote the set of all 2^n multiple disjunctions of the shape $\rfloor L_1, \ldots, L_n \llcorner$ where $L_i = A_i$ or $L_i = \neg A_i$, $1 \leq i \leq n$.

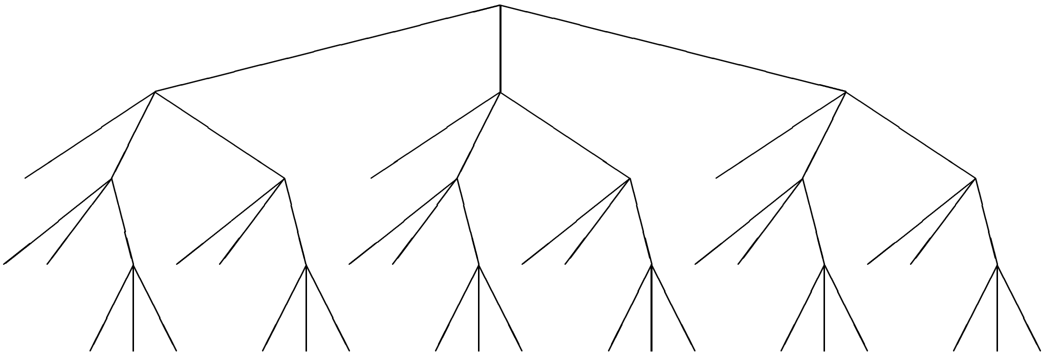


Figure 3.7: Tree structure of a minimal closed tableau for Example 3.3.1, $n = 3$.

Tableaux can be made stronger with respect to indeterministic power by adding the backward variant of the cut rule from sequent systems.

Procedure 3.3.3 (Tableau cut) Given a marked tableau T, choose an unmarked leaf node N, select any ground formula F, attach two successor nodes, and label them with F and $\neg F$, respectively. F is named the *cut formula* of the cut step.

The tableau cut rule is a *data-oriented* inference rule, in the following sense. The tableau cut rule can be simulated by alternatively admitting the addition of arbitrary tautologies of the shape $F \vee \neg F$ to the input formula and afterwards applying standard tableau expansion. This also proves the soundness of the cut rule.

The following proposition guarantees that we can always work with regular tableaux.

Proposition 3.3.6 *Any minimal closed tableau (with cut) for a set of ground formulae S is regular.*

Proof Any irregularity in a closed tableau can be removed by pruning the tableau, as follows. Let T be a closed tableau (with cut). Until all cases of irregularity are removed, repeat the following procedure.

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

Clearly, this operation does not affect the closedness of the tableau. □

Concerning indeterministic power, (regular) tableaux with cut and sequent systems are equivalent (for a proof see [Letz, 1993a]). But, unfortunately, the nice computational properties of analytic tableaux are lost when the cut rule is added in a naïve manner.

Proposition 3.3.7 *The (regular) tableau calculus with cut is infinitely branching, and hence, not polynomially transparent.*

A refinement of the general cut in tableaux is the *analytic* cut.

Procedure 3.3.4 (Tableau analytic cut) Given a marked tableau T for a set of ground formulae S, choose an unmarked leaf node N, select a ground formula F occurring as a subformula in some formula of S or on the path from the root up to N, attach two successor nodes, and label them with F and $\neg F$, respectively.

Proposition 3.3.8 *The (regular) tableau calculus with analytic cut is finitely branching and polynomially transparent.*

A definitely weaker variant of the general cut in tableaux is the atomic cut, with and without the analyticity condition.

Procedure 3.3.5 (Tableau (analytic) atomic cut) Given a marked tableau T for a set of ground formulae S, choose an unmarked leaf node N, select a ground atom A (occurring in a formula of S, for the analytic case), attach two successor nodes, and label them with the literals A and $\neg A$, respectively.

Fortunately, the working with *non-analytic* atomic cut can be avoided, due to the following proposition.

Proposition 3.3.9 *Any minimal closed tableau with atomic cut for a set of ground formulae S is a tableau with analytic cut.*

Proof Any application of non-analytic atomic cut in a closed tableau can be removed by pruning the tableau, as follows. Let T be a closed tableau with cut for S. First, obtain a regular tableau T' with cut by deleting all cases of irregularity from T. Then, as long as applications of non-analytic cut occur, repeat the following procedure.

> Select a node N in T' with two ancestor nodes N_1 and N_2 labelled with an atom A and its negation $\neg A$ not occurring in the input set S or on the path from the root up to N. Remove the edges originating in N and replace them with the edges originating in N_1 (or with the edges originating in N_2).

Clearly, in each step the tableau size decreases. It remains to be shown that the closedness is not affected. Closedness can only be affected if the node N_1 (the case of N_2 is treated analogously) is used as an ancestor in a reduction step, which can only be from a leaf node N' labelled with $\neg A$. Since A does not occur in the formula, N' must result from a cut step performed at its predecessor, so that N' would have a brother node labelled with A. But this contradicts the regularity assumption for T'. □

3.3.4 The Clausal Tableau Calculus

Definition 3.3.7 If a literal L occurs as a disjunct in a clause formula c, then we say that c *contains* L.

Definition 3.3.8 (Proper, compact clause formula) A clause formula c is *proper* if $c \neq -$. A clause formula $c = \rfloor L_1, \ldots, L_n \lfloor$ is said to be *compact* if each literal occurs only once as a disjunct in c.

Note For classical ground logic, attention can be restricted to compact clause formulae. The first-order case, however, demands the handling of general (possibly non-compact) clause formulae. The simple reason is that compact first-order clause formulae may have non-compact substitution instances. In order to be able to generalize the concepts and mechanisms developed for the ground case to the first-order case in a straightforward manner, it is conceptually better to work with general (possibly non-compact) clause formulae even on the ground level.

For sets of proper ground clause formulae, which are all of the disjunctive type, a much simpler form of tableaux and tableau calculus can be employed.

Definition 3.3.9 (Clausal tableau) A *clausal tableau* T *for* a finite set S of proper ground clause formulae is a tableau for S in which each successor set of nodes $N_1, \ldots, N_n$ is labelled with literals $L_1, \ldots, L_n$ such that S contains a clause formula $\rfloor L_1, \ldots, L_n \lfloor$. For any successor set of nodes $N_1, \ldots, N_n$ in a clausal tableau, the generalized disjunction $\rfloor L_1, \ldots, L_n \lfloor$ of their labels is called a *tableau clause formula.* The tableau clause formula immediately below the root node of a clausal tableau is called the *top clause formula* of the tableau.

The tableau displayed above in Figure 3.6 is a clausal tableau. In the clausal tableau *calculus* the reduction rule remains the same, whereas the tableau expansion rule degenerates in the following way.

Procedure 3.3.6 (Clausal tableau expansion) Given a set S of proper ground clause formulae as input and a marked tableau for S, choose a leaf node N and select a clause formula $c = \rfloor L_1, \ldots, L_n \lfloor$ from S, attach n new successor nodes $N_1, \ldots, N_n$ to N, and label them with $L_1, \ldots, L_n$, respectively.

Clausal tableaux (with atomic cut) can polynomially simulate general tableaux (with atomic cut), provided appropriate (definitional) translations are permitted (see [Reckhow, 1976]). As a matter of fact, the converse holds too.[14] Clausal tableaux with atomic cut and semantic trees are even more closely related.

Proposition 3.3.10 *There is a polynomial p such that for any closed clausal tableaux T with atomic cut for a set S of proper ground clause formulae there exists a closed semantic tree T' for the set of clauses corresponding to the clause formulae in S with* $\text{size}(T') < p\,(\text{size}(T))$, *and conversely.*

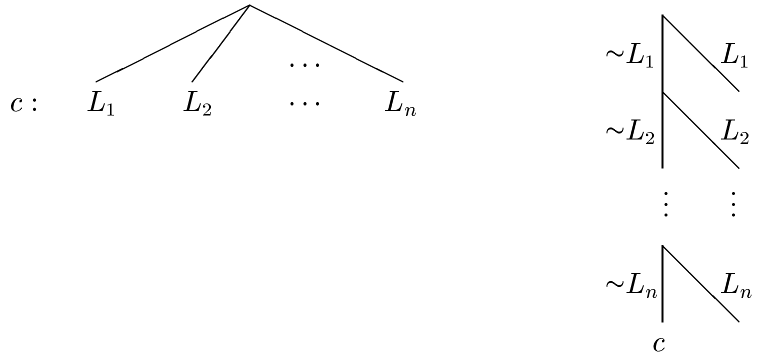


Figure 3.8: Simulation of tableau expansion with semantic trees.

Proof We prove that every tableau inference step can be polynomially simulated by semantic trees. The polynomial simulation of expansion and reduction steps is shown in the Figures 3.8 and 3.9, respectively. The simulation of the atomic cut step is trivial (due to Proposition 3.3.9 we can restrict ourselves to analytic cuts), instead of the nodes simply the edges must be labelled in the semantic tree. The converse simulation is as follows. Semantic tree expansion is simulated by tableau cut, and the labelling of a leaf node with a clause $\{L_1, \ldots, L_n\}$ contradicting the partial interpretation of a semantic tree branch is simulated by a tableau

[14] In order to obtain a fair evaluation of indeterministic power, the use of definitional translations must also be permitted for general tableaux, even if they accept non-normal form formulae. That is, if a system is more general than another wrt to its rules and its input language, then the general system must always polynomially simulate the special one (as opposed to the view in [Reckhow, 1976]).

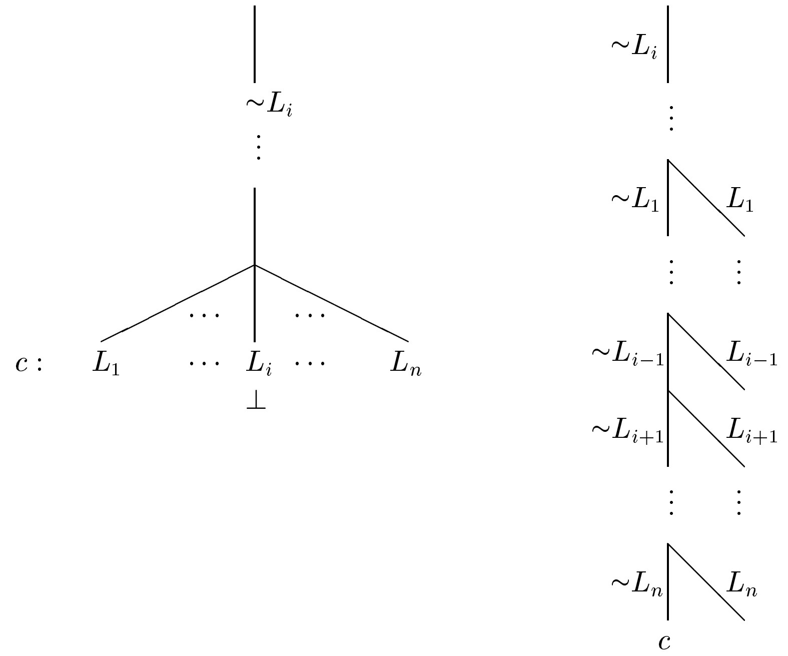


Figure 3.9: Simulation of tableau reduction with semantic trees.

expansion step using the clause formula $\rfloor L_1, \ldots, L_n \lfloor$, followed by n reduction steps. □

3.3.5 The Connection Method

Based on work by Prawitz [Prawitz, 1960, Prawitz, 1969], the *connection method* was introduced by Bibel in [Bibel, 1981, Bibel, 1987] and Andrews [Andrews, 1981]—we shall use Bibel's terminology as reference point. The connection method is not a specific calculus or inference system but a general deduction methodology which emphasizes the importance of *connections* for automated theorem proving. In the original presentation of the connection method, the logical validity of first-order formulae is proven directly, which is the dual of demonstrating the unsatisfiability of the negations of the formulae. In order to keep proximity to the refutational approach pursued in this work, we present the connection method as a framework for proving the unsatisfiability of formulae, which makes no difference concerning the employed notions and mechanisms. The connection method for ground formulae can handle arbitrary formulae constructed over the connectives $\wedge$, $\vee$, and $\neg$ with the restriction that $\neg$ is permitted to dominate atomic formulae only. We present the version for sets of proper clause formulae here.

Definition 3.3.10 (Literal occurrence in a set of clause formulae) For any clause formula $c = \forall x_1 \cdots \forall x_m \rfloor L_1, \ldots, L_n \lfloor$ in a set of clause formulae S, any triple $\langle L_i, i, c \rangle$, $1 \leq i \leq n$, is called a *literal occurrence* in S.

As opposed to the case of clauses, for clause formulae, triples are needed to individuate literal occurrences, since, in the case of non-compact clause formulae, one has to distinguish between different occurrences of one and the same literal as a disjunct in a clause formula.

Definition 3.3.11 (Path, connection, mate, complementarity) Given a finite set of proper clause formulae S of cardinality n. A *path in S* is a set of n literal occurrences in S, exactly one from each clause formula in S. Any subset of a path in S is called a *subpath in S*. A *connection in S* is a two-element subpath $\{\langle K, i, c_1\rangle, \langle L, j, c_2\rangle\}$ in S such that K and L are literals with the same predicate symbol, one literal is negated, and one is not. The literal occurrences in the connection are called *mates* of each other, and c_1 and c_2 are said to be *connected.* A path or subpath in S is called *complementary* if it contains a connection as subset in which the connected literals have the same atoms. If all paths in a set S of proper clause formulae are complementary, then S is named *complementary* too.

The connection method is based on two fundamental principles. The first is the idea that the unsatisfiability of a proper set of clause formulae can be proved by checking all paths in S for complementarity.

Proposition 3.3.11 *A set S of proper ground clause formulae is unsatisfiable if and only if S is complementary.*

Proof Let $S = \{c_1, \ldots, c_n\}$. Then, the equivalent conjunction $c_1 \wedge \cdots \wedge c_n$ can be equivalently transformed into disjunctive normal form by iteratively applying the $\wedge$-distributivity (Proposition 1.2.1 (m)). By Definition 1.7.11, a ground formula in disjunctive normal form in which $-$ does not occur is unsatisfiable if and only if each general disjunction contains a literal and its complement as a conjunct. The paths in S represent exactly the conjunctions of literals occurring in this disjunctive normal form formula. □

There are different methodologies of checking paths for complementarity. One of the most naïve ways is exemplified with the cut-free tableau calculus, which therefore is belonging to the class of path checking procedures.

The second principle of the connection method is to use *(sets of) connections* as control mechanism for path checking, which has no correspondence in the standard tableau approach.

Definition 3.3.12 (Mating, spanning property) Given a finite set of proper clause formulae S. Any set of connections in S is called a *mating for S*. A mating M for S is said to be *spanning* if each path in S contains a connection in M as a subpath; M is called *complementary* if each of its connections is complementary.

Proposition 3.3.12 *A set S of proper ground clause formulae is unsatisfiable if and only if there is a complementary spanning mating for S.*

Note On the ground level, if S has a complementary spanning mating, then the set of *all* complementary connections in S is spanning, too. Consequently, we can always work with the set of all complementary connections. On the first-order level, however, the additional condition of unifiability comes into play which makes it necessary to work with *proper* subsets of the set of all *unifiable* connections. Interestingly, a complementary spanning mating for a set of ground clause formulae S cannot be accepted as a *refutation* of S, since, apparently, the problem of verifying the spanning property of a mating is coNP-complete. Hence, with spanning matings nothing is gained with respect to proving unsatisfiability. In a deduction *enumeration* approach, however, matings can be used to reduce the number of deductions tremendously, as discussed in Subsection 4.4.3.

3.4 Connection Tableaux

One of the basic ideas of the connection method, to use connections as a control structure for path checking, can be formulated as a refinement of clausal tableaux.

Definition 3.4.1 (Connection tableau) If T is a clausal tableau in which each inner node N has a complementary leaf node N' among its successor nodes, then T is called *connected* or a *connection tableau.*

The tableau shown in Figure 3.6 is a connection tableau. The connection tableau *calculus* introduced now describes the standard *top-down* methodology of building up connection tableaux.

3.4.1 The Connection Tableau Calculus

In order to guarantee the connectedness condition, it is reasonable to reorganize the standard tableau inference rules and to define the connection tableau calculus as consisting of three inference rules, *tableau start*, *tableau extension*, and the reduction rule from the standard tablau calculus. Again, we work with marked tableaux.

Procedure 3.4.1 (Tableau start) Given a set of proper ground clause formulae S as input and a one-node tree with root N and label $\top$, a *start step* is simply a tableau expansion step.

The tableau extension step is a particular tableau expansion step immediately followed by a special tableau reduction step.

Procedure 3.4.2 (Tableau extension) Given a set of proper ground clause formulae S as input and a marked connection tableau T for S, choose a leaf node N with literal L which is not marked, select a clause formula $\lrcorner L_1, \ldots, L_n \llcorner$ in S containing $\sim L$ as a disjunct, and attach n successors to N labelled with the literals $L_1, \ldots, L_n$, respectively (this is an expansion step); then mark a successor N' of N which is labelled with the literal $\sim L$ with N (this is a reduction step).

The transition relational formulation of the connection tableau calculus and its regular version can be defined in analogy to tableaux. Apparently, we can formulate the following fundamental difference with the general clausal tableau calculus.

Proposition 3.4.1 *The (regular) connection tableau calculus is finitely branching, polynomially transparent, but not proof-confluent.*

Proof The finite branching rate and the polynomial transparency follow from the fact that the connection tableau calculus is basically a refinement of the tableau calculus (except that in extension steps two tableau inference steps are counted as one, which is not harmful computationally). To recognize the missing proof-confluence, assume an input set S contains a clause formula c containing a literal L with ${\sim}L$ not contained in a formula of S. Then, there exists no closed tableau with c as top clause formula. □

Hence, if we are going to make use of the connection tableau calculus as a proof or even a decision procedure for sets of proper ground formulae, then we are forced to *enumerate* connection tableaux. This is the main weakness of connection tableaux for the ground case, for which a different functionality is demanded than for the first-order case.

Also, the connectedness condition results in a weakening of the indeterministic power of clausal tableaux.

Proposition 3.4.2 *The connection tableau calculus cannot polynomially simulate the clausal tableaux calculus.*

Proof A simple modification of Example 3.3.1 will do, namely, the class presented in Example 3.4.1. The additional tautologies[15] can be used to polynomially simulate the atomic cut rule in the clausal tableau calculus, hence permitting short proofs for this example. But in connection tableaux, except for the start step, the tautologies do not help, since any extension step at a node N with literal L using the tautology $\rfloor L, {\sim}L \llcorner$ just lengthens the respective path by a node labelled with the same literal L. Therefore, the size of any closed connection tableau for an input set S_n is greater than $2 \times (n-1)!$ while the size of S_n is $n \times (2^n + 2)$. □

Example 3.4.1 For any set $\{A_1, \ldots, A_n\}$ of distinct propositional atoms, let S_n denote the set of propositional clause formulae given in Example 3.3.1, augmented with n tautologies of the shape $\rfloor A_i, \neg A_i \llcorner$, $1 \leq i \leq n$.

Moreover, the regularity condition turns out to be harmful for the indeterministic power of connection tableaux.

[15] That these formula are tautologies is not essential for the argument. We could equally well replace every tautology $\rfloor A_i, \neg A_i \llcorner$ with two clause formulae $\rfloor A_i, \neg B_i \llcorner$ and $\rfloor B_i, \neg A_i \llcorner$ with the B_i being n new distinct propositional atoms.

Proposition 3.4.3 *The regular connection tableau calculus cannot polynomially simulate the connection tableau calculus.*

Proof For this result we use another modification of Example 3.3.1, which is given in in Example 3.4.2. The elements of this class have polynomial closed connection tableaux, since the additional clause formulae permit the polynomial simulation of the atomic cut rule, as illustrated in Figure 3.10, for the case of $n = 3$; to gain readability, A_i is abbreviated with i and $\neg A_i$ with $\overline{\imath}$ in the figure. These connection proofs are highly irregular. In order to obtain short proofs, it is necessary to attach the *intermediating* two-literal clause formulae of the shape $\rfloor A_i, A_0 \lfloor$ and $\rfloor \neg A_i, A_0 \lfloor$ again and again. In regular proofs, however, on each branch intermediating clause formulae can be used at most once. Therefore, the size of any closed regular connection tableau for an S_n is greater than $4 \times (n-2)!$ while the size of S_n is $n \times (2^n + 7)$. □

Example 3.4.2 For any set $\{A_1, \ldots, A_n\}$ of distinct propositional atoms, let S_n denote the set of propositional clause formulae given in Example 3.3.1, augmented with

1. n tautologies of the shape $\rfloor A_i, \neg A_i, \neg A_0 \lfloor$, $1 \leq i \leq n$,
2. n clause formulae of the structure $\rfloor A_i, A_0 \lfloor$, $1 \leq i \leq n$, and
3. n clause formulae of the structure $\rfloor \neg A_i, A_0 \lfloor$, $1 \leq i \leq n$,

where A_0 is a new propositional atom.

3.4.2 Tableau Node Selection Functions

There is a source of indeterminism in the tableau and the connection tableau calculi which can be removed without any harm concerning indeterministic power. This indeterminism concerns the selection of the next unmarked node at which an expansion, extension, reduction, or cut step is to be performed.

Definition 3.4.2 (Selection function) A *(node) selection function* ϕ is a mapping assigning to every marked tableau T which is not marked as closed an unmarked leaf node N in T. The node N is called *the node selected by* ϕ.

Proposition 3.4.4 (Strong node selection independency) *Any closed (connection) tableau for a set of ground clause formulae can be constructed with any possible selection function.*

Note This property is extremely important for tableau proof procedures. Thus, we can always work with a fixed selection function, or switch from one selection function to another if necessary, without losing indeterministic power. This latter property does not hold for resolution procedures like, e.g., linear resolution,

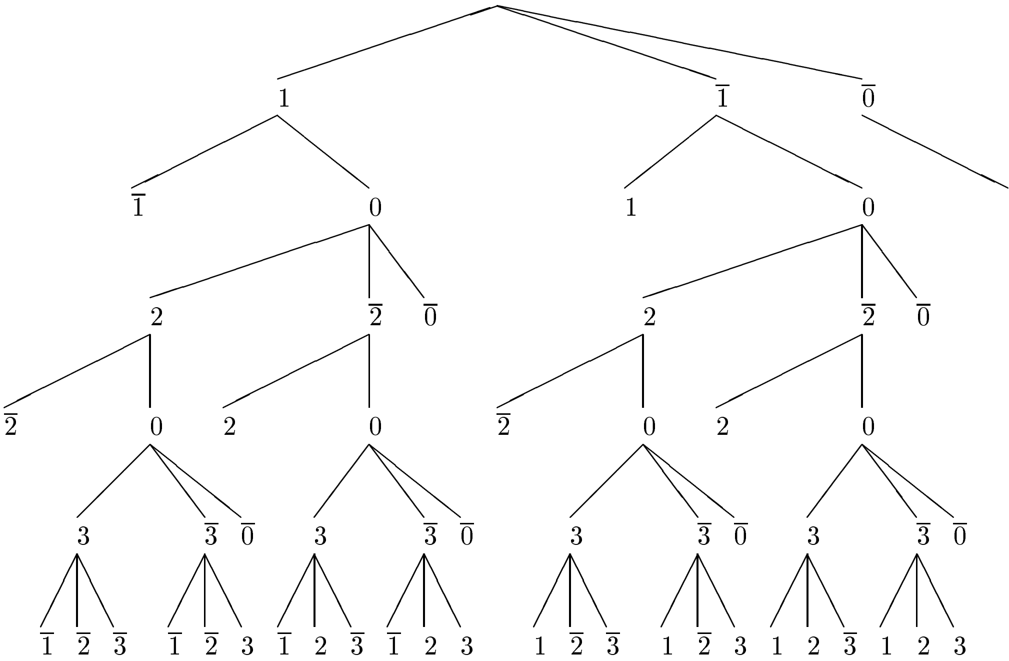

1 $\overline{1}$ $\overline{0}$
$\overline{1}$ 0 1 0 0
2 $\overline{2}$ $\overline{0}$ 2 $\overline{2}$ $\overline{0}$
$\overline{2}$ 0 2 0 $\overline{2}$ 0 2 0
3 $\overline{3}$ $\overline{0}$ 3 $\overline{3}$ $\overline{0}$ 3 $\overline{3}$ $\overline{0}$ 3 $\overline{3}$ $\overline{0}$
$\overline{1}$ $\overline{2}$ $\overline{3}$ $\overline{1}$ $\overline{2}$ 3 $\overline{1}$ 2 $\overline{3}$ $\overline{1}$ 2 3 1 $\overline{2}$ $\overline{3}$ 1 $\overline{2}$ 3 1 2 $\overline{3}$ 1 2 3

Figure 3.10: Polynomial closed connection tableau for Example 3.4.2, $n = 3$.

which is only *weakly* independent of the node selection function, in the sense that all selection function preserve completeness, but they may produce different (shortest) proofs. In other terms, different resolution selection functions induce different resolution dags.

Definition 3.4.3 (Standard selection functions) A *depth-first (depth-last) selection function* selects from any tableau T containing unmarked leaf nodes one with a maximal (minimal) depth. The *depth-first (depth-last) left-most (right-most) selection function* selects that node which is the left-most (right-most) node among the unmarked nodes with maximal (minimal) depth.

3.4.3 From Tableaux to Subgoal Formulae

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 systems is not immediately evident. In order to clarify the interdependencies it is helpful to reformulate the process of tableau construction in terms of formula generation procedures. There is a natural mapping from tableaux to formulae represented by the tableaux, particularly, if only the *open parts* of tableaux are considered, which we call *subgoal trees.*

Definition 3.4.4 (Subgoal tree) The *subgoal tree of* a marked tableau T is the literal tree obtained from T by deleting out all nodes, together with their ingoing edges, which are on branches with marked leaves only.

For proving the unsatisfiability of a set of formulae using the tableau framework, it is not necessary to *explicitly construct* a closed tableau, it is sufficient to *know* that the deduction *corresponds* to a closed tableau. The subgoal tree of a tableau contains only the unmarked leaves of a tableau and those nodes which dominate unmarked leaves. For the continuation of the refutation process, all other parts of the tableau may be disregarded without any harm.[16] Most implementations of tableau calculi work by manipulating subgoal trees instead of tableaux. From subgoal trees it is but a small step to the corresponding logical formulae.

The logical interpretation of tableaux mentioned above—as the disjunction of the conjunctions of the literals on its branches—leads to the definition of so-called *consolvents* (see [Eder, 1991]). Here we do not use the consolvent interpretation of a tableau, because this reading destroys the internal structure of a tableau, which is very uncomfortable for defining refinements and extensions of the calculus. Instead, we use a mapping which mirrors the tableau structure. Given a subgoal tree of a tableau, the corresponding *subgoal formula* is defined inductively as follows.

Definition 3.4.5 (Subgoal formula) (inductive)

1. The *subgoal formula* of the empty subgoal tree is $-$.
2. The *subgoal formula* of a one-node tree with label F is simply F.
3. The *subgoal formula* of a complex tree with root N and label F, and immediate subtrees $t_1, \ldots, t_n$, in this order, is the formula $F \wedge (F_1 \vee \cdots \vee F_n)$ where F_i is the subgoal formula of t_i, for every $1 \leq i \leq n$.

Notation 3.4.1 According to our definition of tableaux and subgoal formulae, any complex subgoal formula has the shape $\top \wedge F$. In order to eliminate this redundancy, we shall abbreviate any complex subgoal formula $\top \wedge F$ by writing just the logically equivalent formula F.

In Figure 3.12 three sequences of subgoal formulae are depicted which are corresponding to three different constructions of the tableau in Figure 3.6 using the connection tableau calculus. In order to facilitate the identification of the inferences on subgoal formulae, the original tableau is redisplayed in Figure 3.11 with numbers as names of the relevant nodes, which appear as upper indices at the literals. The subgoal formula sequences differ with respect to the chosen selection functions, the node selected for the next inference step is marked by framing the indexed literals.

The structure of the subgoal tree encoded by a subgoal formula can be read off easily. For every subformula of the shape $L \wedge F$ the node corresponding to the occurrence of L dominates all nodes represented by the literal occurrences

[16] But note that information about the closed part of a tableau and its structure may be necessary for improving search pruning. This will become relevant for the first-order case.

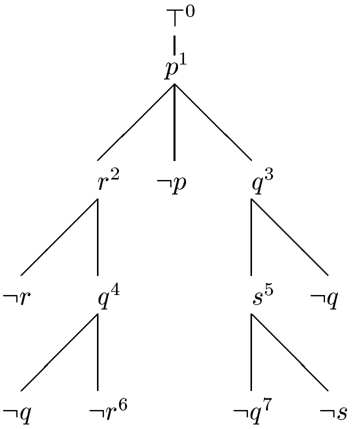


Figure 3.11: Connection tableau of Figure 3.6 with node indices.

depth-first left-most	depth-first right-most	depth-last left-most
$\boxed{\top^0}$	$\boxed{\top^0}$	$\boxed{\top^0}$
$\boxed{p^1}$	$\boxed{p^1}$	$\boxed{p^1}$
$p^1 \wedge (\boxed{r^2} \vee q^3)$	$p^1 \wedge (r^2 \vee \boxed{q^3})$	$p^1 \wedge (\boxed{r^2} \vee q^3)$
$p^1 \wedge ((r^2 \wedge \boxed{q^4}) \vee q^3)$	$p^1 \wedge (r^2 \vee (q^3 \wedge \boxed{s^5}))$	$p^1 \wedge ((r^2 \wedge q^4) \vee \boxed{q^3})$
$p^1 \wedge ((r^2 \wedge q^4 \wedge \boxed{\neg r^6}) \vee q^3)$	$p^1 \wedge (r^2 \vee (q^3 \wedge s^5 \wedge \boxed{\neg q^7}))$	$p^1 \wedge ((r^2 \wedge \boxed{q^4}) \vee (q^3 \wedge s^5))$
$p^1 \wedge \boxed{q^3}$	$p^1 \wedge \boxed{r^2}$	$p^1 \wedge ((r^2 \wedge q^4 \wedge \neg r^6) \vee (q^3 \wedge \boxed{s^5}))$
$p^1 \wedge q^3 \wedge \boxed{s^5}$	$p^1 \wedge r^2 \wedge \boxed{q^4}$	$p^1 \wedge ((r^2 \wedge q^4 \wedge \neg r^6) \vee (q^3 \wedge s^5 \wedge \boxed{\neg q^7}))$
$p^1 \wedge q^3 \wedge s^5 \wedge \boxed{\neg q^7}$	$p^1 \wedge r^2 \wedge q^4 \wedge \boxed{\neg r^6}$	$p^1 \wedge q^3 \wedge s^5 \wedge \boxed{\neg q^7}$
$\bot$	$\bot$	$\bot$

Figure 3.12: Subgoal formula proofs corresponding to the tableau in Figure 3.6.

within the occurrence of F; this involves that the open leaf nodes in the subgoal tree are encoded by literal occurrences not immediately followed by a conjunction symbol.

3.4.4 Connection Matrices

The connection calculus presented in [Bibel, 1987] Chapter III.6 can be viewed as a version of the connection tableau calculus restricted to depth-first selection functions. Here we shall consider a refinement of this connection calculus, without *factorization*, which is studied below. The favourite notation for displaying connection proofs is by writing them as *matrices*, with the columns consisting of the literals in the clause formulae. The information about the paths which have been examined and those which remain to be checked in a certain state, is expressed with some additional data structures. In the original presentation, some worked-off parts remain noted in the deduction. We apply the same technique used for tableaux—the working with subgoal trees—to the connection calculus, by working with *subgoal matrices.* The resulting format is particularly appealing for the presentation of deductions obeying any form of depth-first selection, as shown in Figure 3.13. In the matrix proof we have indicated the *next* inference with arrows at the selected subgoal, –☼ for extension, ☺– · · · — for reduction, and =○= for the path under consideration. Additionally, in any extension step, the clause and the entry literal is given, and in any reduction step the respective predecessor node. The relation of subgoal matrix proofs with the more general subgoal formula and subgoal tree notation is evident. A subgoal matrix is basically a subgoal tree put on its left side. Notice that the subgoal proof on the right-hand side of Figure 3.12 cannot be performed by the mentioned connection calculus and not be represented in the subgoal matrix notation.

$\top$						
p^1	⊥☼	$c_2.2$				
p^1	=○=	q^3				
		r^2	⊥☼	$c_6.1$		
p^1	=○=	q^3				
		r^2	=○=	q^4	⊥☼	$c_5.1$
p^1		q^3				
		r^2	☺⊥	q^4	=○=	$\neg r^6$
p^1	=○=	q^3	⊥☼	$c_3.2$		
p^1	=○=	q^3	=○=	s^5	⊥☼	$c_4.2$
p^1	=○=	q^3	☺⊥	s^5	=○=	$\neg q^7$
$\bot$						

Figure 3.13: Subgoal matrix notation of the left-hand proof in Figure 3.12.

3.4.5 Model Elimination

The *model elimination calculus* was introduced in [Loveland, 1968] and improved in [Loveland, 1978]. Model elimination can be viewed as a refinement of the connection tableau calculus. This approach has various advantages concerning generality, elegance, and the possibility for defining extensions and refinements.[17] Here, we treat a subsystem of the original model elimination calculus 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 subgoal formula deductions of connection tableaux. The weak

[17] The soundness and completeness results for model elimination, for example, are immediate consequences of the soundness and completeness proofs of (regular) connection tableaux, which are very short and simple. Compare these proofs with the extremely involved and lengthy proofs in [Loveland, 1978].

model elimination calculus can be viewed as that refinement of the connection tableau calculus in which the selection of open nodes is performed in a *depth-first right-most* manner. The strong node selection independence guarantees that weak model elimination is a complete refinement of the connection tableau calculus.

Due to the depth-first right-most restriction of the node selection function, a one-dimensional "chain" representation of subgoal formulae is possible (as used in [Loveland, 1968, Loveland, 1978]), in which no logical operators are necessary. The transformation from subgoal formulae with depth-first right-most selection function to model elimination chains works as follows.

Transformation from subgoal formulae to model elimination chains
To any subgoal formula generated with a depth-first right-most node selection function, apply the following operation. As long as logical operators are contained in the string, replace every conjunction $L \wedge F$ with $[L]F$ and every disjunction $L_1 \vee \cdots \vee L_n$ with $L_1 \cdots L_n$.

In a model elimination chain, the occurrences of bracketed literals denote the non-leaf nodes and the occurrences of unbracketed literals the leaf nodes of the subgoal tree corresponding to the input subgoal formula. For every node N corresponding to an occurrence of an unbracketed literal L, the bracketed literal occurrences to the left of L encode the nodes dominating N. From the subgoal formula proofs in Figure 3.12 only the middle one can be represented in model elimination. The model elimination proof is depicted in Figure 3.14. Additionally, we have given a precise specification of the applied inference rules, using the following abbreviations:

S: $0, c_i$ denotes a start step into top clause formula c_i,
E: j, c_i denotes an extension step at node j into the i-th disjunct of c_i,
R: j, k denotes a reduction step at node j with a dominating node k.

p^1	S:	$0, c_1$
$[p^1]\, r^2\, q^3$	E:	$1, c_2.2$
$[p^1]\, r^2\, [q^3]\, s^5$	E:	$3, c_3.2$
$[p^1]\, r^2\, [q^3]\, [s^5]\, \neg q^7$	E:	$5, c_4.2$
$[p^1]\, r^2$	R:	$7, 3$
$[p^1]\, [r^2]\, q^4$	E:	$2, c_6.1$
$[p^1]\, [r^2]\, [q^4] \neg r^6$	E:	$4, c_5.1$
$-$	R:	$6, 2$

Figure 3.14: Model elimination notation of the middle proof in Figure 3.12.

Note It is important to emphasize that the node selection function determines the branching rate of the search tree of the calculus, i.e., the number of possible tableaux that can be generated in one inference step from a given tableau, which is essential for the first-order level. Consequently, all forms of restrictions on the node selection from the side of the *calculus* may heavily reduce the potential of search pruning in this calculus. This subject will be discussed in the next chapter.

3.4.6 Further Structural Restrictions on Tableaux

Connectedness and regularity are restrictions of the tableaux *structure*. Such restrictions are completely independent of the structure of the underlying set S of input formulae or of the structures of *other* tableaux for S. In particular, any structural restriction on tableaux of some type Φ meets the following monotonicity condition. If T is a tableau of type Φ for an input S, then T is a tableau of type Φ for any superset of S. Below we shall discuss important other restrictions which do not satisfy this monotonicity condition.

In [Plaisted, 1990] a refinement of connection tableaux is discussed in which the *reduction* rule may be omitted either for all subgoals with atoms or for all subgoals with negated atoms. This refinement can be formulated as a structural restriction on marked tableaux. We demonstrate this for the case of forbidding reduction steps at nodes labelled with atoms. The corresponding restriction on the structure of marked tableaux is as follows. If a node N in such a tableau is labelled with an atom, then no successor of N labelled with an atom must be marked, and if a node N is labelled with a negated atom, then at most one successor of N labelled with an atom is permitted to be marked. Interestingly, this partial restriction of reduction steps cannot be formulated as a restriction of pure (i.e., unmarked) tableaux.

Moreover, this partial restriction of reduction steps is incompatible with the regularity condition.

Proposition 3.4.5 *There are unsatisfiable sets of ground clause formulae such that every proof in the regular connection tableau calculus has to perform reduction steps both at nodes labelled with atoms and at nodes labelled with negated atoms.*

Proof We use the set of clause formulae given in Example 3.4.3. As illustration of the proof consider Figure 3.15. If the first clause formula is chosen as top clause formula, then, in any case, reduction steps have to be applied to nodes labelled with the literals $\neg q$ and r in the four possible subtrees T dominated by the p-node on the left-hand side; and similarly to nodes labelled with the literals $\neg q'$ and r' in the four possible subtrees T' on the right-hand side. Using one of the other clause formulae as top clause formula does not help, since then the clause $\neg p \vee \neg p'$ must be entered by an extension step, producing either an open leaf literal $\neg p$ or $\neg p'$. In either case a tree from one of two classes T or T' need to be attached then. □

Example 3.4.3 Consider a set of clause formulae of the following structures

$$\begin{array}{ll}
\rfloor \neg p, \neg p' \lfloor, & \\
\rfloor q, p, \neg r \lfloor, & \rfloor q', p', \neg r' \lfloor, \\
\rfloor q, s \lfloor, & \rfloor \neg q', s' \lfloor, \\
\rfloor s, \neg, q \lfloor, & \rfloor \neg s', \neg q' \lfloor, \\
\rfloor r, t \lfloor, & \rfloor \neg t, r \lfloor, \\
\rfloor r', t' \lfloor, & \rfloor \neg t', r' \lfloor.
\end{array}$$

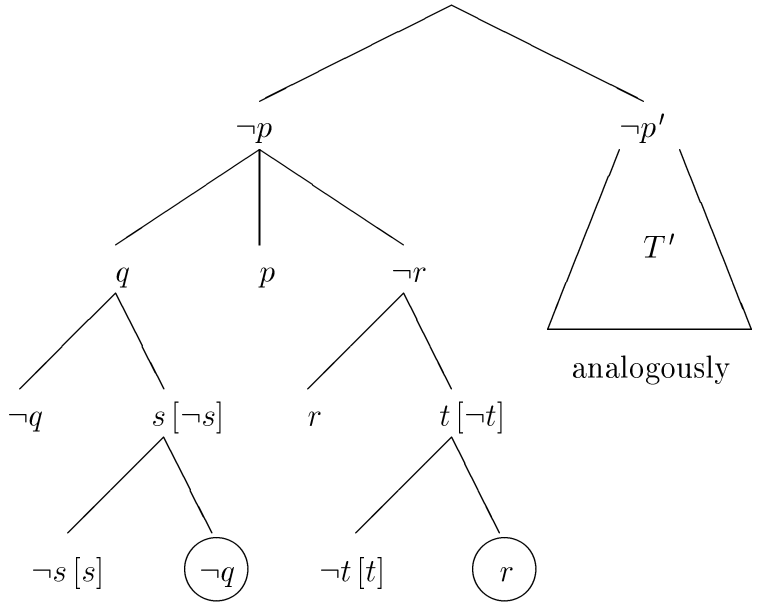


Figure 3.15: Necessity of full reduction for regular connection tableaux.

In order to preserve completeness for the restricted use of reduction rules, the regularity condition has to be weakened (see [Plaisted, 1990]). We do not pursue further this asymmetric approach.

Another refinement of connection tableaux, which has not been recognized so far and which is compatible with the regularity restriction, results from a sharpening of the connectedness condition.

Definition 3.4.6 (Strong connection) A connection $\{\langle L, i, c_1\rangle, \langle K, j, c_2\rangle\}$ is called *strong* if $L = \sim K$ and K is the only literal contained in c_2 with a complement in c_1. The literal occurrences in a strong connection are named *strong mates* of each other. Two clause formulae are *strongly connected* if they are connected and every connection between them is strong.

Example 3.4.4 Two clause formulae of the form $\rfloor P(a), \neg Q(a), P(b), P(a) \lfloor$ and $\rfloor \neg P(a), \neg Q(a), \neg Q(b), \neg P(a), \neg Q(b), \neg P(a) \lfloor$ are strongly connected.

In terms of clauses, strong connectedness can be expressed as follows. If c_1' and c_2' are the clauses containing the literals contained in two ground clause formulae c_1 and c_2, respectively, then the strong connectedness of c_1 and c_2 entails that there is exactly one ground resolvent c of c_1' and c_2', and c is not tautological.

Definition 3.4.7 (Strong connection tableau) A clausal tableau T is called *strongly connected* or a *strong connection tableau* if T is connected and any two tableau clause formulae c_1 and c_2 with c_1 lying immediately above c_2 are strongly connected.

Like connectedness and regularity, strong connectedness is a pure tableau structure condition, in the sense that marked tableaux are not needed to verify the condition.

3.4.7 The Completeness of Connection Tableaux

Now, we wish to furnish the completeness proof of regular strong connection tableaux, which yields as corollaries the completeness of all more general tableau variants discussed above. In fact, something slightly stronger can be proven, using the following notions.

Definition 3.4.8 (Essentiality, relevance, minimal unsatisfiability) An element F of a set of formulae S is called *essential in* S if S is unsatisfiable and $S \setminus \{F\}$ is satisfiable. An element F of a set of formulae S is named *relevant in* S if there exists an unsatisfiable subset $S' \subseteq S$ such that F is essential in S'. An unsatisfiable set of formulae S is said to be *minimally unsatisfiable* if each formula in S is essential in S.

Proposition 3.4.6 (Completeness of regular strong connection tableaux) *For any finite unsatisfiable set S of proper ground clause formulae and any clause formula c which is relevant in S, there exists a closed regular strong connection tableau for S with top clause formula c.*

For the completeness proof we need an additional notion and a basic lemma.

Definition 3.4.9 (Strengthening) The *strengthening* of a set of clause formulae S *by* a set of literals $P = \{L_1, \ldots, L_n\}$, written $P \mathbin{☼} S$, is the set of clause formulae obtained by first removing all clause formulae from S containing literals from P and afterwards adding the n clause formulae $\rfloor L_1 \lfloor, \ldots, \rfloor L_n \lfloor$.

Lemma 3.4.7 (Strong mate lemma) *Let S be an unsatisfiable set of ground clause formulae. For any literal L contained in any relevant clause formula c in S there exists a clause formula c' in S such that*

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

(ii) for every literal $L' \neq L$ in c': its complement $\sim L'$ is not contained in c, and

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

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 interpretation $\mathcal{I}$ for S_0 with $\mathfrak{I}(S_0 \setminus \{c\}) = \top$ and $\mathfrak{I}(c) = \bot$, for the formula assignment $\mathfrak{I}$ of $\mathcal{I}$; $\mathfrak{I}$ assigns $\bot$ to every literal in c. Define $\mathcal{I}' := (\mathcal{I} \setminus \{{\sim}L\}) \cup \{L\}$. Its assignment $\mathfrak{I}'$ maps c to $\top$. The unsatisfiability of S_0 guarantees the existence of a clause formula c' in F_0 with $\mathfrak{I}'(c') = \bot$. We prove that c' meets the conditions (i) – (iii). First, the clause formula c' must contain the literal ${\sim}L$, since otherwise $\mathfrak{I}(c') = \bot$, which contradicts the selection of $\mathcal{I}$, hence (i). Secondly, for any other literal $L' \neq L$ in c': $\mathfrak{I}(L') = \mathfrak{I}'(L') = \bot$. As a consequence, L' must not occur complemented in c, which proves (ii). Finally, the essentiality of c' in S_0 entails that there exists an interpretation $\mathcal{I}''$ with $\mathfrak{I}''(S_0 \setminus \{c'\}) = \top$ and $\mathfrak{I}''(c') = \bot$, for the assignment $\mathfrak{I}''$ of $\mathcal{I}''$. Since ${\sim}L$ is in c', $\mathfrak{I}''(L) = \top$. Therefore, c' is essential in $S_0 \cup \{L\}$, and also in its subset $\{L\} ☼ S_0$. From this and the fact that $\{L\} ☼ S_0$ is a subset of $\{L\} ☼ S$ follows that c' is relevant in $\{L\} ☼ S$. □

Proof of Theorem 3.4.6 Let S be a finite unsatisfiable set of proper ground clause formulae and c any relevant clause formula in S. A closed regular strong connection tableau T for S with top clause formula 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 formula. Then iterate the following non-deterministic procedure, as long as the intermediate tableau is not yet closed.

> Choose an arbitrary open leaf node N in the current tableau with literal L. Let c be the tableau clause formula 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 formula c' which is relevant in $P ☼ S$, 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 formula 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 is closed, or because no clause formula 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 formula c'. This will be demonstrated by induction on the node depth. The induction base, $n = 1$, is evident, by the Strong Mate Lemma (3.4.7). 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 formula c, and with a path set $P \cup \{L\}$ such that c is relevant in $P ☼ S$, the induction assumption. Let S_0 be any minimally unsatisfiable subset of $P ☼ S$ 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 formula of P' ☼ 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' ☼ S. □

3.5 Controlled Integration of the Cut Rule

The regular connection tableau calculus has proven successful in the practice of automated deduction [Stickel, 1988, Letz et al., 1992], although, concerning indeterministic power, the calculus is extremely weak. In this section we introduce different extensions of the calculus which attempt to remedy this weakness without introducing to much additional indeterminism. All discussed extensions can be viewed as *controlled* integrations of the cut rule.

It is apparent that the connectedness condition on tableaux blocks any reasonable application of the tableau cut rule, as it is defined in Procedure 3.3.3 on p. 116. This is because the connectedness enforces that any application of cut at a node N labelled with a literal L must label the two attached successor nodes with the literals L and $\sim L$, respectively, with the result that absolutely no advance is made towards the closing of the tableau. The effect of the cut rule on the shortening of tableau proofs can only be achieved for connection tableaux if the *tautology rule* is used, which is a generalized form of the cut rule.

Procedure 3.5.1 (Tautology rule for clausal tableaux) Given a marked tableau T, choose an unmarked leaf node N, select any tautological clause formula $\rfloor L_1, \ldots, L_n \lfloor$, attach n new successor nodes, and label them with $L_1, \ldots, L_n$, respectively.

Proposition 3.5.1 *(Regular) connection tableaux with the tautology rule can linearly simulate semantic trees and tableaux with atomic cut.*

Proof Any cut step with a cut formula A at a node N labelled with a literal L can be simulated by applying the tautology rule using the clause formula $\rfloor \sim L, A, \neg A \lfloor$. □

Corollary 3.5.2 *Connection tableaux cannot polynomially simulate (regular) connection tableaux with the tautology rule.*

Proof Immediate from Proposition 3.5.1 and Proposition 3.4.2 on p. 123. □

It is clear that the tautology rule is an inference rule of a theoretical value only, since the uncontrolled addition of the tautology rule to the connection tableau calculus completely destroys the good reductive properties of the calculus. The question is whether there exist other forms of additional inference rules which are better suited for automated deduction.

3.5.1 Factorization

The *factorization* rule was used in model elimination [Loveland, 1978] and in the connection calculus [Bibel, 1987], Chapter III.6. On the general level of the (connection) tableau calculus, which permits arbitrary node selection functions, the rule can be introduced as follows. Consider a closed tableau containing two nodes N_1 and N_2 with the same literal as label. 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 motivates the use of *factorization* as an additional inference rule. Factorization allows to label a node N_1 as solved in case there is another node N_2 labelled with the same literal, provided that the set of ancestors of N_2 is a subset of the set of ancestors of N_1. Possible 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. Applied to a set of clause formulae

$$\{\rfloor p, q\lfloor\,, \rfloor p, \neg q\lfloor\,, \rfloor \neg p, q\lfloor\,, \rfloor \neg p, \neg q\lfloor\,\}$$

which denotes an instance of Example 3.3.1 on p. 116, for $n = 2$, factorization yields a shorter proof, as shown in Figure 3.16. Factorization is indicated with an arc. Obviously, in order to preserve soundness this rule must be constrained to prohibit solution cycles. Thus, in Figure 3.16 factorization of the node N_4 on the right hand side with the node N_3 with the same literal on the left hand side is not allowed 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.

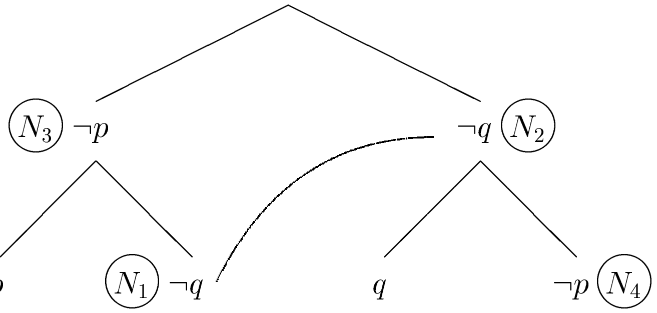


Figure 3.16: Factorization step in a connection tableau for Example 3.3.1, $n = 2$.

Definition 3.5.1 (Factorization dependency relation) A *factorization dependency relation on a tableau T* is a strict partial ordering $\prec$ on the tableau nodes.

Procedure 3.5.2 (Tableau factorization) Given a marked tableau T and a factorization dependency relation $\prec$ on its nodes. First, select a leaf node N_1 with literal L and another node N_2 labelled with the same literal such that

1. N_1 is dominated by a node N which has the node N_2 among its immediate successors, and
2. $M_1 \not\prec N_2$, where M_1 is the brother node of N_2 on the branch from the root to N_1, including the latter.[18]

Then, mark N_1 with N_2 and modify $\prec$ by first adding the pair of nodes $\langle N_2, M_1 \rangle$, and then forming the transitive closure of the relation. We say that the node 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 3.16, when the node 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 subgoal N_3 depends on the solution of the subgoal N_2. After that, factorization of the node N_4 with the node N_3 is not possible any more, and we have to proceed with a tableau extension step at the node N_4.

Note It is clear that the factorization dependency relation only relates brother nodes, i.e., nodes which have the same immediate predecessor.

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 3.5.3 (Strong node selection independency of factorization) *Any closed (connection) tableau with factorization for a set of ground clause formulae can be constructed with any possible selection function.*

The applications of factorization at a node N_1 with a node N_2 can be subdivided into two cases. Either, the node N_2 has been solved already, or the node N_2 or some of the nodes dominated by N_2 are not yet marked. 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 solution exists. Conversely, the first case will be called a *pessimistic* application of factorization.

Note If we are working with subgoal trees, i.e., completely remove solved parts of a tableau, 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 subgoals render cyclic factorization attempts impossible. It is for this reason, that the integration approaches of factorization into model elimination or the connection calculus have not mentioned the factorization dependency relation.

[18] Note that M_1 may be N_1 itself.

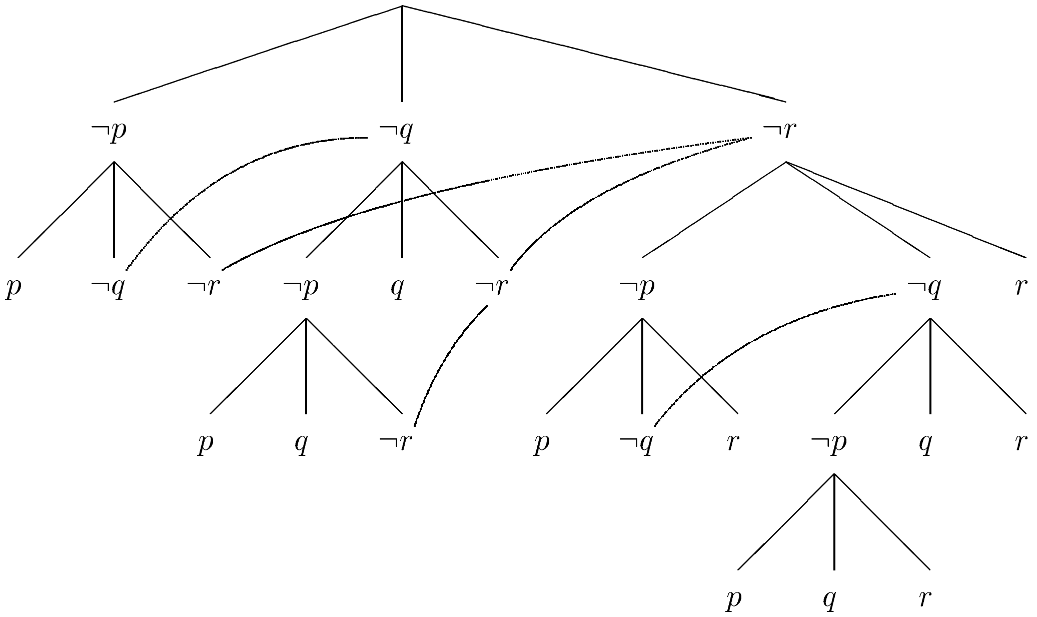


Figure 3.17: Linear closed connection tableau with factorization for Example 3.3.1, for the case of $n = 3$.

The addition of the factorization rule increases the indeterministic power of (connection) tableaux significantly. Thus, the critical class for tableaux given in Example 3.3.1, for which no polynomial proof exists (see Figure 3.7 on p. 116), has linear closed connection tableaux with factorization, as shown in Figure 3.17. In fact, the factorization rule is a certain restricted version of the cut rule. Connection tableaux with factorization, however, cannot polynomially simulate tableaux with atomic cut or regular connection tableaux with the tautology rule. Both results will be shown in the next subsection.

3.5.2 The Folding Up Rule

An inference rule, which is stronger than factorization concerning indeterministic power, is the so-called *folding up rule* (in German: “Hochklappen”). Folding up generalizes the *C-reduction* rule introduced for the model elimination format in [Shostak, 1976]. In contrast to factorization, for which pessimistic and optimistic application do not differ concerning deductive power, the increase in indeterministic power of the folding up rule results from its pessimistic nature. The theoretical basis of the rule is the possibility of extracting *lemmata* from solved parts of a tableau, which can be used on other parts of the tableau. Folding up represents a particularly efficient realization of this idea.

We explain the rule with an example. Given the situation displayed in Figure 3.18, where the bold 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 subgoals labelled with the literals r and q.

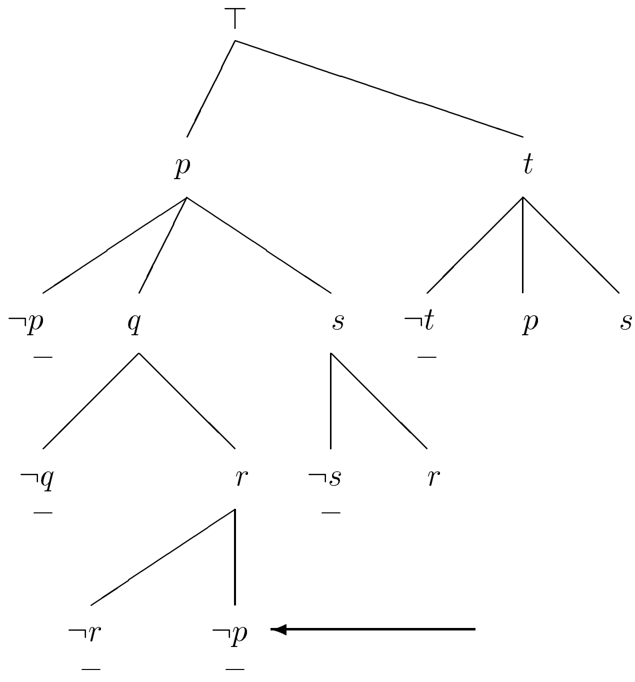


Figure 3.18: Connection tableau before folding up.

In the solutions of those subgoals the predecessor labelled with p has been used for a reduction step. Apparently, this amounts to the derivation of two lemmata $\rfloor \neg r, \neg p \lfloor$ and $\rfloor \neg q, \neg p \lfloor$ from the underlying formula. The new lemma $\rfloor \neg q, \neg p \lfloor$ 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* clause formulae, 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:

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

Also, the memorization is not done beside the tableau, but *within* the tableau itself, namely, by "folding up" a solved subgoal to the edge which dominates its solution context. More precisely, the folding up of a solved subgoal N to an edge E means labelling E with the negation of the literal at N. Thus, in the example above the edge E incident to the p-node on the left-hand side of the tableau is successively labelled with the literals $\neg r$ and $\neg q$, as displayed in Figure 3.19; sets 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 leaf

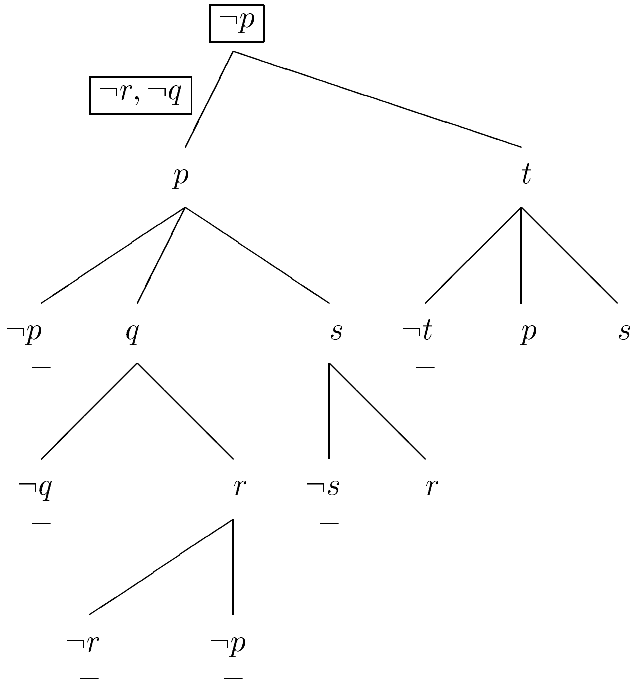


Figure 3.19: Connection tableau after three times folding up.

node 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 marking 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.

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

Definition 3.5.2 (Edge-labelled tableau) An *edge-labelled tableau (E-tableau)* is just a tableau as defined on p. 112, with the only modifications that also the edges are labelled by the labelling function λ, namely, with sets of literals, and that the root is not labelled with $\top$ but with sets of literals, too.

Procedure 3.5.3 (E-tableau folding up) Given a marked E-tableau T and a non-leaf node N with literal L which dominates marked leaves only. Let M be the set of nodes which are markings $\mu(N_i)$ of the leaf nodes N_i dominated by N. Obtain the set M' from M by removing N and all nodes dominated by N; note that all nodes in M' are on the path from the root to N, excluding the latter. Now, either M' contains solely the root node, in which case the label of the root is changed by adding the literal $\sim\!L$. Or M' contains an inner node N' which is dominated by all other nodes; then the label of the edge leading to N' is changed by adding $\sim\!L$.

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

Procedure 3.5.4 (E-tableau reduction) Given a marked E-tableau T, select a leaf node N with literal L, then, either select a dominating node N' with literal $\sim\!L$ and mark N with N', or select a dominating edge or the root E with $\sim\!L \in \lambda(E)$ and mark N with the node to which the edge is incident or with the root, respectively.

The *tableau* and the *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 set, and additionally labelling all newly generated edges with the empty set.

It is important to emphasize that the folding up rule is properly stronger concerning indeterministic power than the factorization rule.

Proposition 3.5.4 *Connection tableaux with factorization cannot polynomially simulate connection tableaux with folding up.*

Example 3.5.1 Consider a set S consisting of clause formulae of the structures

$$\begin{array}{ll}
\rfloor \neg p_0 \lfloor\,, & \\
\rfloor p_0, \neg p_1^1, \ldots, \neg p_m^1 \lfloor\,, & \\
\rfloor p_i^1, \neg p_1^2, \ldots, \neg p_m^2 \lfloor\,, & \text{for } 1 \leq i \leq m, \\
\qquad\quad \cdots & \\
\rfloor p_i^{n-1}, \neg p_1^n, \ldots, \neg p_m^n \lfloor\,, & \text{for } 1 \leq i \leq m, \\
\rfloor p_1^n \lfloor\,, & \\
\ldots & \\
\rfloor p_m^n \lfloor\,. &
\end{array}$$

Proof We use the class defined in Example 3.5.1. It can easily be recognized that any closed connection tableau for S with top clause formula $\rfloor \neg p_0 \lfloor$ has $\sum_{i=0}^{n} m^i$ leaf nodes. Also, factorization is not possible if we start with the top clause formula $\rfloor \neg p_0 \lfloor$, since no two subgoals N_1, N_2 with identical literals occur with N_2 being a brother node of an ancestor of N_1. Therefore, the example class is intractable for connection tableau with factorization, for this specific top clause

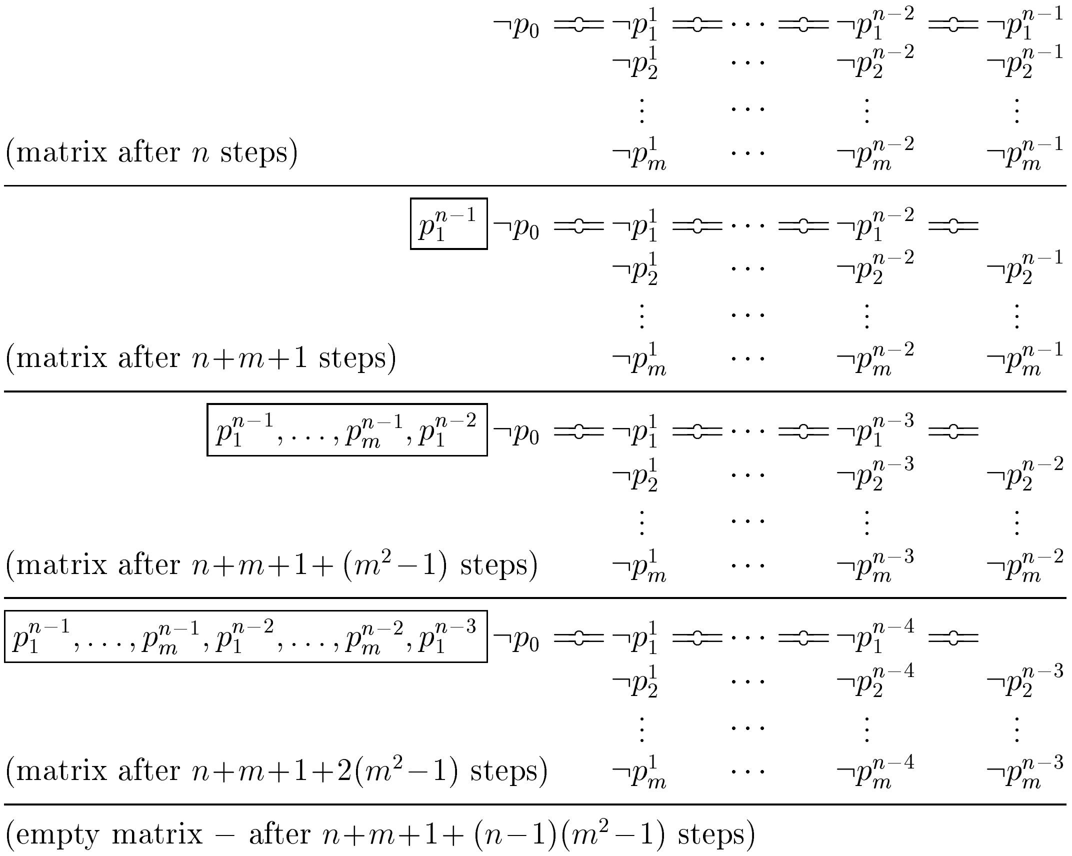

$\neg p_0$ $\neg p_1^1$ $\cdots$ $\neg p_1^{n-2}$ $\neg p_1^{n-1}$
$\neg p_2^1$ $\cdots$ $\neg p_2^{n-2}$ $\neg p_2^{n-1}$
$\neg p_m^1$ $\cdots$ $\neg p_m^{n-2}$ $\neg p_m^{n-1}$
(matrix after n steps)
p_1^{n-1} $\neg p_0$ $\neg p_1^1$ $\cdots$ $\neg p_1^{n-2}$
$\neg p_2^1$ $\cdots$ $\neg p_2^{n-2}$ $\neg p_2^{n-1}$
$\neg p_m^1$ $\cdots$ $\neg p_m^{n-2}$ $\neg p_m^{n-1}$
(matrix after $n+m+1$ steps)
$p_1^{n-1}, \ldots, p_m^{n-1}, p_1^{n-2}$ $\neg p_0$ $\neg p_1^1$ $\cdots$ $\neg p_1^{n-3}$
$\neg p_2^1$ $\cdots$ $\neg p_2^{n-3}$ $\neg p_2^{n-2}$
$\neg p_m^1$ $\cdots$ $\neg p_m^{n-3}$ $\neg p_m^{n-2}$
(matrix after $n+m+1+(m^2-1)$ steps)
$p_1^{n-1}, \ldots, p_m^{n-1}, p_1^{n-2}, \ldots, p_m^{n-2}, p_1^{n-3}$ $\neg p_0$ $\neg p_1^1$ $\cdots$ $\neg p_1^{n-4}$
$\neg p_2^1$ $\cdots$ $\neg p_2^{n-4}$ $\neg p_2^{n-3}$
$\neg p_m^1$ $\cdots$ $\neg p_m^{n-4}$ $\neg p_m^{n-3}$
(matrix after $n+m+1+2(m^2-1)$ steps)
(empty matrix – after $n+m+1+(n-1)(m^2-1)$ steps)

Figure 3.20: Linear connection proof with folding up for Example 3.5.1.

formula. However, there are linear connection proofs with factorization if one of the clause formulae

$$\rfloor p_i^{n-1}, \neg p_1^n, \ldots, \neg p_m^n \lfloor, \ 1 \le i \le m$$

is taken as top clause formula. In order to obtain an unsatisfiable class which is intractable for *any* selection of the top clause formula, we can apply the same trick used in the proof of Proposition 3.4.5 on p. 130. We modify the class given in Example 3.5.1 by adding a literal $\neg p_0'$ to the top clause formula $\rfloor \neg p_0 \lfloor$, and by adding renamed copies of the other clause formulae where the new literals are all consistently renamed and distinct from the old ones. For the resulting formula it does not matter with which clause formula we start, since now in *any* construction of a closed connection tableaux with factorization inevitably either $\neg p_0$ or $\neg p_0'$ must occur as a subgoal. And in the proof of this subgoal no factorization steps are possible, so that the resulting closed subtableau is isomorphic to the large one for the old formula class. Consequently, the new example class is intractable for connection tableau with factorization. Both the formula class from Example 3.5.1 and the modified class have linear proofs for connection tableau with folding up, as shown in Figure 3.20 for the initial class with $\rfloor \neg p_0 \lfloor$ as top clause formula; in the figure, we have used the presentation framework of connection matrices. The displayed proof needs $1+m+n+(n-1)(m^2-1)$ inference steps, while the number of literal occurrences in the respective clause set is $1+m+1+(n-1)(m+1)m+n = 2+m+n+(n-1)(m^2+m)$. □

Conversely, the polynomial simulation in the other direction is possible, for a certain class of selection functions.

Proposition 3.5.5 *For arbitrary depth-first selection functions, (connection) tableaux with folding up linearly simulate (connection) tableaux with factorization.*

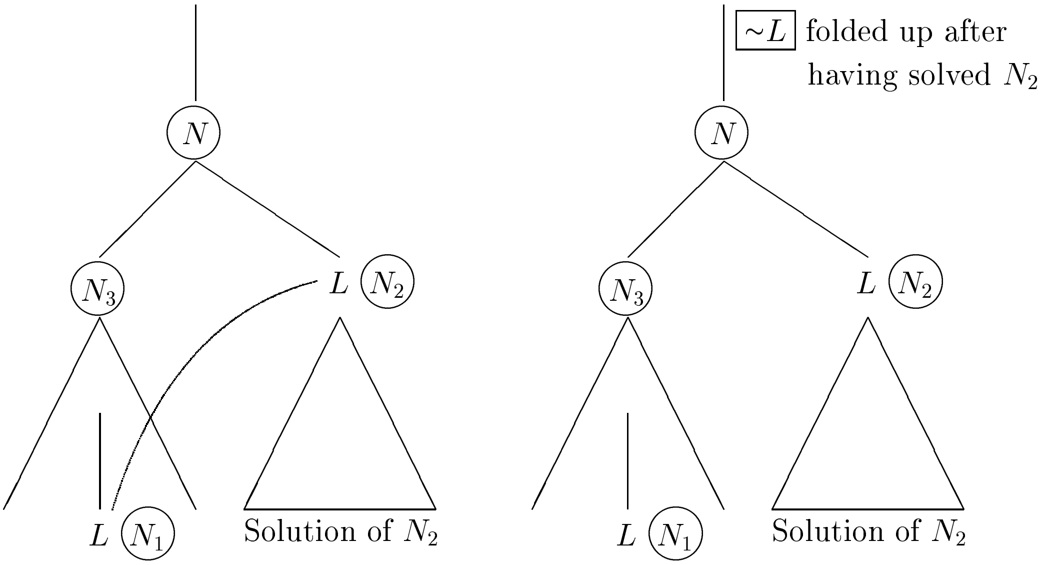


Figure 3.21: Simulation of factorization by folding up.

Proof Given any closed (connection) tableau T with factorization, let $\prec$ be its factorization dependency relation. By the strong node selection independency of factorization (Proposition 3.5.3 on p. 136), T can be constructed with any selection function. We consider a construction $S = (T_0, \ldots, T_m, T)$ of T with a depth-first selection function ϕ which respects the partial order of the factorization dependency relation $\prec$, i.e., for any two nodes N_1, N_2 in the tableau, $N_1 \prec N_2$ involves that N_1 is selected before N_2; such a selection function exists since $\prec$ solely relates brother nodes. The deduction process S can be directly simulated by the (connection) tableau calculus with folding up, as follows. Using the same selection function ϕ, any expansion (extension) and reduction step in S is simulated by an expansion (extension) and reduction step. But, whenever a subgoal has been completely solved in the simulation, it is folded up. Since in the original deduction process, due to the pessimistic application of factorization, the factorization of a node N_1 with a node N_2 (with literal L) involves that N_2 has been solved before, in the simulation the literal L must have been folded up before. Now, *any* solved subgoal can be folded up at least one step, namely, to the edge E above its predecessor. Since E dominates N_1, the original factorization step can be simulated by a reduction step. The simulation of factorization by folding up is graphically shown in Figure 3.21. □

Finally, we show that the folding up rule, although strictly more powerful than factorization, is still a hidden version of the cut rule.

Proposition 3.5.6 *Tableaux with atomic cut and regular connection tableaux with the tautology rule linearly simulate (connection) tableaux with folding up.*

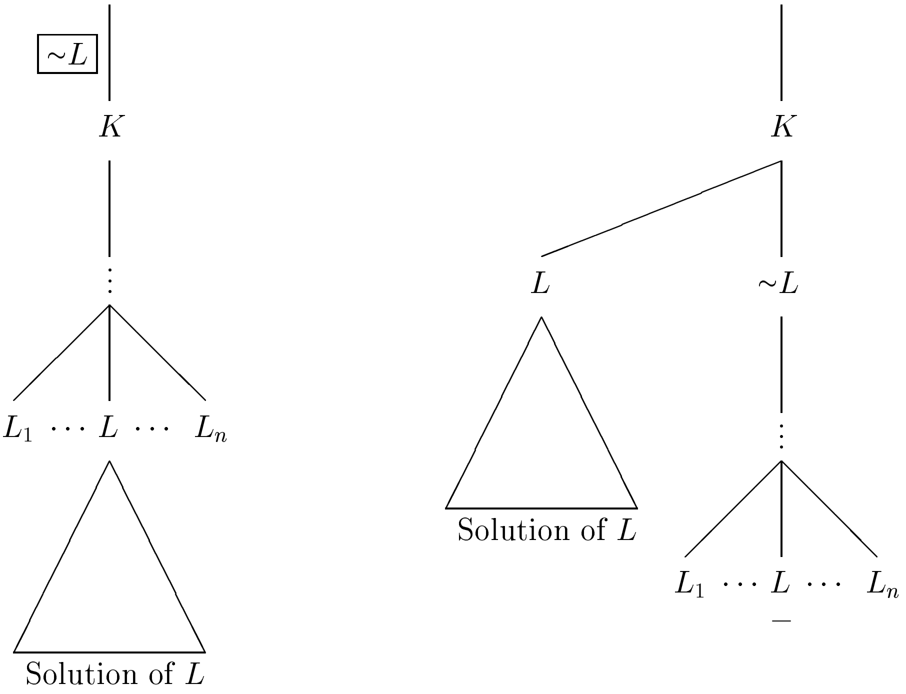


Figure 3.22: Simulation of folding up by cut.

Proof We perform the simulation proof for tableaux with cut. Given a tableau derivation with folding up, each folding up operation at a node N' adding the negation $\sim L$ of a solved subgoal L to the label of an edge incident to a node N, can be simulated as follows. Perform a cut step at the node N with the cut formula L, producing two new nodes N_1 and N_2 labelled with L and $\sim L$, respectively; shift the solution of L from N' 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'. Apparently, the unmarked branches of both tableaux can be injectively mapped to each other such that all pairs of corresponding branches contain the same sets of literals, respectively. The simulation is graphically shown in Figure 3.22. □

Corollary 3.5.7 *Tableaux with atomic cut and regular connection tableaux with the tautology rule can linearly simulate (connection) tableaux with factorization.*

3.5.3 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 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. The folding down rule is sound, since it can be simulated by the cut rule, as illustrated in Figure 3.23. The rule can 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 extreme new search space reduction can be achieved this way. It should be noted that it is very difficult to identify this refinement in the factorization framework.

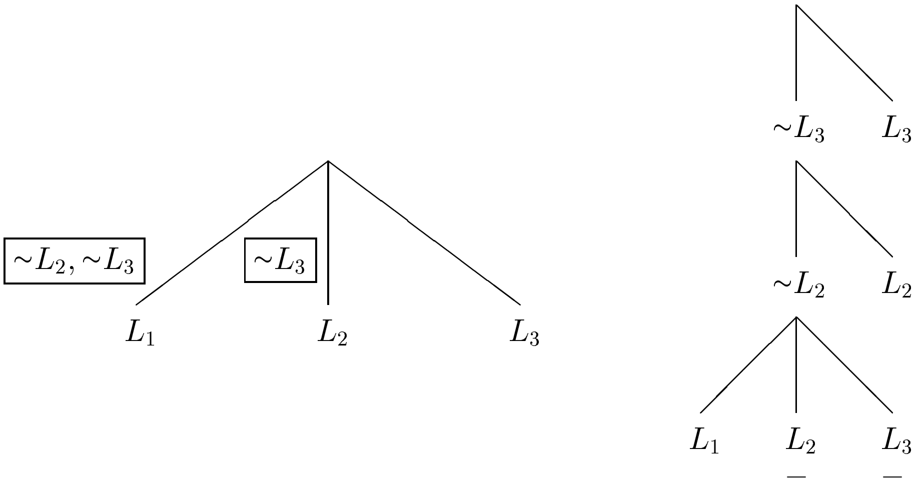


Figure 3.23: Simulation of folding down by cut.

3.5.4 Enforced Folding Up 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 subgoal *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 indeterministic 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 would be imported, resulting in an unnecessary increase of the search space. 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 must have the same literal as label. But apparently, 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 sharpened regularity condition. A straightforward remedy is to apply the sharpened condition to the *subgoal trees* of tableaux only.

Definition 3.5.3 (Strong regularity) An E-tableau T is called *strongly regular* if it is regular and on no branch of the subgoal tree of T a literal appears more than once, be it as a label of a node or within the label set of an edge or the root.

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 strongly regular connection tableaux (Theorem 3.4.6 on p. 132) cannot be applied. In fact, the new calculus is *not complete* for arbitrary selection functions.

Proposition 3.5.8 *There is an unsatisfiable set S of ground clause formulae 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 3.5.2 Consider a set of clause formulae of the structures

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

Proof Let S be the set of ground clause formulae given in Example 3.5.2, which is minimally unsatisfiable. The non-existence of a refutation with the top clause formula $\rfloor p, s, r \lfloor$ for a certain unfortunate selection function ϕ is illustrated in Figure 3.24. If ϕ selects the s-node, then two alternatives exist for extension, separated by a $\bigvee$. 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 formula, 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 formula. The same situation holds for any other top clause formula 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 same trick employed in the proofs of Proposition 3.4.5 on p. 130 and Proposition 3.5.4 on p. 140; by adding appropriate literals and clause formulae to the set one can easily obtain an input set in which the incompleteness result holds for *any* top clause formula. □

For depth-first selection functions, however, the new calculus is complete.

Theorem 3.5.9 (Completeness for depth-first selection functions of strongly regular connection tableaux with enforced folding up) *For any finite unsatisfiable set S of proper ground clause formulae, any depth-first tableau node selection function, and any clause formula c which is relevant in S, there exists a refutation of S with top clause formula c in the strongly regular connection tableau calculus with enforced folding up.*

Proof The completeness proof is very similar to the one for strongly regular connection tableaux (Theorem 3.4.6 on p. 132). We proof that for any subgoal N with a certain property there exists an inference step producing only subgoals with the same property. This inherited property is that the tableau clause formula determined by the position of N and its brother nodes is relevant in the strengthening[19] of S by the extended path set[20] P from the root to N, excluding the latter. Suppose that N with literal L is such a subgoal with tableau clause formula c which is relevant in P ☼ S. If N is selected first for solution, either a reduction step can be performed at N, or, by the Strong Mate Lemma 3.4.7, a

[19] As introduced in definition 3.4.9 on p. 132.

[20] The *extended* path set of a path contains all literals at the nodes of the path or in the union of the labels of the edges and the root.

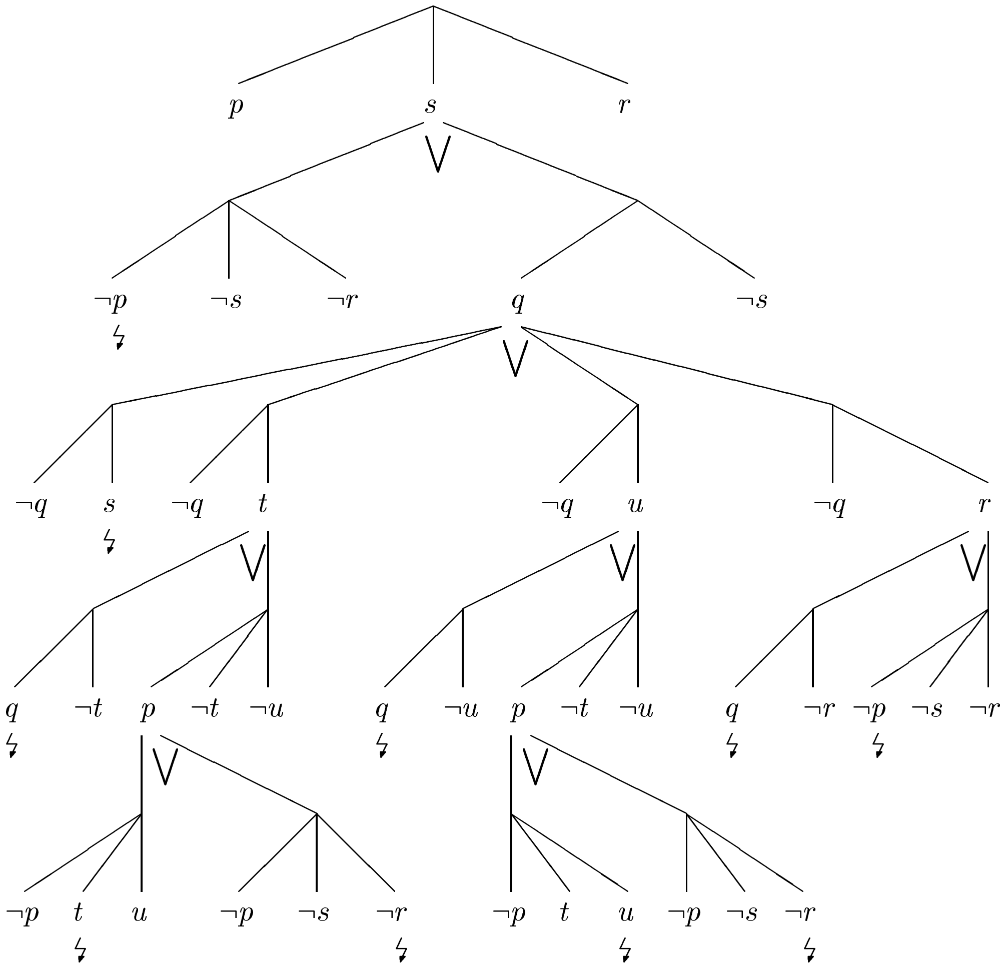

p s r
¬p ¬s ¬r q ¬s
↯
¬q s ¬q t ¬q u ¬q r
↯
q ¬t p ¬t ¬u q ¬u p ¬t ¬u q ¬r ¬p ¬s ¬r
↯ ↯ ↯ ↯
¬p t u ¬p ¬s ¬r ¬p t u ¬p ¬s ¬r
↯ ↯ ↯ ↯

Figure 3.24: Incompleteness for some selection functions of the strongly regular connection tableau calculus with enforced folding up.

clause formula c' exists for an extension step which is relevant in $(P \cup \{L\}) \sun S$, and we are done. Otherwise, brother subgoals of N might have to be solved first, leading to an increase of the context of N. The relevance of c in $P \sun S$ entails the existence of a subset S' of S such that c is essential in $P \sun S'$. Now we permit only solutions of the brother nodes of N using clause formulae from $S' \setminus \{c\}$ for extension; such solutions exist due to the completeness of the regularity restriction. By the soundness of the folding up rule, during such solutions of the brothers of N only literals can be inserted above N which are logically implied by the satisfiable set $(P \sun (S' \setminus \{c\})$. Consequently, L must be relevant in the increased context too, and the second case reduces to the first one. The successful termination of any tableau construction satisfying the mentioned properties follows from the relevance of the top clause formula c in S and from the fact that for any input set only regular tableaux of finite depths exist. □

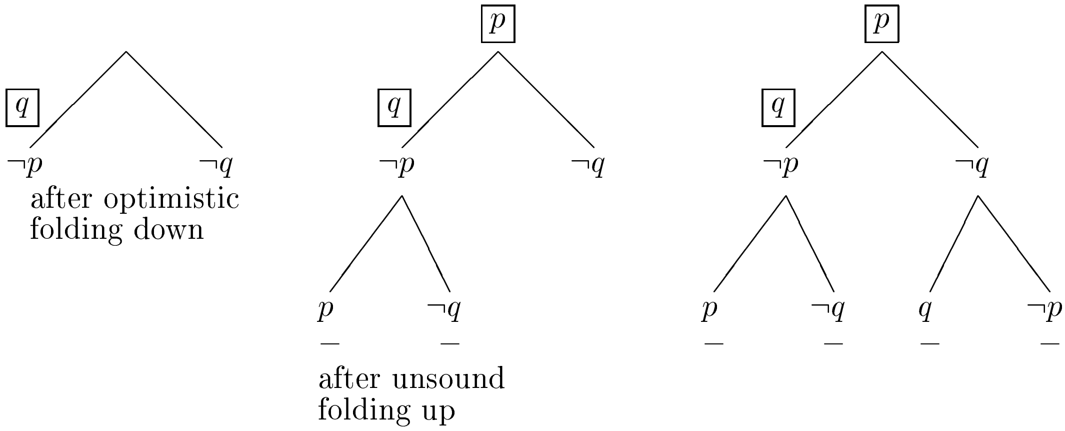
The new calculus promises to play an important role in the practice of automated deduction. While, concerning indeterministic power, the calculus is definitely superior to the regular connection tableaux, it may also be better off concerning search pruning.

Combining Folding Up and Folding Down

The interesting question may be raised whether it is possible to combine the pessimistic folding up rule with the optimistic folding down rule. We explain the combination for depth-first selection functions. Whenever a subgoal is selected for solution, before the solution process is started, all other unsolved brother nodes are folded down to the edge leading to N. The additional literals on the edge increase the number of inference possibilities, but they also increase the possibilities for a failure of the strong regularity test, and hence achieve additional search pruning. A naïve combination of folding down with the folding up rule, however, immediately results in an unsound calculus, as illustrated in Figure 3.25 with a "refutation" of the satisfiable set of clause formulae

$$\{\,\lrcorner\neg p, \neg q\llcorner\,, \lrcorner\neg p, q\llcorner\,, \lrcorner\neg q, p\llcorner\,\}.$$

In the incorrect deduction, the $\neg p$-subgoal is selected first for solution. Before it is solved the unsolved $\neg q$-subgoal is folded down to the edge above the $\neg p$-subgoal. Then the latter is solved using the framed literal q. Thereupon, according to the way the reduction and folding up operations have been defined, the $\neg p$-subgoal may be folded up to the root; this is the unsound operation. Afterwards, the $\neg q$-subgoal can be solved using the framed literal p.



after optimistic folding down

after unsound folding up

Figure 3.25: An unsound combination of folding up and folding down.

We briefly sketch how a sound combination of folding up and folding down could be achieved. Apparently, the literals inserted into the labels of the edges by folding down operations need to be treated differently. The easiest solution would be to explicitly use the simulation of folding down by cut illustrated in Figure 3.23 on p. 144. In this simulation the dependency structure of the optimistic folding down rule is expressed in a pessimistic manner, which is compatible with the folding up rule.

Chapter 4

First-Order Calculi

In this chapter we study and develop first-order calculi and proof procedures which are most suited for automated deduction. In the first section the Herbrand compactness property of first-order logic is reviewed, and proof procedures are described that can be viewed as direct algorithmic transpositions of the compactness property. Also, we introduce the notion of *Herbrand complexity*, which is an important lower bound on the sizes of proofs in many calculi, termed *Herbrand calculi*. The second section is devoted to proving some fundamental results on first-order resolution. On the one hand, it is shown that, due to the possibility of renaming lemmata, resolution is not polynomially bounded by Herbrand complexity, and hence superior to Herbrand calculi concerning indeterministic power. On the other hand, however, the renaming of lemmata destroys the polynomial transparency of resolution, both in the strong and in the weak sense. In Section 3, the first-order versions of connection tableau calculi are introduced, which by their very nature are Herbrand calculi. It is shown that in the first-order case new powerful pruning methods may be applied that can be implemented very efficiently using syntactic disequation constraints. Furthermore, the reductive potential of optimizing tableau node selection functions is exhibited. This demonstrates the superiority of the tableau format over more restricted frameworks. In the fourth section proof procedures based on connection tableau calculi are presented, which are fundamentally different from resolution proof procedures. Using the matings framework, a new important *global* search pruning method is developed and integrated into the connection tableau format. We conclude with formally identifying a general source of redundancy contained in any decomposition-based logic calculi. This redundancy motivates the future development of *additional* global search pruning methods using information coming from the proof process itself.

4.1 Herbrand Procedures

First-order logic differs from ground or 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 logical validity of first-order formulae[1] (or the unsatisfiability of sets of first-order formulae) whereas, when subscribing to *Church's Thesis*, the non-validity of first-order formulae (or the satisfiability of sets of first-order formulae) is not effectively recognizable[2].

We will concentrate on the indirect case of proving the *unsatisfiability* of sets of first-order formulae. Also note that we will assume throughout the whole chapter that a definitional logical language is available and whenever (sets of) expressions are substituted or unified, the *definitional* application of substitutions is to be performed or a polynomial unification algorithm is to be used, respectively. The existence at least of semi-decision procedures is due to a particular property of first-order logic, namely, its compactness.

4.1.1 The Compactness Property

Definition 4.1.1 (Herbrand base ordering) A *Herbrand base ordering* $\prec$ *on* a set S of Skolem formulae is a strict linear ordering on the Herbrand base of S.

Proposition 4.1.1 *For every set S of Skolem formulae there exists a Herbrand base ordering on S.*

Definition 4.1.2 (Herbrand interpretation tree) Given a Herbrand base ordering $\prec$ on a set S of Skolem formulae. The *Herbrand interpretation tree* T for $\prec$ is the semantic tree defined as follows. All branches in T have the same lengths, namely, the cardinality of the Herbrand base of S, and the edges outgoing of every node N are labelled with the n-th atom in $\prec$ and its negation, respectively, where $n-1$ is the depth of N in T. We say that T is a Herbrand interpretation tree *of* S.

The branches of a Herbrand interpretion tree of a set S of Skolem formulae encode precisely the Herbrand interpretations existing for S.

Theorem 4.1.2 (Compactness Theorem) *Any unsatisfiable set of Skolem formulae has a finite unsatisfiable subset.*

Proof Let $\prec$ be any Herbrand base ordering on S, and T the Herbrand interpretation tree for $\prec$. A node N in T is called a *failure node* for S if there exists a formula Φ in S such that all Herbrand interpretations falsify Φ which are corresponding to the branches with the initial segment from the root up to N. We say that Φ *closes* the path from the root up to N. Because of the one-to-one correspondence between branches and Herbrand interpretations, the unsatisfiability of S entails that every branch in T must pass through a failure node. Now,

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

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

we may cut all parts of the Herbrand interpretation tree which are dominated by failure nodes; the resulting labelled tree T' is called the *failure tree* of T for S. By *König's Lemma* [Knuth, 1968], T' must be finite. Now, any finite subset S' of S must be unsatisfiable which contains for any of the finitely many branches in T' a formula from S which closes the respective branch. □

Definition 4.1.3 (Herbrand instance) Given a formula Φ with matrix F in a set of Skolem formulae S. A ground formula F' is called a *Herbrand instance of* Φ *wrt* S if there exist a variable substitution σ into the Herbrand universe of S such that $F\sigma = F'$.

Theorem 4.1.3 (Herbrand Instance Theorem) *For any unsatisfiable set* S *of Skolem formulae there exists a finite unsatisfiable set* S' *of ground formulae such that every formula in* S' *is a Herbrand instance of a formula in* S.

Proof Due to Proposition 1.7.8 on p. 58 and Definition 1.2.18 item 8 on p. 10, a set S of Skolem formulae is unsatisfiable if and only if the union S' of the sets of Herbrand instances of its elements wrt S is unsatisfiable. By the Compactness Theorem 4.1.2, S' must have a finite unsatisfiable subset. □

4.1.2 Direct Herbrand Procedures

The first attempts [Gilmore, 1960, Davis and Putnam, 1960, Davis et al., 1962] to devise and implement proof procedures for first-order logic can be viewed as direct mechanizations of the Herbrand Instance Theorem. Such procedures consist of two relatively loosely coupled subprocedures. Given a set S of Skolem formulae, the first subprocedure selects a set S' of Herbrand instances of the formulae in the input set, while the second subprocedure is simply a decision procedure for ground formulae. When the second procedure detects the unsatisfiability of S', the unsatisfiability of S has been demonstrated. When the second procedure outputs the satisfiability of S', however, nothing is said about the actual logical status of the input. In this case the first procedure must select another set of Herbrand instances. In order to obtain completeness, the first subprocedure needs to *enumerate* increasing sets of ground formulae. Corresponding to the two subprocedures, there are two types of problems which render direct Herbrand procedures unsuccessful for automated deduction. On the one hand, the enumeration routine either may enumerate too many satisfiable sets of Herbrand instances before it arrives at the first unsatisfiable one, or the first encountered unsatisfiable set of Herbrand instances may be too large. On the other hand, the ground decison procedure may need too much time to determine the logical status of the input sets.

4.1.3 Improved Herbrand Procedures

Originating with [Prawitz, 1960], a significant improvement of naïve Herbrand procedures could be achieved, although the approach is still subscribing to the two-step methodology. The improvement is best formulated in the framework of *matings* [Bibel, 1981, Bibel, 1987, Andrews, 1981], which were introduced in Subsection 3.3.5 on pp. 120. First, we have to generalize the matings terminology to the first-order case. Recall that a *path* in a set of proper clause formulae S is a set of literal occurrences in S, exactly one from each clause formula in S, that any subset of a path in S is called a *subpath* in S, and that a *connection* in S is a two-element subpath in S whose literals have different signs and equal predicate symbols.

Definition 4.1.4 (Unifiable connection, mate] Given a finite set of proper clause formulae S. A connection $C = \{\langle K, i, c_1\rangle, \langle L, j, c_2\rangle\}$ in S is said to be *unifiable* if there is a variable substitution σ with $K\sigma = {\sim}L\sigma$; C is said to be *weakly unifiable* if there are variable substitutions σ and τ with $K\sigma = {\sim}L\tau$. The literal occurrences in a (weakly) unifiable connection are called *(weakly) unifiable mates* of each other.

Definition 4.1.5 (Unifiable mating) Given a finite set of clause formulae S. Any set M of connections in S is called *unifiable* or a *unifiable mating for* S if there exists a substitution σ such that for every pair of two connected literals K and L in M: $K\sigma = {\sim}L\sigma$.

Definition 4.1.6 (Compound instance, multiplicity] Given a set S of clause formulae, a *compound instance* of S is any finite set S' whose elements are all instances of matrices of clause formulae in S. A compound instance S' of S is called a *multiplicity* of S if all its elements are (variable-renamed) variants of matrices of clause formulae in S.

The matings characterization of unsatisfiability for the first-order case is expressed in the following proposition.

Proposition 4.1.4 *A set S of proper clause formulae is unsatisfiable if and only if there exists a unifiable spanning mating for a multiplicity of S.*

Proof Let S be an unsatisfiable set of proper clause formulae. By the Herbrand Instance Theorem 4.1.3, there exists a finite unsatisfiable set S' of clause formulae which are ground instances of the matrices of the clause formulae in S. S' is a compound (ground) instance of S. Proposition 3.3.12 on p. 121 guarantees the existence of a complementary spanning mating M for S'. Now, a unifiable spanning mating for a multiplicity of S can be easily constructed from S' and M; obtain a multiplicity S'' of S by taking, for any ground clause formula c in S' which is a ground instance of the matrix of a clause formula Φ in S', a new disjointly renamed variant F' of the matrix of Φ'; then, obtain the mating M' by replacing every connection in M with the connection between the respective renamed matrices of clause formulae in S''. Since there is a ground substitution σ which unifies M', M' is a unifiable spanning mating for S''. □

For any multiplicity only finitely many matings exist, therefore, we can immediately infer the following proposition.

Proposition 4.1.5 *It is decidable whether a multiplicity of a set of clause formulae has a unifiable spanning mating.*

This property motivates a significant improvement of the two-step methodology of direct Herbrand procedures. Instead of enumerating sets of Herbrand instances, one enumerates multiplicities. Example 4.1.1 illustrates that in most cases this approach is superior to the enumeration of ground instances.

Example 4.1.1 Given a set S containing two clause formulae of the structures

$\forall x_1 \cdots \forall x_n \lrcorner P(x_1, \ldots, x_n, a_1, \ldots, a_n) \llcorner$, and
$\forall y_1 \cdots \forall y_n \lrcorner \neg P(a_1, \ldots, a_n, y_1, \ldots, y_n) \llcorner$.

For each clause formula in S there exist n^n Herbrand instances, but only one of them (per clause formula) may contribute to a refutation. There is a low probability that the right instances are chosen early if sets of ground instances are enumerated blindly. In the matings approach, however, one would start with the multiplicity consisting of the two matrices itself of the formulae in S and obtain a unifiable spanning mating within one step.

Note It should be noted however that, according to the current state of knowledge, the problem of verifying the existence of a unifiable spanning mating for a multiplicity is *more difficult* than the problem of verifying the unsatisfiability of a set of ground clause formulae. This is because the latter is a coNP-complete problem, whereas the former is complete for the *union* of coNP and NP. This is because any *constraint satisfaction problem* can be viewed (or polynomially reformulated) as a problem of finding a unifiable spanning mating. Furthermore, even the improved method is not optimally suited for the purposes of automated deduction. The weakness is that the selection of a multiplicity and the search for a unifiable spanning mating are separated subprograms. In order to arrive at a successful system both subprograms need to be interleaved more closely. The connection tableau calculi discussed later on can be viewed as such intimate interleavings of the generation of multiplicities and the examination whether they have unifiable spanning matings.

4.1.4 Herbrand Complexity and Herbrand Calculi

A fundamental property which both approaches mentioned above have in common, the direct and the improved one, is that the size of any refutation of an input set S is bounded from below by some unsatisfiable set of Herbrand instances of the input S.

Definition 4.1.7 (Herbrand complexity) The *Herbrand complexity* of an unsatisfiable set S of Skolem formulae is the minimum of the sizes of the unsatisfiable sets of Herbrand instances of S.[3]

Herbrand complexity can be used to characterize so-called *Herbrand calculi.*

Definition 4.1.8 (Herbrand calculus) A *Herbrand calculus* is any calculus for refuting sets of Skolem formulae in which the size of any refutation D of a set S is bounded from below by the Herbrand complexity of S.

Herbrand calculi are extremely weak concerning proof lengths if compared with logic calculi of the traditional generative type. This is expressed in the following proposition, which was proved in [Statman, 1979].

Proposition 4.1.6 *There exists an infinite class $\{S_1, S_2, S_3, \ldots\}$ of unsatisfiable sets of formulae such that the smallest Herbrand instance S_i' of any set S_i from the class has a size which is not bounded by an elementary function of the size of a proof in a first-order Frege/Hilbert or sequent system of the negation of an appropriate translation of S_i.*

In the next section, we shall discuss *first-order resolution*, which does not fulfill the requirement of a Herbrand calculus.

4.2 First-Order Resolution

With the advent of the resolution calculus [Robinson, 1965a], the development of Herbrand calculi was pushed into the background until the beginning of the eithies. While the resolution paradigm, at least in its moulding as a forward reasoning approach, is not suited for propositional or ground logic, the incorporation of unification renders the calculus successful for first-order logic.[4]

4.2.1 Resolution with Unification and Factoring

Resolution for sets of first-order clauses is generated from ground resolution by the incorporation of two mechanism, namely, unification and factoring.

Definition 4.2.1 (Resolution rule) Given two clauses c_1 and c_2, and two clauses $r_1 = \{K_1, \ldots, K_m\}$ and $r_2 = \{L_1, \ldots, L_n\}$ with $r_1 \cap c_1 = \emptyset$ and $r_2 \cap c_2 = \emptyset$. The *resolution rule* has the shape:

$$\frac{\{K_1, \ldots, K_m\} \cup c_1 \qquad\qquad \{L_1, \ldots, L_n\} \cup c_2}{(c_1 \cup c_2\tau)\sigma}$$

[3] Under the assumption that the Herbrand instances are formulated using definitional expressions.

[4] Apparently, this can only be the case because the problems considered as relevant for first-order logic are of a completely different type than the ones considered as relevant for ground or propositional logic.

where τ is a renaming substitution which renders the variables in $r_2 \cup c_2$ disjoint from the variables in $r_1 \cup c_1$, and σ is a unifier for $r_1 \cup \{\sim L_1, \ldots, \sim L_n\}$. The clause $(c_1 \cup c_2\tau)\sigma$ is called a *resolvent of* $r_1 \cup c_1$ and $r_2 \cup c_2$ *over* r_1 and r_2, and $r_1 \cup c_1$ and $r_2 \cup c_2$ are termed *parent clauses* of the resolvent.

Note First-order resolution is much more complex than ground resolution, particularly concerning the contained indeterminism. While in the ground case for every pair of parent clauses at most *one* non-tautological resolvent exists, in the first-order case there may be *exponentially many* of them, as illustrated with the two clauses $\{P(x_1, \ldots, x_n), Q(x_1), \ldots, Q(x_n)\}$ and $\{\neg Q(y)\}$. The main reason is the factoring rule, which permits to group together any unifiable subset of a clause. Unfortunately, by omitting factoring and resolving over single literals only one loses completeness. This can be verified with the two clauses $\{P(x,y), P(y,x)\}$ and $\{\neg P(u,v), \neg P(v,u)\}$. In Subsection 4.2.6 it is shown that the *unrestricted* factoring rule is responsible for the fact that resolution can never be made polynomially transparent.

Fortunately, the complexity of a single resolution step is under control.

Proposition 4.2.1 *Given any two clauses c_1 and c_2, the time needed for indeterministically computing any resolvent c of c_1 and c_2 is polynomially* ($\mathrm{O}(n \log n)$) *bounded by the size of c_1 and c_2, and* $\mathrm{size}(c) < \mathrm{size}(c_1) + \mathrm{size}(c_2) - 1$*, where as the size of a clause we take the number of symbol occurrences.*

Proof The time complexity of resolution is due to the complexity of the unification operation plus the complexity of *merging* identical literals in the resolvent. The size bound follows from Proposition 1.6.8 on p. 53 and the fact that resolution *removes* at least two literals from the input. □

The proof objects for first-order resolution, *resolution deductions*, *resolution dags*, and *resolution trees*, are defined in analogy to the ones for the ground case on pp. 93.

Resolution is sound and refutation-complete for finite unsatisfiable sets of clauses.

Proposition 4.2.2 (Soundness of resolution) *If there is a resolution proof of a clause c from a set of clauses S, then $S \models c$.*

Proof Similar to the one for the ground case (p. 95). □

Lemma 4.2.3 (Lifting Lemma) *Given two first-order clauses c_1 and c_2, a ground substitution σ, and a renaming substitution τ making the variables in c_2 disjoint from the variables in c_1. Then, for any ground resolvent c of $c_1\sigma$ and $c_2\tau\sigma$, there exist a first-order resolvent c' of c_1 and c_2 and a substitution σ' with $c'\sigma' = c$.*

Proof Suppose c is a ground resolvent of $c_1\sigma$ and $c_2\tau\sigma$. Let r_1 and r_2 be the sets of literals in c_1 and $c_2\tau$ unified by the substitution σ, respectively. The existence of a first-order resolvent c' of c_1 and c_2 over r_1 and r_2 which meets the demanded property follows immediately from the Unification Theorem 1.5.12 on p. 40. □

Proposition 4.2.4 (Completeness of first-order resolution) *For any unsatisfiable set S of first-order clauses there exists a refutation of S by first-order resolution.*

Proof By the Herbrand Instance Theorem there is an unsatisfiable set S' of Herbrand instances of the clauses in S. The completeness of ground resolution guarantees the existence of a ground resolution deduction $D = (c_1, \ldots, c_n)$ of S' with $c_n = \emptyset$. Now, an iterative application of the Lifting Lemma assures that there is a first-order resolution deduction $D' = (c_1', \ldots, c_n')$ of S in which every clause c_i', $1 \leq i \leq n$, can be instantiated to c_1, respectively. Consequently, c_n must be the empty clause, and D' a first-order resolution refutation of S'. □

The properties of ground purity and ground subsumption can be lifted to the first-order case in a straightforward manner.

Definition 4.2.2 (Purity) Let L be a literal in a clause c of a set of clauses S. The literal occurrence L_c is called

1. *strongly pure in S* if the literal $\sim L$ is not weakly unifiable[5] with a literal K in a clause of S,
2. *pure in S* if the literal $\sim L$ is not weakly unifiable with some literal K in another clause of S,
3. *weakly pure in S* if, for any subset r of c containing the literal L, each resolvent of c with some other clause c' in S over r and some subset of c' is tautological.

Proposition 4.2.5 (Purity deletion) *Let L be a literal in a clause c of an unsatisfiable set of clauses S. If L_c is strongly pure, pure, or weakly pure in S, then $S \setminus \{c\}$ is unsatisfiable.*

Definition 4.2.3 (Subsumption) Given two clauses c_1 and c_2. We say that c_1 *(properly) subsumes* c_2 if there is a variable substitution σ such that $c_1\sigma$ is a (proper) subset of c_2.

Properly subsumed clauses may be deleted, due to the following fact.

Proposition 4.2.6 (Subsumption reduction) *If a clause c is subsumed by another clause in a set S of clauses, then $S \equiv (S \setminus \{c\})$.*

Note Although the question whether a clause subsumes another one is an NP-complete problem [Kapur and Narendran, 1986], the subsumption problem is much simpler than the *implication* problem between two clauses, which is undecidable.

[5] Two literals L and K are weakly unifiable if there are variable substitutions σ and τ with $L\sigma = K\tau$.

4.2.2 Refinements of Resolution

The resolution refinements of *linearity*, *regularity*, and the *tree* refinement considered in Subsection 3.2.6 on p. 104 and the *Davis/Putnam calculus* introduced in Subsection 3.2.5 on p. 100 can be lifted to the first-order case, resulting in complete first-order calculi. All of those, however, are not very successful in the practice of automated deduction. *Connection graph resolution* [Kowalski, 1975] can be viewed as an interesting generalization of the Davis/Putnam calculus. Instead of replacing a clause by *all* resolvents possible over a certain literal, in connection graph resolution the *connections* between the literals used in a resolution step are deleted and the deletion information is inherited, which gives more flexibility. Unfortunately, no practically useful *strong* completeness result is known for connection graph resolution. A concrete inference system based on connection graph resolution is the *Markgraf Karl Refutation Procedure* [Bläsius et al., 1981, Ohlbach and Siekmann, 1991].

There are various other refinements of resolution, particularly useful in practice being *hyper-resolution* [Robinson, 1965b] which is a special form of *semantic resolution* [Slagle, 1967]. Hyper-resolution seems to be the preferred strategy applied in the Otter system [McCune, 1988], which is currently the most widely used automatic theorem prover. A number of resolution refinements and deletion techniques like subsumption reduction are available in the system. Due to sophisticated implementation and indexing techniques, Otter can handle very large sets of clauses in an efficient way.

4.2.3 Resolution vs Herbrand Calculi

Resolution is not a Herbrand calculus according to the characterization given in Definition 4.1.8 on p. 154, that is, there may be resolution refutations for sets S of clauses which are significantly smaller in size than the smallest unsatisfiable Herbrand instance of (the clause formulae in) S. In fact, this even holds for linear resolution.

Proposition 4.2.7 *There is an infinite class C of clause sets such that, for any element $S \in C$, the Herbrand complexity of a set S' of clause formulae corresponding to S is exponential in the size of a shortest (linear) resolution refutation of the input S.*

Example 4.2.1 Consider a set $\{S_1, S_2, S_3, \ldots\}$ of sets of clauses with the following structures:

$$\begin{array}{llrrl}
c_1\colon & \{ & & P(0,\ldots,0) & \}, \\
c_2\colon & \{ & \neg P(x_1,\ldots,x_{n-1},0), & P(x_1,\ldots,x_{n-1},1) & \}, \\
c_3\colon & \{ & \neg P(x_1,\ldots,x_{n-2},0,1), & P(x_1,\ldots,x_{n-2},1,0) & \},
\end{array}$$

$$\begin{array}{rllr}
c_4\colon & \{ & \neg P(x_1,\ldots,x_{n-3},0,1,1), \quad P(x_1,\ldots,x_{n-3},1,0,0) & \}, \\
 & & \qquad\qquad\qquad \ldots & \\
c_n\colon & \{ & \neg P(x_1,0,1,\ldots,1), \qquad\qquad\qquad P(x_1,1,0,\ldots,0) & \}, \\
c_{n+1}\colon & \{ & \neg P(0,1,\ldots,1), \qquad\qquad\qquad\qquad P(1,0,\ldots,0) & \}, \\
c_{n+2}\colon & \{ & \neg P(1,\ldots,1) & \},
\end{array}$$

where P is an n-ary predicate symbol in the respective set and 0 and 1 denote constants.

Proof We use the clause set specified in Example 4.2.1. For any set S_n in the clause set, there exists a linear resolution refutation of $2n$ resolution steps, as illustrated for the case of $n = 4$, i.e., for the input set:

$$\begin{array}{rlllr}
c_1\colon & \{ & & P(0,0,0,0) & \}, \\
c_2\colon & \{ & \neg P(x_1,x_2,x_3,0), & P(x_1,x_2,x_3,1) & \}, \\
c_3\colon & \{ & \neg P(x_1,x_2,0,1), & P(x_1,x_2,1,0) & \}, \\
c_4\colon & \{ & \neg P(x_1,0,1,1), & P(x_1,1,0,0) & \}, \\
c_5\colon & \{ & \neg P(0,1,1,1), & P(1,0,0,0) & \}, \\
c_6\colon & \{ & \neg P(1,1,1,1) & & \},
\end{array}$$

The corresponding short linear resolution proof is the following:

$$\begin{array}{rlllrl}
c_7\colon & \{ & \neg P(x_1,x_2,0,1), & P(x_1,x_2,1,1) & \}, & (c_2.1,c_3.2) \\
c_8\colon & \{ & \neg P(x_1,x_2,0,0), & P(x_1,x_2,1,1) & \}, & (c_7.1,c_2.2) \\
c_9\colon & \{ & \neg P(x_1,0,1,1), & P(x_1,1,1,1) & \}, & (c_8.1,c_4.2) \\
c_{10}\colon & \{ & \neg P(x_1,0,0,0), & P(x_1,1,1,1) & \}, & (c_9.1,c_8.2) \\
c_{11}\colon & \{ & \neg P(0,1,1,1), & P(1,1,1,1) & \}, & (c_{10}.1,c_5.2) \\
c_{12}\colon & \{ & \neg P(0,0,0,0), & P(1,1,1,1) & \}, & (c_{11}.1,c_{10}.2) \\
c_{13}\colon & \{ & \neg P(0,0,0,0), & & \}, & (c_{12}.2,c_6.1) \\
c_{14}\colon & \{ & & & \}, & (c_{13}.1,c_1.1)
\end{array}$$

where on the right we have indicated the parent clauses and literals for deducing the respective resolvent. The smallest unsatisfiable Herbrand instance of the set of clause formulae corresponding to the clauses in S_4, however, consists of the following formulae:

$$\begin{array}{rlll}
c_1\colon & \lrcorner & & P(0,0,0,0) \llcorner, \\
c_2^1\colon & \lrcorner & \neg P(0,0,0,0), & P(0,0,0,1) \llcorner, \\
c_3^1\colon & \lrcorner & \neg P(0,0,0,1), & P(0,0,1,0) \llcorner, \\
c_2^2\colon & \lrcorner & \neg P(0,0,1,0), & P(0,0,1,1) \llcorner, \\
c_4^1\colon & \lrcorner & \neg P(0,0,1,1), & P(0,1,0,0) \llcorner, \\
c_2^3\colon & \lrcorner & \neg P(0,1,0,0), & P(0,1,0,1) \llcorner, \\
c_3^2\colon & \lrcorner & \neg P(0,1,0,1), & P(0,1,1,0) \llcorner, \\
c_2^4\colon & \lrcorner & \neg P(0,1,1,0), & P(0,1,1,1) \llcorner, \\
c_5\colon & \lrcorner & \neg P(0,1,1,1), & P(1,0,0,0) \llcorner, \\
c_2^5\colon & \lrcorner & \neg P(1,0,0,0), & P(1,0,0,1) \llcorner, \\
c_3^3\colon & \lrcorner & \neg P(1,0,0,1), & P(1,0,1,0) \llcorner, \\
c_2^6\colon & \lrcorner & \neg P(1,0,1,0), & P(1,0,1,1) \llcorner,
\end{array}$$

$$\begin{array}{lll}
c_4^2\colon & \rfloor\ \neg P(1,0,1,1), & P(1,1,0,0)\ \lfloor, \\
c_2^7\colon & \rfloor\ \neg P(1,1,0,0), & P(1,1,0,1)\ \lfloor, \\
c_3^4\colon & \rfloor\ \neg P(1,1,0,1), & P(1,1,1,0)\ \lfloor, \\
c_2^8\colon & \rfloor\ \neg P(1,1,1,0), & P(1,1,1,1)\ \lfloor, \\
c_6\colon & \rfloor\ \neg P(1,1,1,1), & \qquad\qquad\ \ \lfloor.
\end{array}$$

It is apparent that, for any element S_n, the numbers of instances of the input formulae $c_1, c_2, c_3, \ldots, c_{n-1}, c_n, c_{n+1}, c_{n+2}$ in the minimal unsatisfiable Herbrand instance of the corresponding set of clause formulae S'_n are $1, 2^{n-1}, 2^{n-2}, \ldots, 2^2, 2^1, 2^0, 1$, respectively, so that the total number of formulae in the Herbrand instance is $2^n + 1$. □

The given exponential bound is tight, which can be recognized along the following lines.

Proposition 4.2.8 *Tree resolution is a Herbrand calculus.*

Proof Given a tree resolution refutation for a set of clauses S, instantiate every variable occurring in the clauses of the tree with the same constant from the Herbrand universe of S. The set of clause formulae corresponding to the clauses at the leaves of the tree constitute an unsatisfiable Herbrand instance of the set of clause formulae corresponding to S. □

Proposition 4.2.9 *Tree resolution can exponentially simulate resolution.*

Corollary 4.2.10 *The size of any resolution refutation of a set S of clauses is exponentially bounded by the Herbrand complexity of the set of clause formulae corresponding to S'.*

Since resolution can maximally achieve an exponential speed-up with respect to Herbrand complexity, it is straightforward to prove that resolution is as weak as Herbrand calculi when compared with traditional logic calculi.

Corollary 4.2.11 *There exists an infinite class $\{S_1, S_2, S_3, \ldots\}$ of unsatisfiable sets of clauses such that the smallest resolution refutation of any S_i is not bounded by an elementary function of the size of a proof in a first-order Frege/Hilbert or sequent system of the negation of an appropriate translation of S_i.*

Note In [Baaz and Leitsch, 1992] it is shown that by adding appropriate new Skolem functions a nonelementary proof length reduction for resolution can be achieved.

The following result clarifies the relation between resolution and linear resolution.

Proposition 4.2.12 *Linear resolution cannot polynomially simulate resolution.*

Proof This can be shown by an easy modification of the class given in Example 4.2.1, using the same trick applied several times in the last chapter. Except for the first clause $\{P(0,\ldots,0)\}$, augment the set with a copy of each clause in which the predicate symbol P is renamed into another fixed predicate symbol P'. Then, modify the first clause by adding the atom $P'(0,\ldots,0)$. Now, any linear resolution deduction must first operate either in the P-part only or in the P'-part only, until eventually it uses the modified first clause as a parent clause. Any further deduction in the other part must inevitable generate ground instances (or superset of ground instances) of the clauses in the other part, so that the resulting refutation will be exponential. □

4.2.4 First-Order Resolution and Polynomial Transparency

In all investigations of the last section we have implicitly made use of the assumption that the number of inference steps of a resolution deduction give a representative measure for the actual size of the deduction. This assumption of the polynomial transparency of resolution was correct for the discussed examples. In general, however, this assumption cannot be made.

Proposition 4.2.13 *Resolution for first-order logic is not polynomially transparent.*

Example 4.2.2 Consider a set S of clauses of the structures

$$\begin{array}{l}\{\neg P(x)\},\\ \{P(s(x)),\neg P(x)\},\\ \{P(0)\},\end{array}$$

where 0 denotes a constant.

Proof We use the set of clause given in Example 4.2.2. By performing self-resolution on the second clause c_0 of S and then repeatedly applying self-resolution to the deduced resolvents, in k steps one can generate a clause c_k of size $> 2^k$. From c_k the empty clause can be deduced in two further resolution steps. Clearly for any polynomial p there exists a proof $D = (D_1,\ldots,D_m)$ of this type such that $\text{size}(D) > p(\text{size}(S), m)$, that is, the size of D cannot be bounded by any polynomial of the size of the input and the number of resolution steps. □

Consequently, in contrast to propositional logic, for first-order logic the number of resolution steps is not an adequate measure for the complexities of resolution derivations and proofs. The apparent reason is the following. Due to the renaming of derived clauses, resolution violates the logp size step-reliability (Definition 2.3.10 on p. 81).[6]

[6] It should be emphasized that the reason is indeed the *renaming* of derived clauses and not their *multiple use* as parent clauses.

But one may object that a resolution proof of the specified type is not an optimal one, and that there exists a shorter resolution proof for S which immediately derives the empty clause, by simply resolving the two unit-clauses $\{\neg P(x)\}$ and $\{P(0)\}$. For this short proof the relation between the proof size and the proof steps is polynomial modulo the input size.

The question is now whether at least for some *short* resolution proofs the sizes and the inference steps are always polynomially related, or in our terminology, whether *weak* polynomial transparency can be guaranteed for resolution. Unfortunately, the answer to this question is no, too. There is an infinite class of clause sets for which *every* resolution proof is exponential in size with respect to the input formula, whereas there are proofs that get by on polynomially many resolution steps. Example 4.2.3 specifies a formula class with this property. Assume in the following that, for any $1 \leq i \leq n$, $\mathfrak{P}_i$ is the value of the i-th prime number, and that $s^k(x)$ abbreviates a term of the structure $\underbrace{s(\cdots s(}_{k-\text{times}} x)\cdots)$.

Example 4.2.3 For any positive integer n, let S_n denote a set of Horn clauses of the following structure:

$$\begin{array}{l}
\{\neg P_1(s(x)), \ldots, \neg P_n(s(x))\}, \\
\{P_1(s^{\mathfrak{P}_1}(x)), \neg P_1(x)\}, \\
\qquad \cdots \\
\{P_n(s^{\mathfrak{P}_n}(x)), \neg P_n(x)\}, \\
\{P_1(0)\}, \\
\quad \cdots \\
\{P_n(0)\}.
\end{array}$$

If in this class of Horn sets the function symbol s is interpreted as the successor function, and if the denotation of a predicate P_i is the set of natural numbers divisible by the i-th prime number, then such a set can be used to compute common multiples of primes. Apparently, from these considerations we can derive the following lemma.

Lemma 4.2.14 *Given a set S_n of the type specified in Example 4.2.3, let $c\,\theta$ be any ground instance of the first clause $c \in S_n$ such that $(S_n \setminus \{c\}) \cup \{c\,\theta\}$ is unsatisfiable. Then the largest occurring term in $c\,\theta$ must denote a common multiple of the first n prime numbers.*

Since the least common multiple of a sequence $\mathfrak{P}_1, \ldots, \mathfrak{P}_n$ of primes equals $\prod_{i=1}^{n} \mathfrak{P}_i$, the following result gains importance.

Lemma 4.2.15 *There is no polynomial p such that for every positive integer n: $p(\sum_{i=1}^{n} \mathfrak{P}_i) > \prod_{i=1}^{n} \mathfrak{P}_i$.*

Proof Consider the chain

$$\frac{\prod_{i=1}^{n}\mathfrak{P}_i}{\sum_{i=1}^{n}\mathfrak{P}_i} \geq \frac{\prod_{i=1}^{n}\mathfrak{P}_i}{n\cdot\mathfrak{P}_n} \geq \frac{1}{n}\prod_{i=1}^{n-1}\mathfrak{P}_i = \frac{2}{n}\prod_{i=2}^{n-1}\mathfrak{P}_i \approx \frac{2}{n}\prod_{i=1}^{n-1}(i\ln\mathfrak{P}_i) \geq \frac{2}{n}\prod_{i=2}^{n-1} i = \frac{2n!}{n^2}.$$

The only non-trivial step concerns the approximate equation. Here the famous result from analytic number theory is used that

$$\lim_{x\to\infty}\frac{\pi(x)\cdot\ln x}{x} = 1 \qquad (\star)$$

where π is the prime number function, i.e., $\pi(x)$ is the number of primes $\leq x$. Since $\pi(\mathfrak{P}_i) = i$, by substituting $\mathfrak{P}_i$ for x in $(\star)$ we get that $\mathfrak{P}_i \approx i\cdot\ln\mathfrak{P}_i$, which is what is employed in the chain above. □

An immediate consequence of this result is that $\prod_{i=1}^{n}\mathfrak{P}_i$ cannot be polynomially bounded by the size of the input formula S_n.

Lemma 4.2.16 *There is no polynomial p such that for every positive integer n: $p(\text{size}(S_n)) > \prod_{i=1}^{n}\mathfrak{P}_i$, where S_n is a set of the type specified in Example 4.2.3.*

The formula class described in Example 4.2.3 is intractable for resolution.

Proposition 4.2.17 *There is no polynomial p such that for every positive integer n: $p(\text{size}(S_n))$ is greater than the size of any resolution refutation of S_n.*

In the proof of this proposition we shall exploit the fact that the sets in the class consist of Horn clauses, for which the following lemma holds.

Lemma 4.2.18 *If t is a resolution refutation dag for a set of Horn clauses, then t contains one branch b—called* the negative branch*—on which exactly the* negative clauses *of the refutation lie, i.e., those clauses which are void of positive literals.*

Proof of Lemma 4.2.18 It suffices to notice that, on the one hand, in such a dag no non-negative clause can dominate a negative clause, and, on the other hand, every negative clause must be derived from a negative and a non-negative clause. □

Proof of Proposition 4.2.17 Let t be an arbitrary resolution refutation dag for a set S_n, and let b be the negative branch of t, which exists by Lemma 4.2.18. Clearly, each occurrence of a negative clause on b is used only once as a parent clause in t. Consequently, replacing all clauses on the branch b by appropriate ground instances does not alter the length of the branch, while the resulting dag remains a refutation—of resolution with *free*, i.e., not necessarily most general, unification rule. If this partial instantiation is performed on t, the negative branch b' of the resulting refutation dag t' must contain ground instances

$$\{\neg P_1(s^\xi(0)), \ldots, \neg P_n(s^\xi(0))\}$$

of the first clause $c \in S_n$. Let $c_0, \ldots, c_k$ be the clauses on the initial segment of the branch b' from the root labelled with c_0 (the empty clause) up to the first instance c_k of c. Obtain t'' by making c_k a leaf of t' (it may already be one) plus removing the nodes and edges which are no more accessible from the root. Apparently, t'' still remains a refutation dag. Since c_k is the only instance of c in t'', $(S_n \setminus \{c\}) \cup \{c_k\}$ must be unsatisfiable. From Lemma 4.2.14 follows that in c_k the maximal term depth $\xi \geq \prod_{i=1}^n \mathfrak{P}_i$. Consider now the non-negative clauses $s_1, \ldots, s_k$ in the refutation t'' which are resolution partners of the clauses $c_1, \ldots, c_k$ respectively—let us call those non-negative clauses the *side* clauses. The structure of S_n guarantees that each side clause either has the form

$$\{P_i(s^l(x)), \neg P_i(x)\}$$

or the form

$$\{P_i(s^l(0))\}.$$

Consequently, if ascending the branch b' by one step towards the root from c_i to c_{i-1}, $1 \leq i \leq k$, the clause size can only decrease by at most the size of the respective side clause s_i:

$$\text{size}(c_{i-1}) \geq \text{size}(c_i) - \text{size}(s_i).$$

Therefore,

$$\text{size}(c_0) \geq \text{size}(c_k) - \sum_{i=1}^k \text{size}(s_i).$$

Because $\text{size}(c_0) = 1$, and since the side clauses have not been modified by the partial instantiation operation, we get that

$$\text{size}(t) > \sum_{i=1}^k \text{size}(s_i) \geq \text{size}(c_k) - 1 > \prod_{i=1}^n \mathfrak{P}_i.$$

An application of Lemma 4.2.16 completes the proof. □

The existence of intractable formula classes for resolution is nothing exceptional, even for the propositional case (at least since Haken's work [Haken, 1985]). The special property of the class considered here concerns the relation between the proof sizes and the numbers of derivation steps. Although all resolution proofs for the sets in the class are superpolynomial, there are short proofs in terms of *inference steps*.

Proposition 4.2.19 *There is a polynomial p such that for every set S_n from the class specified in Example 4.2.3 there exists a resolution refutation $D_1, \ldots, D_m$ of S_n such that $m < p(\mathrm{size}(S_n))$.*

Proof Let $\xi = \prod_{i=1}^{n} \mathfrak{P}_i$, i.e., the least common multiple of the primes $\mathfrak{P}_1, \ldots, \mathfrak{P}_n$. Then a polynomial-step proof can be constructed as follows. For every clause of the type

$$\{P_i(s^{\mathfrak{P}_i}(x)), \neg P_i(x)\}$$

perform self-resolution and repeatedly apply self-resolution to the respective resolvents. Within k steps this operation deduces clauses in which the number of occurrences of the function symbol s in the positive literals successively takes the values $\mathfrak{P}_i\, 2^1, \mathfrak{P}_i\, 2^2, \ldots, \mathfrak{P}_i\, 2^k$. This is done as long as $\mathfrak{P}_i\, 2^k \le \xi$. Then, after at most k further resolution steps which use clauses from this derivation, each clause at most once, a clause of the structure

$$\{P_i(s^{\xi}(x)), \neg P_i(x)\}$$

can be deduced. Accordingly, for any $1 \le i \le n$, we need at most $2 \log_2 \frac{\xi}{\mathfrak{P}_i}$ steps, which is less than $2 \log_2 \xi$, hence for all i: less than $2n \log_2 \xi$. Lastly, in further $2n$ resolution steps the empty clause can be derived by resolving these clauses with the facts and the resulting n facts $P_i(s^{\xi}(0))$, $1 \le i \le n$, with the first clause. The whole refutation takes less than $2n + (2n \log_2 \xi) \le 4n \log_2 \xi$ steps. It remains to be shown that this value is polynomially bounded by the size of S_n. For this purpose we may just use $\zeta = \sum_{i=1}^{n} \mathfrak{P}_i$ as a lower bound for the size of S_n and consider the chain

$$4n \log_2 \prod_{i=1}^{n} \mathfrak{P}_i \le 4n \log_2 \left(\frac{\sum_{i=1}^{n} \mathfrak{P}_i}{n} \right)^{n} = 4n^2 \log_2 \frac{\zeta}{n} < 4\zeta^3.$$

The first inequality holds because of properties of the arithmetical mean, while the others are trivial. □

The Propositions 4.2.17 and 4.2.19 have as an immediate consequence that, even if only step-minimal proofs are considered, the number of steps of a resolution proof may not be a representative measure for the complexity of the proof.

Theorem 4.2.20 *Resolution for first-order logic is not weakly polynomially transparent.*

The violation of the logp size step-reliability turns out to be fatal, even if only short proofs are counted.

4.2.5 Improvements of the Representation of Formulae

The situation is quite instructive, because we can illustrate at the example of resolution the three principal solution methodologies when facing the polynomial intransparency of a transition relation $\vdash$.

The first approach is to weaken the transition relation $\vdash$ and to define a transition relation $\vdash'$, for example, by taking out each pair $\langle S, S' \rangle$ which violates the logp size step-reliability, since this may be the problematic property, like in the case of resolution. The most radical method to perform this modification on the resolution calculus is to forbid the renaming or even the multiple use of lemmata. The latter results in the calculus of *tree resolution*.

Proposition 4.2.21 *Tree resolution is polynomially transparent.*

Proof Let there be a resolution tree—i.e., an upward tree—T for a set of clauses S with bottom clause c_n computed with n resolution steps. The resolution tree has $n+1$ leave nodes $L_1, \ldots, L_{n+1}$ labelled with input clauses $s_1, \ldots, s_{n+1}$. By Proposition 4.2.1 on p. 155, for any clause c at a node N with successor nodes N_1 and N_2 labelled with parent clauses c_1 and c_2, $\text{size}(c) < \text{size}(c_1) + \text{size}(c_2)$. Due to the tree structure of T, $\text{size}(c_n) < \sum_{i=1}^{n+1} \text{size}(s_i) < (n{+}1) \times \text{size}(S)$. Consequently, $\text{size}(T) < (n{+}1)^2 \times \text{size}(S)$. □

Note Polynomial transparency also holds for another refinement of resolution, namely, *V-resolution* [Chang and Lee, 1973]. V-resolution is more powerful than tree resolution in that general resolution dags are permitted, but derived clauses must not be renamed and whenever a derived clause is used as a parent clause, then the resulting unifier must be applied to the clause and to the clauses derived from it.

Unfortunately, such weakenings of general resolution have the unacceptable consequence that many proofs are thrown out which are short in steps *and* small in size. This holds for the short resolution deductions discussed in Subsection 4.2.3. Also, eliminating problematic pairs from a transition relation does not work for arbitrary transition relations. This leads to the second alternative. In order to preserve the problem solving functionality of the relation, that is, to guarantee that the transitive closures—or at least the provable states—of both transition relations remain identical, in the general case, each problematic step must be replaced by a series of computationally innocuous steps. For logic calculi, this amounts to a redefinition of the notion of an inference step.

Both methods are relatively unappealing for the practical working with logic calculi, since in no case the *indeterministic power* of a calculus is increased, either it is weakened or it remains unchanged, and only the presentation structure of the calculus is modified. The *real* importance of the notion of polynomial transparency for the advance of science is that it can motivate research following the third approach. The third approach is to let the general structure of a transition relation as it is, and to try to remedy the polynomial intransparency of the transition relation. Since the typical stumble-block for attaining polynomial transparency is the violation of the logp size step-reliability, a promising research direction consists of improving the *data structures* of the elements in the transition relation in such a way that they can be represented with less space than in the original relation, with the hope to gain polynomial transparency this way. The advantage of such an attempt, if it succeeds, is that the distances between the elements in the transition relation can be preserved while the real computing cost and sizes properly decrease.

The difference between the solution methodologies is that the second approach always succeeds, whereas the third one may fail in principle. This case will be considered below (in Subsection 4.2.6).

Number Terms in the Object Language

Similar to the case of the unification operation, which, in order to attain the polynomial time step-reliability of an inference system, has enforced the necessity to represent logical terms as dags, one should think about the development of more sophisticated mechanisms which would admit a notation for resolvents polynomially bounded in size by the number of their derivation steps, with respect to the input set. An obvious improvement is to integrate into the object language the same vocabulary of upper indices we already used in our meta-language for the purpose of polynomially specifying terms of exponential depth. It is apparent that with the use of such *number terms* the transparency problems of the Examples 4.2.2 and 4.2.3 can be solved, even polynomial transparency in the strong sense can be achieved for these examples. One can predict that number terms will play an important role in future automated deduction systems.[7]

We shall not pursue further the attempt of extending the representation of logical formulae, instead we want to present a critical example class which may turn out to be a hard problem for the efforts to achieve polynomial transparency. These new formulae are obtained from the previous class of Example 4.2.3 by augmenting the arity of the function symbol s by 1. This means that the previous formula class is just an abstraction of the new class.

Example 4.2.4 For any positive integer n, let S_n denote a set of Horn clauses of the following structure:

$$\begin{array}{l}\{\neg P_1(s(x,y)),\ldots,\neg P_n(s(x,y))\},\\ \{P_1(s(s(x,y_1),y_2)),\neg P_1(x)\},\\ \{P_2(s(s(s(x,y_1),y_2),y_3)),\neg P_2(x)\},\end{array}$$

[7] Much more than polynomial unification algorithms, which have turned out to be relatively unimportant for the practice of deduction systems. This can be verified by observing that the examples (particularly Example 4.2.2) for demonstrating the necessity of number terms are much simpler and occur more frequently in practice than the ones which demand polynomial unification techniques.

$$
\begin{aligned}
&\{P_3(s(s(s(s(s(x,y_1),y_2),y_3),y_4),y_5)),\neg P_3(x)\},\\
&\quad\cdots\\
&\{P_n(\underbrace{s(s(\cdots s(s(}_{\mathfrak{P}_n-\text{times}}x,y_1),y_2),\cdots,y_{n-1}),y_n)),\neg P_n(x)\},\\
&\{P_1(0)\},\\
&\quad\cdots\\
&\{P_n(0)\}.
\end{aligned}
$$

In the new class the second argument of the function symbol s does not play any role at all, the variables at these positions are just *dummy* variables. Consequently, the results concerning proof steps and proof lengths carry over from Example 4.2.3 to this example. But there is a crucial difference between both examples, which becomes apparent when self-resolution is applied to a clause of the mixed type in Example 4.2.4. Let us demonstrate this with the input clause corresponding to the prime number 3:

$$\{P_2(s(s(s(x,y_1),y_2),y_3)),\neg P_2(x)\}.$$

In its self-resolvent

$$\{P_2(s(s(s(s(s(s(x,y_1),y_2),y_3),y_4),y_5),y_6)),\neg P_2(x)\}$$

the number of distinct dummy variables has doubled. In general, in any such self-resolution step the resolvent contains $2n-1$ more distinct variables than the original clause. Accordingly, for this class of clause sets, in any polynomial-step proof of an instance S_n, clauses are needed in which not only the term depth is exponential (which could be remedied by using number terms in the object language) but also the number of *distinct* variables. And to this problem no obvious solution is in sight.[8]

4.2.6 The Impossibility of Resolution Transparency

Although the current data structures for resolution do not achieve the *weak* polynomial transparency of resolution, we have no apparent reason to abandon hope that such data structures might exist. For the case of the *strong* polynomial transparency, however, according to which for *every* resolution deduction the inference steps must provide a representative complexity measure of the deduction, one can prove that such data structures cannot exist.

[8] There seems to be an interesting analogy between decidability and complexity properties with respect to the distinction of clause formulae containing unary function symbols only from those containing binary function symbols. While the former are decidable and permit the successful application of number terms, the latter are undecidable and polynomial transparency cannot be achieved using number terms.

Proposition 4.2.22 (Impossibility of resolution transparency) *It is impossible to render resolution polynomially transparent without having to increase the distances in the resolution transition relation.*

Example 4.2.5 Consider the set S of three clauses of the following shapes

$$\begin{array}{l}\{P(x\,.\,y), \neg P(x), \neg P(y)\},\\ \{P(0)\},\\ \{P(1)\}\end{array}$$

where 0 and 1 are constants and $s\,.\,t$ denotes a term of the structure $f(s,t)$, for some binary function symbol f.

Proof The iterative application of self-resolution to the first clause in Example 4.2.5 and to the resulting resolvents, after n steps produces a clause of the structure

$$\{P(x_1\,.\,\cdots\,.\,x_{2^n+1}), \neg P(x_1), \ldots, \neg P(x_{2^n+1})\}.$$

In two further resolution steps, extensively employing factoring and using the two other clauses in the input set, any positive unit clause of the form

$$\{P(c_1\,.\,\cdots\,.\,c_{2^n+1})\}, \qquad c_i \in \{0,1\}, \qquad \text{for } 1 \leq i \leq c_{2^n+1},$$

may be deduced. The set of unit clauses derivable in this manner can be viewed to encode the set $\mathcal{S}$ of all strings of lengths c_{2^n+1} over the alphabet $\{0,1\}$. If a data structure or general technique would exist rendering resolution polynomially transparent *without* increasing the original number of inference steps in the calculus, then it must be possible to encode any of the strings in the set $\mathcal{S}$ with a size polynomially bounded by the input size and $n+2$. This, however, contradicts elementary facts of Kolmogorov complexity theory [Li and Vitányi, 1990]. □

The apparent reason for the impossibility of making resolution polynomially transparent is the factoring rule, which may render a highly regular structure strongly irregular within a single inference step, or, in terms of Kolmogorov complexity theory, factoring can turn a regular string into a *random* string within a single step. Consequently, in order to remedy the intransparency of resolution, the factoring rule need to be restricted. A further interesting open question is whether the problems with the factoring rule also have an influence on the *weak* polynomial transparency of resolution.

4.3 First-Order Connection Tableaux

In contrast to the standard way of generalizing the tableau calculus from the ground case to the first-order case, by including different rules for quantifier elimination [Smullyan, 1968], the working with Skolemized formulae renders the first-order calculus significantly simpler and also facilitates the incorporation of unification into the tableau calculus. We consider *clausal* first-order tableaux.

4.3.1 Clausal First-Order Tableaux

Definition 4.3.1 (Clausal first-order tableau) A *clausal first-order tableau for* a finite set S of proper clause formulae is a pair $\langle t, \lambda \rangle$ consisting of an ordered tree t and a labelling function λ on its nodes such that the root is labelled with the verum $\top$, and each successor set of nodes $N_1, \ldots, N_n$ is labelled with literals $K_1, \ldots, K_n$ such that there exists a variable substitution σ and a clause formula $\forall x_1 \cdots \forall x_m \rfloor L_1, \ldots, L_n \lfloor$ in S with $K_i = L_i\sigma$, for $1 \leq i \leq n$.

The notions of tableau (top) clause formula, the closedness of a tableau, and marked tableaux can be transmitted unchanged from the ground case (Section 3.3).

Definition 4.3.2 (First-order connection tableau) A *first-order connection tableau* for a finite set S of proper clause formulae is a first-order clausal tableau for S in which each inner node N labelled with a literal L has a leaf node N' among its successor nodes which is labelled with the literal $\sim\!L$.

The static specifications of first-order tableaux and connection tableaux put no particular restrictions on the instantiations that may be applied to the renamed clause formulae from the input set in their use as tableau clause formulae. The procedural counterparts of the static deduction objects, however, shall be defined using *unification* as instantiation operation, this way achieving finite branching rates of the calculi. The two inference rules of the *tableau calculus with unification* are the following straightforward generalizations of the inference rules for the ground case. Again, we shall work with marked tableaux.

Procedure 4.3.1 (First-order tableau expansion) Given a set S of proper clause formulae as input and a marked first-order tableau T for S, choose a leaf node N which is not marked, select a clause formula $c \in S$, obtain a variant $\rfloor L_1, \ldots, L_n \lfloor$ of the matrix of c in which the variables are disjoint from the variables in the literals occurring in T and in any predecessor tableau of T,[9] then attach n new successor nodes $N_1, \ldots, N_n$ to N and label them with $L_1, \ldots, L_n$ respectively.

Procedure 4.3.2 (Tableau reduction with unification) Given a marked tableau T, choose an unmarked leaf node N with literal L, select a dominating node N' with literal L' such that there is a most general unifier σ for $\{\sim\!L, L'\}$, then apply the substitution σ to the tableau literals,[10] and mark N with N'.

The *connection tableau calculus with unification* consists of three inference rules, the tableau reduction rule with unification plus the following two inference rules.

[9] Such renamings can easily be achieved without having to look at the tableau each time, namely, by carrying along a counter which is incremented whenever a new clause formula is chosen for expansion.

[10] Again we presuppose the working with definitional expressions and the definitional application of substitutions.

Procedure 4.3.3 [First-order tableau start] Given a set of proper clause formulae S as input and a one-node tree with root N and label $\top$, simply perform a first-order tableau expansion step.

Procedure 4.3.4 [Tableau extension with unification] Given a set of proper clause formulae S as input and a marked connection tableau T for S, choose a leaf node N with literal L which is not marked, apply a tableau expansion step at N, select a node N' among the immediate successors of N, perform a tableau reduction step at N' with the predecessor N, and mark N' with N.

Although with the two (three) inference rules of the (connection) tableau calculus with unification not *every* first-order tableau can be generated, the inference rules are adequate with respect to the static specifications of first-order (connection) tableaux, in the following manner.

Proposition 4.3.1 *The first-order (connection) tableau calculus can only generate marked first-order (connection) tableaux, and conversely, given any marked (connection) tableau T for a set of clause formulae, then, for any node selection function, there exists a sequence of inference steps in the (connection) tableau calculus with unification and an output tableau T' which is isomorphic to T and more general than T.*[11]

Proof The fact that the (connection) tableau calculus with unification can only generate first-order connection tableaux is obvious. For the converse, let T be a marked first-order (connection) tableau for an input set S with m marked nodes. It is apparent that, ignoring the arguments of the literals, the respective propositional marked (connection) tableau skeleton T' of T can be constructed by the propositional (connection) tableau calculus, for any selection function. From T' obtain a tableau T'' by adding the arguments of the renamed input clauses. By the definition of first-order (connection) tableaux, there exists a substitution σ which, when applied to the literals in T'', produces T. Let $c_1 = \rfloor L_1, \ldots, L_m \lfloor$ be a clause formula consisting of the literals at the marked nodes $N_1, \ldots, N_m$ respectively in T'', and $c_2 = \rfloor K_1, \ldots, K_m \lfloor$ the clause formula, in which, for $1 \leq i \leq m$, K_i is the complement of the literal at the node by which N_i is marked in T''. The substitution σ must be a unifier for $\{c_1, c_2\}$. Now, the sequence of unification steps to be performed for any selection function in the (connection) tableau calculus with unification in order to obtain a more general tableau than T with the skeleton T' can be viewed as a single unification operation of the set $\{c_1, c_2\}$. The different selection functions just reflect certain different selections of disagreement sets.[12] By the Unification Theorem 1.5.12 on p. 40, any selection function produces a unifier which is more general than σ. □

[11] A tableau T' is *more general* than an isomorphic tableau T if there is a substitution σ such that for any literal L occurring at a node N in T, if L' is the literal at the node corresponding to N, then $L = L'\sigma$.

[12] In fact, the existing tableau node selections functions do not even exploit the *full* freedom of selection possible in the unification process.

4.3.2 The Completeness of First-Order Connection Tableaux

While the regularity restriction can be transmitted unchanged from the ground case to the first-order case, the lifting of the *strong* connectedness condition (Definition 3.4.7 on p. 132), however, is a more delicate problem. A *direct* transmission to the first-order case leads to incompleteness, as illustrated in Figure 4.1 with the unsatisfiable set of clause formulae $\rfloor P(x,y), P(y,x) \lfloor$ and $\rfloor \neg P(u,v), \neg P(v,u) \lfloor$, for which no closed and strongly connected first-order tableau exists.

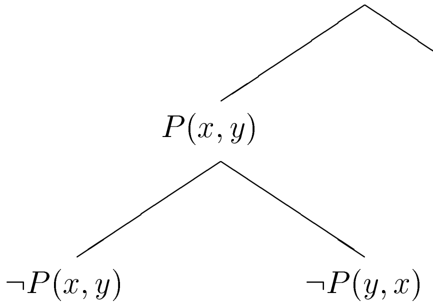


Figure 4.1: The incompleteness of the strong connectedness for first-order logic.

The apparent reason for the incompleteness of the strong connectedness in the first-order case is that a strong connection between certain *instances* of two clause formulae need not be strong for the *original* formulae. This consideration also shows how to weaken the strong connectedness in order to preserve completeness for the first-order case.

Definition 4.3.3 (Potential strong connectedness) A connection $\{\langle L, i, c_1\rangle, \langle K, j, c_2\rangle\}$ is called *potentially strong* if there exists a substitution σ such that $\{\langle L\sigma, i, c_1\sigma\rangle, \langle K\sigma, j, c_2\sigma\rangle\}$ is a strong connection. A first-order tableau T is called *potentially strongly connected* if T has a substitution instance which is strongly connected.

Example 4.3.1 In two clause formulae of the form $\rfloor \neg Q(x), \neg P(y,x) \lfloor$ and $\rfloor Q(v), P(w,v) \lfloor$ the first literals in each formula are strongly connected while the others are not.

The weaker variant of strong connectedness retains the eliminative effect on tableaux, as shown in Figure 4.2. Thus, if the first clause from Example 4.3.1 appears as a tableau clause, then the second clause must not be attached to the node labelled with $\neg P(y,x)$. The second clause can be attached to the node labelled with $\neg Q(x)$ without violating the condition. Note, however, that if in the second case a subsequent unification step enforces the variables y and w to be unified, then the potential strong connectedness is violated too, since no strongly connected instance of the resulting tableau exists.

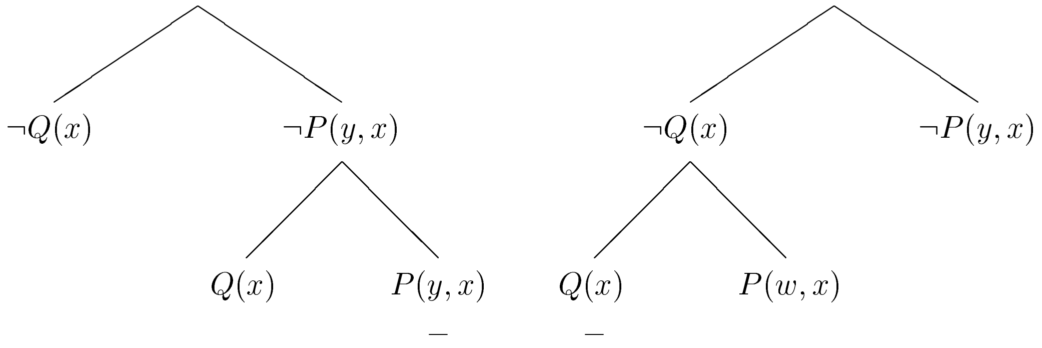


Figure 4.2: Violation (left) and satisfaction (right) of the potential strong connectedness for tableaux employing the clause formulae from Example 4.3.1.

Theorem 4.3.2 (Completeness of regular strongly connected first-order tableaux) *For any finite unsatisfiable set S of proper clause formulae and any clause formula c which is relevant in S, there exists a closed regular strongly connected first-order tableau for S with an instance of the matrix of c as top clause formula.*

Proof By the Herbrand Instance Theorem (p. 151) there exists a finite unsatisfiable set S' of ground formulae such that every formula in S' is a Herbrand instance of a formula in S. Due to the completeness of regular strong connection tableaux for the ground case (Theorem 3.4.6 on p.132), there exists a closed regular strong connection tableau T for S'. T is a closed regular strongly connected first-order tableau for S. □

Corollary 4.3.3 (Completeness of the connection tableau calculus with unification for regular potentially strongly connected tableaux) *For any finite unsatisfiable set S of proper clause formulae and any clause formula c which is relevant in S, there exists a refutation T of S in the connection tableau calculus with unification in which c is used as expansion clause in the start step and T is regular and potentially strongly connected.*

Proof Immediate from Theorem 4.3.2 and Proposition 4.3.1. □

It is clear that the size of a closed first-order tableau for a set of clause formulae S is bounded from below by the Herbrand complexity of S, so that all first-order tableau calculi are Herbrand calculi. Also, the number of inference steps in the (connection) tableau calculus with unification is a representative measure for the size of the deduction.

Proposition 4.3.4 *The (connection) tableau calculus with unification is polynomially transparent.*

Proof An application of Lemma 2.3.7 on p. 81 will do. The polynomial time step-reliability of the calculi is obvious if definitional expressions and polynomial unification techniques are used. To recognize the logp size step-reliability, we assume that dag expressions (Definition 1.4.10 on p. 23) are used. As the size of a tableau we take the sum of the sizes of the literal occurrences in the tableaux. Then, every expansion step increases the tableau size at most by the size of the largest clause formula from the input set, which is a constant increase rate for any given input. Due to the dag format, by Proposition 1.6.8 on p. 53, a reduction step does not change the size of a tableau. Finally, the size increase resulting from an extensions step is the size increase of the contained expansion step. □

4.3.3 Dynamic Pruning of First-Order Tableaux

In the first-order case besides regularity interesting new useful search pruning techniques are applicable, which have no significance in the ground case.

Tautology Elimination

Normally, it is a good strategy to eliminate certain clause formulae from the input set which can be shown to be redundant for finding a refutation. Tautological clause formulae are of such a sort.[13] In the ground case tautologies may be eliminated once and for ever in a preprocessing phase, 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 a clause formula expressing the transitivity of a relation.

Example 4.3.2 (Transitivity) $\forall x \forall y \forall z \ \rfloor \neg P(x,y), \neg P(y,z), P(x,z) \lfloor$.

Suppose that during the construction of a tableau this clause formula is used in an extension step—for the sake of the argument let us take the clause formula itself and assume that the *rest* of the tableau be renamed. Suppose further that after some subsequent inference steps the variables y and z are instantiated to the same term t. Then a tautological instance $\rfloor \neg P(x,t), \neg P(t,t), P(x,t) \lfloor$ of the transitivity formula has been generated. Apparently, 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.

Note 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 parent 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 clause), but not for non-Horn clauses.

[13] Although tautologies may render the *basic* calculus stronger concerning indeterministic power, as shown in the last chapter.

Subsumption Reduction

An essential pruning method in resolution theorem proving is subsumption reduction, 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 clause formulae are generated in the tableau approach, the *forward* variant of subsumption reduction can be exploited in the (connection) tableau framework, too. First, we have to say what subsumption means for *clause formulae.*

Definition 4.3.4 (Subsumption for clause formulae) Given two clause formulae c_1 and c_2. We say that c_1 *(properly) subsumes* c_2 if there is a variable substitution σ such that the set of literals contained in $c_1\sigma$ is a (proper) subset of the set of literals contained in c_2.

Similar to the dynamic generation of tautologies, it may happen, that a substitution instance of a clause formula is created which is properly subsumed by another clause formula from the input set. To give an example, suppose the transitivity formula from above and a unit clause formula $\rfloor P(a,b) \lfloor$ be contained in the input set. If now the transitivity formula 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 formula $\rfloor \neg P(a,y), \neg P(y,b), P(a,b) \lfloor$ is properly subsumed by $\rfloor P(a,b) \lfloor$. Apparently, for any closed tableau using the former tableau formula a smaller closed tableau exists which uses the latter instead.

Note Again, there is the possibility of a pruning overlap with the regularity and the non-tautology conditions. It should be emphasized, however, that subsumption reduction is not a pure *tableau structure* restriction, since a case of proper subsumption cannot be defined by merely looking at the tableau. Additionally, it is necessary to take the respective input set into account. Consequently, subsumption reduction is not a *monotonic* reduction rule in the sense defined in Subsection 3.4.6 (pp. 130).

4.3.4 Syntactic Disequation Constraints

The question may be raised whether in the first-order case it is always possible with tenable cost to check the tableau conditions of regularity, tautology, and subsumption-freeness 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 all the mentioned search pruning mechanisms, namely, *syntactic disequation constraints.*

Let us illustrate the technique first at the example of the dynamic tautology elimination. Using the transitivity formula

$$\rfloor \neg P(x,y), \neg P(y,z), P(x,z) \lfloor$$

from above, there are two classes of instantiations which may render instances of the formula tautological. Either x and y are instantiated to the same term, or y and z. Apparently, 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 *disequations* of the form $(s_1, \ldots, s_n) \neq (t_1, \ldots, t_n)$, where the s_i and t_i are terms. A disequation contraint is violated if *every* pair $\langle s_i, t_i \rangle$ in the constraint is instantiated to the same term t_i, respectively. In the transitivity example above 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 tableaux construction works with *constrained clause formulae*. Whenever, a constrained clause formula is used for tableau expansion, then the formula and its constraints are consistently renamed, the expansion is performed with the formula part and the constraints part is integrated into a special constraint store.

Regularity can also be captured using disequation constraints. Obviously, regularity constraints have to be generated dynamically. Whenever a new renaming c of a (constrained) clause formula is attached to a branch b by expansion, then for every literal $L = [\sim]P(s_1, \ldots, s_n)$ contained in the formula part of c, disequation constraints of the shape $(s_1, \ldots, s_n) \neq (t_1, \ldots, t_n)$ are generated where the $(t_1, \ldots, t_n)$ are the argument sequences of literals appearing on b with the predicate symbol P and the same sign as L.

Subsumption is essentially treated in the same manner as tautology. Recall the example from above where in addition to the transitivity formula a unit clause formula $\rfloor P(a, b) \lfloor$ is supposed to be contained 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.[14] It is apparent that constraints resulting from different sources—tautology, regularity, or subsumption—need not be distinguished in the tableau construction. In order to capture *all* cases of subsumption, however, a new type of terms, so-called *structure variables*, need to be introduced. To explain the necessity for doing this, assume that the transitivity formula and a unit clause formula of the shape $\rfloor P(f(v), g(v)) \lfloor$ 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 in the constraint a variable is contained which does not occur in the transitivity formula. Since formulae are always renamed before integrated into a tableaux, the variable v will not occur as an ordinary variable in a tableau, so that the constraint is absolutely

[14] 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.

useless, since it can never be violated. Apparently, 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 terms which have the *structures* $f(t)$ and $g(t)$, respectively. This can be conveniently achieved by adding *structure variables*, denoted with a '#' before the variable name, which are distinguished from ordinary variables by the constraint handler. The respective disequation constraint $(x, z) \neq (f(\#v), g(\#v))$ then is violated if x and z are instantiated to terms of the structures $f(s)$ and $g(t)$ where $s = t$.

Note With the theorem prover SETHEO [Letz et al., 1992] it could be experimentally verified that the deletion of irregular tableaux and tableaux containing tautological or properly subsumed formulae may reduce the search space by magnitudes, although no complete constraint handling was implemented in the system. In the new Version 3.0 (Spring 1993) of SETHEO the full constraint mechanism is integrated. The new system demonstrates that disequation constraint information can be generated, stored, updated, and examined in a very efficient way.

The keeping of the constraint information alongside the tableau in a special constraint store also facilitates the working with subgoal trees instead of tableaux, since all relevant structure information of the solved part of a tableau is contained in the constraint part.

4.3.5 Search Trees and Selection Functions

There is a source of indeterminism in the discussed tableau calculi which can be removed without any harm concerning indeterministic power, namely, the choice of the tableau node selection function being employed when building up a tableau. Therefore, it is reasonable to consider tableau calculi in which this indeterminism is not contained any more.

Definition 4.3.5 (Determined tableau calculus) A *determined tableau calculus* is a pair $\langle C, \phi \rangle$ consisting of a tableau calculus C and a tableau node selection function ϕ.

Any determined tableau calculus uniquely determines the *search tree* of a given input set S of clause formulae.

Definition 4.3.6 ((Tableau) search tree) Let S be a set of clause formulae and $\mathcal{C} = \langle C, \phi \rangle$ a determined tableau calculus. The *(tableau) search tree* of S in $\mathcal{C}$ is a tree $\mathcal{T}$ labelled with tableaux defined as follows.

1. The root of $\mathcal{T}$ is labelled with the trivial tableau, consisting of a root node only.

2. Every non-leaf node 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 tableau node in T selected by ϕ, and the successor nodes are labelled with the respective resulting tableaux.

The leaf nodes of a (tableau) search tree can be partitioned into two sets of nodes, the ones labelled with tableaux that are marked as closed, called *success nodes*, and the others which are labelled with open tableaux to which no successful inference steps can be applied, called *failure nodes*.

In a *pure* deduction enumeration approach, according to which *all* possible deductions are examined,[15] the part of the search tree down to the first proof can be taken as a useful approximation of the *actual cost* of finding a proof using the underlying determined calculus.

Definition 4.3.7 (Relevant part of a search tree) Let $\mathcal{T}$ be a search tree and let n be the minimal distance of a success node from the root of $\mathcal{T}$. The *relevant part* of the search tree $\mathcal{T}$ is the subtree obtained from $\mathcal{T}$ by cutting off all nodes with a depth $> n$.

For any determined tableau calculus $\mathcal{C}$, the complexity of the relevant part of the search tree of a given input set S in $\mathcal{C}$ can be taken as the *actual* complexity of the calculus $\mathcal{C}$ for the input S.[16] It is important to emphasize that a variation of the selection function can dramatically change the actual complexities of the calculus. This gives rise to the application of *heuristic* methods in the definition of selection functions. Due to its greater freedom of choosing between selection functions, the (connection) tableau format is superior to frameworks supporting depth-first selection functions only, like connection matrices and model elimination chains. This can be demonstrated formally as follows.

Proposition 4.3.5 *Let C be the regular connection tableau calculus with unification. There exist sets S of formulae for which the (relevant part of) the search tree is exponential in size wrt S for any determined calculus $\langle C, \phi \rangle$ using a depth-first selection function ϕ, whereas there are search trees linear in size for some free selection functions.*

Proof We use Example 4.3.3 and start the tableau construction with the relevant top clause formula $\rfloor \neg P(x,y), \neg P(y,x) \lfloor$. Any depth-first selection function inevitable runs into the exponential search space induced by S'. Using a free selection function, however, after the first extension step with $\rfloor P(x,b), \neg R(x) \lfloor$,

[15] In the next section the natural limitations of pure tableau enumeration procedures with respect to search pruning will be investigated and the theoretical reasons will be given why with a pure deduction enumeration method it is impossible *in principle* to remove all redundancies contained in proof search.

[16] Or at least as an interesting *upper bound* for the actual complexity of the calculus.

one may shift to the other subgoal in the top clause formula and perform again an extension step with $\rfloor P(x,b), \neg R(x)\lfloor$. After the second step the clause formula $\rfloor R(a), \Phi\lfloor$ is no more accessible and the search tree is linear. □

Example 4.3.3 Let S be a set consisting of the union of the clause formulae

$$\begin{array}{l} \rfloor \neg P(x,y), \neg P(y,x)\lfloor, \\ \rfloor P(x,b), \neg R(x)\lfloor, \\ \rfloor R(a), \Phi\lfloor, \\ \rfloor R(x), \neg Q(f^n(x))\lfloor, \\ \rfloor Q(f(x)), \neg Q(x)\lfloor, \\ \rfloor Q(b)\lfloor, \end{array}$$

and a set S' containing clause formulae with connections to Φ only such that the search tree of $S' \cup \{\rfloor\Phi\lfloor\}$ has exponentially many nodes with a depth $\leq n$, for any determined (regular connection) tableau calculus.

4.3.6 Extensions of First-Order Connection Tableaux

The transmission of the factorization rule and the folding up and folding down operations from the ground case to the first-order case is straightforward. To obtain first-order *factorization*, one simply has to generalize the ground factorization rule by performing unification between the respective literals. The cases of first-order *folding up* and *folding down* are even trivial, since the ground folding up and down need not to be changed at all; the first-order variants are achieved by using the reduction rule with unification.

In the first-order case, however, a further significant difference appears between the folding up operation and the explicit storing of lemmata beside a tableau (as described in [Letz et al., 1992] and [Loveland, 1978]). When a lemma c which has been dynamically added to the input set is used in a subsequent extension step, then the variables in c may be soundly renamed as in any extension step using input formulae. According to the folding up operation, however, a (unit) lemma is stored in the tableau itself, and all usages of the lemma must have a common substitution instance, as illustrated with the following Example 4.3.4.

Example 4.3.4

$$\begin{array}{l} \rfloor \neg P(x), \neg P(a_1), \ldots, \neg P(a_n)\lfloor, \\ \rfloor P(x), \neg Q(x)\lfloor, \\ \rfloor P(x), Q(x)\lfloor. \end{array}$$

Assume that in the construction of a connection tableau we are starting with the top clause formula $\rfloor \neg P(x), \neg P(a_1), \ldots, \neg P(a_n)\lfloor$, select the subgoal $\neg P(x)$, and solve it completely. Then, the folding operation puts the literal $P(x)$ into the label set of the root node. Now, the second subgoal $P(a_1)$ can be solved by a reduction step using $P(x)$. But in the reduction step x gets instantiated to a_1, so that afterwards the lemma is no more available for reduction steps from the other subgoals. The lemmata made available by the folding up operation are just *single-instance* lemmata. Using a standard lemma technique, which explicitly would add the formula $\forall x \rfloor P(x) \lfloor$ to the input set, one could solve every other subgoal in the top clause formula with a single extension step using different renamings of $\rfloor P(x) \lfloor$. So in the first-order case an additional difference concerning proof lengths comes in, which is not present in the ground case.

Note The single-instance property of folding up guarantees that the accordingly extended first-order (connection) tableau calculi remain polynomially transparent. Also, single-instance lemmata can be *implemented* in an extremely efficient way, since no copying is necessary. The price of this restriction is that the calculi remain Herbrand calculi, that is, the Herbrand complexity of any unsatisfiable set of clause formulae is a lower bound to the size of any refutation in the extended first-order (connection) tableau calculi. The standard technique of explicitly adding derived lemmata to the input set, however, renders the first-order calculi polynomially intransparent, induces higher branching rates of the search spaces, and demands more expensive implementation techniques. On the other hand, the sizes of refutations in those calculi are not polynomially bounded by the Herbrand complexity of an input set, so that significantly shorter proofs may exist than for the polynomially transparent versions. Which of the two versions will turn out to be superior in practice depends on the examples considered as relevant.

One could also think about a *multiple-instance* variant of folding up. The basic problem to be solved in such an approach is that the renaming of the variables in a literal folded up to an edge must be limited in certain ways in order to preserve soundness, as demonstrated in Figure 4.3 for the *satisfiable* input set given in Example 4.3.5. Referring to the figure, suppose that after two extension steps and one reduction step the subgoal $\neg Q(x)$ is solved completely, and is folded up to the edge above the node labelled with $\neg P(x)$. If then the unmarked subgoal labelled with $\neg Q(b)$ is permitted to be solved with a *renaming* of the context unit lemma $Q(x)$, i.e., *without* instantiating the variable x to b, then the subgoal labelled with $\neg P(x)$ would have been solved in an unsound manner, correctly it should be instantiated to $P(b)$. Afterwards, the subgoal labelled with $\neg R(x)$ could be solved by an extension step, which would not be possible if it be correctly labelled with $\neg R(b)$.

Example 4.3.5 Consider a satisfiable set of clause formulae of the form

$$
\begin{array}{l}
\rfloor \neg P(x), \neg R(x) \lfloor, \\
\rfloor P(x), \neg Q(x), \neg Q(b) \lfloor, \\
\rfloor P(x), Q(x) \lfloor, \\
\rfloor R(a) \lfloor.
\end{array}
$$

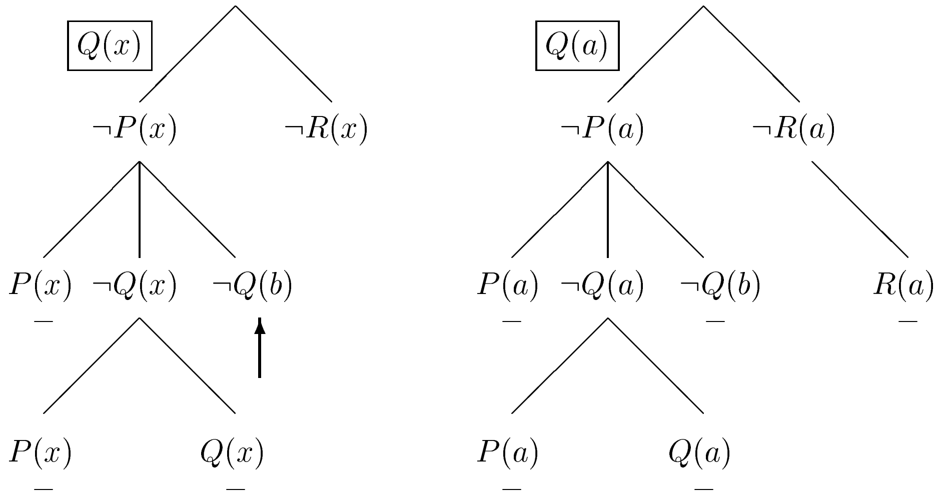


Figure 4.3: Unsoundness of the renaming of context unit lemmata.

4.4 Connection Tableaux Procedures

In contrast to the most successful style of resolution theorem proving, which is based on *formula* enumeration or saturation procedures, such an approach is not possible in the connection tableau framework, because, unlike resolution and unlike the tableau calculi without the connectedness condition, the *connection* tableau calculi are not *proof-confluent*, that is, not every proof attempt can be completed successfully. This possibility of making irreversible decisions in the calculus demands a different organization of the proof process, namely, as a *proof* enumeration instead of a *formula* enumeration procedure.

4.4.1 Explicit Tableau Enumeration

In Subsection 3.4.3 we have introduced the notion of *subgoal formulae*, according to which every *subgoal tree* of a tableau can be encoded as a formula.The process of tableau construction could therefore be viewed as the *enumeration* of subgoal formulae, with the objective to derive the logical falsum, just as in the standard formula saturation procedures using resolution calculi. Accordingly, one could design connection tableau proof procedures just in the same manner resolution procedures are constructed, the difference from resolution being that one would have to handle sets of subgoal formulae instead of sets of clauses. Also, new subgoal formulae would not be derived by performing inference operations *between* subgoal formulae but between a subgoal formula and an input clause formula. This manifests the *linear* approach of connection tableau calculi. A proof procedure for tableaux or subgoal formulae could be achieved by simply exploring the search tree of a determined tableau calculus in a breadth-first manner starting from the root.

Tableau Subsumption

One important strategy to achieve search pruning in a proof enumeration setting is to refine the *calculus* in such a way that the number of possible proof attempts with a certain resource decreases, which has been our favourite pruning strategy up to know. Another strategy is to improve the *proof procedure* so that information coming from the proof search itself can be used to even eliminate proof attempts not excluded by the calculus. This is motivated, since certain redundancies in proof search can only be detected when *comparing different deductions*, by considering the tableau search tree. Let N_1 and N_2 be two nodes in a search tree $\mathcal{T}$ labelled with tableaux (or subgoal trees) T_1 and T_2 respectively. In case it can be seen that T_1 can only be completed to a refutation if T_2 can be completed to a refutation, then it is possible to ignore the entire subtree dominated by N_1. The simplest application possibility of such a search tree reduction is when T_1 and T_2 are identical. Interestingly, this very often occurs in practice, particularly when working with subgoal trees instead of tableaux.[17] This will be demonstrated on a formal level in the next section. But over and above identity of tableaux, a more reductive notion of redundancy between tableaux should be defined, in the spirit of the notion of *subsumption* for resolution procedures. This is an important topic for future research.

Consolution

As a matter of fact, inference operations could be performed *between* subgoal formulae, too. Such an approach is pursued with the *consolution calculus* [Eder, 1991], which can be viewed as a generalization both of the connection tableau and the resolution framework. The consolution calculus manipulates special normalized subgoal formulae, so-called *consolvents*, which result from simply transforming subgoal formulae into disjunctive normal form. In terms of subgoal trees, a consolvent is just the disjunction of the literals on the branches of a subgoal tree. In [Eder, 1991] a single consolution step is defined as a macro step consisting of the following operations, which are reformulated in the tableau framework here. Consolution takes two subgoal trees T_1 and T_2, renames the variables in T_2, and expands T_1 by attaching copies of the renamed tree[18] to *all* leaf nodes of T_1. On the resulting tree, arbitrary many reduction, factorization, and branch shortening steps may be applied. Input clause formulae are just treated as subgoal trees of depth 1. Resolution can then be viewed as a consolution *refinement* manipulating subgoal trees in which all branches are shortened to length 1.

[17] Note that if we are working with subgoal trees supplied with a set of disequation constraints to be satisfied, then the constraint parts must be taken into consideration when comparing different subgoal trees.

[18] One could even attach differently renamed copies, which would result in a further strengthening of consolution.

Consolution seems mainly useful as a framework for *comparing* calculi, since, apparently, consolution is not polynomially transparent, not even in the ground case. Thus, every unsatisfiable set of n ground clause formulae can be refuted within n consolution steps. This shows that the calculus needs a redefinition of what has to be counted as a single inference step, similar to the original definition of the Davis/Putnam calculus. A further interesting question to be investigated is whether consolution is superior to resolution concerning indeterministic power.

4.4.2 Tableau Enumeration by Backtracking

The explicit enumeration of tableaux or subgoal trees (formulae) suffers from two disadvantages. The first one is also present in the standard resolution procedures, namely, the extreme branching rate of the search tree, which very quickly leads to the situation that the available memory on a computer is exhausted. For tableaux or subgoal formulae, which are much more complex structures than clauses, an explicit enumeration procedure may even be practically impossible. The second disadvantage is that the cost for adding new tableaux or subgoal formulae significantly increases during the proof process as the sizes of the proof objects increase, which is not the case for resolution procedures. These weaknesses give sufficient reason why in practice no-one has seriously pursued an explicit tableau enumeration approach up to now.

Bounded Depth-First Iterative Deepening Search

A more successful paradigm is to perform tableau enumeration in an implicit manner, using *consecutively bounded depth-first iterative deepening search* procedures, as follows. The tableau search tree is cut by imposing conditions, called *completeness modes*, on the tableaux at the nodes of the tree. These conditions are monotonic, i.e., if a tableau T at a node N violates the conditions, then do all the tableaux in the search tree dominated by N. Let us introduce completeness modes formally.

Definition 4.4.1 (Completeness mode) A *completeness mode* is a total mapping m from the set of all tableaux to the set of natural numbers satisfying the following property. For every tableau search tree $\mathcal{T}$ and every $n \geq 0$,

1. there is a $k \geq 0$ such that for every node N in $\mathcal{T}$ with a depth $> k$: $m(T) > n$, for the tableau T at the node N, and
2. for every node N with label T in $\mathcal{T}$, if $m(T) \leq n$, then for every node N' with label T' dominating N: $m(T') \leq n$.

Apparently, given any completeness mode m, any natural number n, and any tableau search tree $\mathcal{T}$, there is a *finite* initial segment of $\mathcal{T}$ such that exactly the nodes in this segment have labels T with $m(T) \leq n$. Using a completeness mode

m and an initial natural number n, the proof procedure starts by completely exploring the finite initial segment of the search tree determined by m and n. If no success node is contained in the initial segment, n is incremented and the larger initial segment is explored, and so forth. Since m is assumed to be total on the set of all tableaux, it is guaranteed that eventually a proof will be found if a success node exists in the search tree. Due to the construction process of tableaux from the root to the leaves, many tableaux have identical or structurally identical subparts. This motivates to explore finite initial segments in a *depth-first* manner, by strongly exploiting *structure sharing* techniques. Accordingly, at each time only one tableau is in memory, which is extended following the branches of the search tree, and truncated, when a leaf node of the respective inital segment of the search tree is reached. Although according to this methodology initial parts of the search tree are explored multiply, no significant efficiency is lost if the initial segments increase exponentially [Korf, 1985].

The most natural completeness modes are the *number of inferences* (used in [Stickel, 1988]) and the *depth* of a tableau. In [Letz et al., 1992] results of an experimental comparison between both modes are given.

4.4.3 Permutability of Tableaux and The Matings Optimization

In proof procedures using a pure tableau enumeration approach a source of redundancy is contained which cannot be removed by methods of refining the tableau *calculi.* Calculus refinements like (strong) connectedness, (strong) regularity, tautology-, and subsumption-freeness are *local* pruning methods in the sense that the violation of the conditions can be determined from looking at the respective tableau only (plus at the input formula, in the case of subsumption), whereas reference to *alternative* tableaux is never needed to check the conditions. A more global view, however, by which certain tableaux are grouped together into equivalence classes, can reveal that it is not necessary to construct *all* tableaux in such a class but only *one representative* of the class. A particularly interesting notion of equivalence classes of tableaux is provided by the matings framework. In Figure 4.4, it is shown that for one and the same spanning mating for the input set $\{\rfloor\neg p\lfloor, \rfloor p, q\lfloor, \rfloor\neg q, p\lfloor\}$ there are two closed regular connection tableaux with the relevant formula $\rfloor\neg p\lfloor$ as top clause formula. Apparently, only one of the two tableaux need to be considered. 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.

Motivated by this observation, we propose a technique to avoid the multiple traversal of certain matings, by restricting the applicability of reduction steps.

Proposition 4.4.1 (Matings optimization) *Given any total ordering $\prec$ on the elements of an unsatisfiable set S of proper clause formulae. Starting with any relevant clause formula in S, there is a refutation T of S in the regular connection*

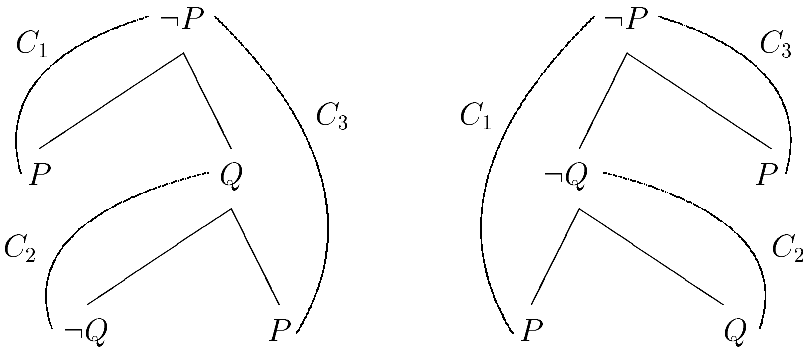


Figure 4.4: Two closed tableaux for one and the same spanning mating.

tableau calculus with unification satisfying the following property. Let $N_1, \ldots, N_n$ be any successor set of nodes in T labelled with literals $L_1, \ldots, L_n$ stemming from a clause formula $c \in S$. Then, at no node N_i, $1 \leq i \leq n$, reduction steps with a dominating node N are permitted at which an extension step with a clause $c' \prec c$ has been performed.

Proof Consider an unsatisfiable set S^* of Herbrand instances of S. The fact that the mentioned restriction of reduction steps preserves completeness is demonstrated by using the proof of Theorem 3.4.6 (pp. 132). In this proof it was demonstrated that, for any unmarked node N labelled with a literal L and on a branch with literal set P, an extension step can be performed with *any* clause formula containing $\sim\!L$ from a minimally unsatisfiable subset of P ☼ S^*. Let S' be such a minimally unsatisfiable subset and $S'_{\sim L}$ its subset of clause formulae containing $\sim\!L$. According to the mentioned completeness proof, we can always select a ground formula $c\sigma$ from $S'_{\sim L}$ for extension such that, for some *original* formula c in S of which $c\sigma$ is a Herbrand instance: $c \not\prec c'$, for all original formulae of the ground formulae in $S'_{\sim L}$. The lifting to the first-order case is trivial. □

Applied to the example shown in Figure 4.4, the matings optimization achieves that, for any clause ordering, one of the two tableaux is no more derivable. Since the multiple traversal of sets of connections occurs recursively in a pure tableau enumeration procedure, the matings optimization can result in an exponential search space reduction for tableau procedures. This illustrates the benefit of integrating different frameworks.

Incompatibility problems with the Strong Connectedness

Unfortunately, the matings optimization is not compatible with the strong connectedness condition on regular tableaux. As a counterexample, consider Example 4.4.1, using an ordering in which $\rfloor \neg Q(a), P \lfloor \prec \rfloor P, Q(a) \lfloor$.

Example 4.4.1 $\{ \rfloor P, Q(a) \lfloor,\ \rfloor \neg Q(a), P \lfloor,\ \rfloor \neg P, Q(a) \lfloor,\ \rfloor \neg P, \neg Q(x) \lfloor \}$

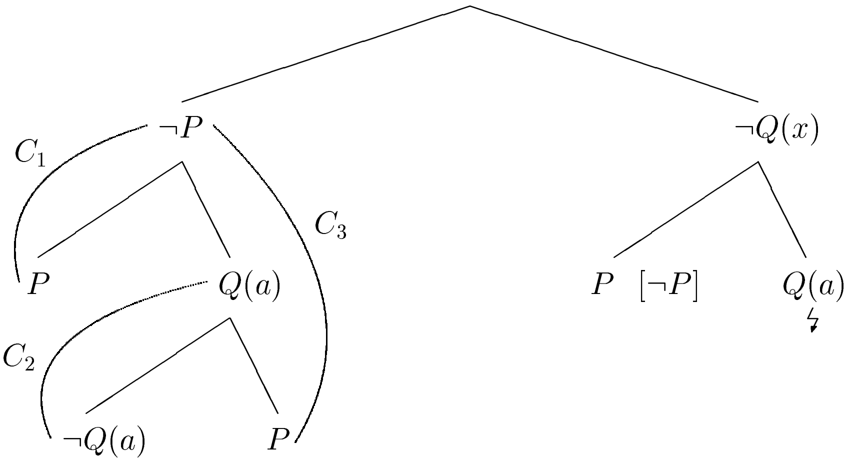


Figure 4.5: Deduction process for Example 4.4.1.

We describe the failure of finding a refutation for a backtracking-driven tableau procedure. If we take the fourth clause formula, which is relevant in the set, as top clause formula, enter the first clause formula, 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 connectedness condition. Therefore, backtracking has to occur, up to the state in which merely the top clause formula remains. Afterwards, only the second clause formula may be entered, followed by an extension step into the first one. But now the matings optimization forbids a reduction step at the subgoal labelled with P, since it would violate the given clause ordering and 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 formula as top clause formula.[19]

Consequently, there is a certain trade-off between pruning the calculus and pruning the proof procedure.

4.4.4 A General Limitation of Pruning the Calculus

Even if redundancies due to the *permutability* of proofs are eliminated, by using methods like the matings optimization, there still remains a lot of redundancy in the search tree which cannot be captured by *local* techniques. The fundamental reason for this redundancy is contained in the very nature of the logic calculi themselves which we are employing, namely, their methodology of separating a problem into subproblems and solving the subproblems separately.

[19] It is even possible to construct an example in which for *no* clause ordering a refutation exists.

The situation can be explained best using the terminology of *strengthenings* introduced in Definition 3.4.9 on p. 132. Apparently, if a set S of clause formulae is unsatisfiable then any strengthening of S by some set of literals $\{L_1, \ldots, L_n\}$ is also unsatisfiable. Furthermore, if $\{L\} \text{☼} S$ is a strengthening of S with L being contained in an essential clause formula of S, then the unit clause formula $\rfloor L \lfloor$ is essential in the strengthening $\{L\} \text{☼} S$.

In the process of demonstrating the unsatisfiability of a set of clause formulae using a top-down approach, for example, during the generation of a semantic tree or a tableau, we always implicitly make use of the strengthening operation, namely, whenever we perform the branching operation. In a bottom-up oriented procedure like resolution, of course, the strengthening operation is applied reversely, just in the way semantic trees are a reversed description of resolution trees.

We will now present a phenomenon of logic which sheds light on a problematic property of proof search. As we have already mentioned, it is crucial for the purposes of optimizing proof search to avoid as much redundancy as possible. Thus we should strive for identifying a minimally unsatisfiable subset of the input set under investigation, or, equivalently, a subset in which every relevant formula is essential. The problematic property of logic with respect to search pruning is that even if we have identified a minimally unsatisfiable subset of an input set, the strengthening process may introduce new redundancies, regardless whether it is applied in a forward or in a backward manner. Let us formulate this more precisely.

Proposition 4.4.2 *If a set of clause formulae S is minimally unsatisfiable, and L is a literal occurring in formulae of S, then the strengthening $\{L\} \text{☼} S$ may contain more than one minimally unsatisfiable subsets, or, equivalently, not every relevant clause in $\{L\} \text{☼} S$ may be essential.*

Proof We use a set S constructed by M. Schramm, which consists of the following propositional clause formulae

$$\begin{array}{l}
\rfloor p, \neg q \lfloor\,, \\
\rfloor p, \neg r, \neg s \lfloor\,, \\
\rfloor q, r \lfloor\,, \\
\rfloor q, s \lfloor\,, \\
\rfloor \neg p, q, \neg s \lfloor\,, \\
\rfloor \neg p, \neg r, s \lfloor\,, \\
\rfloor \neg p, \neg q, r \lfloor\,, \\
\rfloor \neg p, \neg q, \neg r, \neg s \lfloor\,.
\end{array}$$

S is minimally unsatisfiable, as shown with the table of interpretations in Figure 4.6. In the figure, overlining abbreviates negation, and writing literals side by side denotes disjunction. Also, in order to make the distinction between truth values more visible we have denoted the truth value $\top$ with $\bullet$ and the truth value

$\perp$ with $\circ$. The fact that each formula is essential in the initial set is expressed with boxes in the columns of the formulae which exclusively are falsified by the interpretation in the respective line. In the strengthening $\{p\}$ ☼ S the clause formulae $\rfloor q, r \lfloor$ and $\rfloor q, s \lfloor$ both are relevant but no more essential, since the new clause formula $\rfloor p \lfloor$ is also falsified by the interpretations which have rendered the formulae essential in S. □

				S								
p	q	r	s	$p\bar{q}$	$p\bar{r}\bar{s}$	qr	qs	$\bar{p}q\bar{s}$	$\bar{p}\bar{r}s$	$\bar{p}\bar{q}r$	$\bar{p}\bar{q}\bar{r}\bar{s}$	p
●	●	●	●								$\boxed{\circ}$	
●	●	●	○						$\boxed{\circ}$			
●	●	○	●							$\boxed{\circ}$		
●	●	○	○							$\boxed{\circ}$		
●	○	●	●					$\boxed{\circ}$				
●	○	●	○				○		○			
●	○	○	●			○		○				
●	○	○	○			○	○					
○	●	●	●	○	○							$\boxed{\circ}$
○	●	●	○	$\boxed{\circ}$								$\boxed{\circ}$
○	●	○	●	$\boxed{\circ}$								$\boxed{\circ}$
○	●	○	○	$\boxed{\circ}$								$\boxed{\circ}$
○	○	●	●		$\boxed{\circ}$							$\boxed{\circ}$
○	○	●	○				$\boxed{\circ}$					○
○	○	○	●			$\boxed{\circ}$						○
○	○	○	○			○	○					○

$\{p\}$ ☼ S

Figure 4.6: Illustration of the proof of Proposition 4.4.2.

In more concrete terms, if we perform an expansion step with, e.g., the clause formula $\rfloor p, \neg q \lfloor$ of the example in question, then there are at least two minimally unsatisfiable subsets contributing to a refutation of the extended branch on which p lies. Or, in terms of resolution, there are two different minimal clause sets from which the unit clause $\{\neg p\}$ may be derived. Since any calculus uses (variants of) the strengthening operation as inference mechanism, the existence of such examples destroys the hope that one can develop extremely restricted calculi which guarantee for each unsatisfiable formula the existence of exactly one proof. This observation illuminates a natural restriction of every work towards the avoidance of redundancy using calculus restriction only: however sophisticated the efforts, there will always remain redundancy.

Consequently, an important future research topic is to develop *global* pruning techniques which extract and use information from the search process itself.

Conclusion

We conclude this work with a brief summary of its main contributions to the advance of science, and mention the most important future research perspectives.

First, it has been demonstrated that the field of automated deduction can benefit a lot from meta-theoretical work of the type presented in Chapter 2. The formalization of intuitively existing abstraction ideas for deductions has produced a number of fundamental concepts for measuring the complexities of logic calculi. Particularly, the new notion of *polynomial transparency* promises to serve as a useful and research-stimulating property of deduction systems, and of transition relations in general. The application of the concept has provided new insights into the inferential power of basic deduction mechanisms, like *lemmata* and the *renaming* of formulae, and motivates the development of even more compact data structures than the ones considered in this work. A further challenging research perspective is to compare the difficulties of rendering certain transition relations polynomially transparent with other problems in complexity theory.

Secondly, in this thesis a number of calculi and inference mechanisms have been compared which play a central role in the area of automated deduction. We have uncovered new results concerning mutual polynomial simulation between the considered proof systems. Furthermore, the framework of *connection tableaux* has been developed which turned out as an optimal environment for reformulating and improving some of the well-known calculi like model elimination and the connection calculus. The structural richness of this framework simplifies the presentation of many calculi and permits more compact and elegant completeness and simulation proofs than for some of the original formalisms. This is important for further refinements and extensions of the systems and may help avoiding redundant work in the different frameworks, as illustrated with a study of the factorization and the C-reduction operations and their relation with lemmata and the atomic cut rule. An interesting future task is a complete clarification of *all* simulation possibilities between the presented systems.

Finally, we have designed *proof procedures* based on connection tableaux. It is demonstrated that the developed pruning mechanisms can be implemented very efficiently, by using a constraint technology based on syntactic term inequations. Also, two fundamental results are given which demonstrate that *local* pruning methods, i.e., methods that are restricted to the structures of deductions, are not sufficient for avoiding all of the redundancies occurring during proof search. Additionally, it is necessary to consider *global* techniques which compare deductions with one another. Here the use of the *matings* framework facilitates a gain in efficiency which cannot be achieved with the pure tableau format. In future also the complexities of *proof procedures* need to be investigated, that is, the effort for *finding* proofs. Since slight modifications of the control strategy can dramatically change the behaviour of proof procedures, it is very difficult to find a reliable and robust measure for the complexity of proof procedures.

References

[Aho et al., 1974] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Andrews, 1981] P. Andrews. Theorem Proving via General Matings. *Journal of the Association for Computing Machinery*, 28(2):193–214, 1981.

[Baaz and Leitsch, 1992] M. Baaz and A. Leitsch. Complexity of Resolution Proofs and Function Introduction. *Annals of Pure and Applied Logic*, 57:181–215, 1992.

[Beth, 1955] E. W. Beth. Semantic Entailment and Formal Derivability. *Mededlingen der Koninklijke Nederlandse Akademie van Wetenschappen*, 18(13):309–342, 1955.

[Beth, 1959] E. W. Beth. *The Foundations of Mathematics.* North-Holland, Amsterdam, 1959.

[Bibel, 1981] W. Bibel. On Matrices with Connections. *Journal of the ACM*, 28:633–645, 1981.

[Bibel, 1985] W. Bibel. Automated Inferencing. *Journal of Symbolic Computation*, 1:245–260, 1985.

[Bibel, 1987] W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, Braunschweig, second edition, 1987.

[Bläsius et al., 1981] K. Bläsius, N. Eisinger, J. Siekmann, G. Smolka, A. Herold, and C. Walther. The Markgraf Karl Refutation Proof Procedure. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 511–518, Vancouver, 1981.

[Boy de la Tour, 1990] T. Boy de la Tour. Minimizing the Number of Clauses by Renaming. *Proceedings of the 10th International Conference on Automated Deduction*, pages 558–572, 1990.

[Buro and Kleine Büning, 1992] M. Buro and H. Kleine Büning. Report on a SAT Competition. Technical report, Universität Paderborn, 1992.

[Chang and Lee, 1973] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[Church, 1936] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58:345–363, 1936.

[Colmerauer, 1982] A. Colmerauer. Prolog and Infinite Trees. In *Logic Programming*, K. L. Clark and S.-A. Tärnlund (eds.), pages 231–251. Academic Press, 1982.

[Cook, 1971] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, Vol. 6, pages 151–58, 1971.

[Cook and Reckhow, 1973] S. A. Cook and R. A. Reckhow. Time Bounded Random Access Machines. *Journal of Computer and Systems Sciences*, 7:354–375, 1973.

[Cook and Reckhow, 1974] S. A. Cook and R. A. Reckhow. On the Lengths of Proofs in the Propositional Calculus. *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, Seattle, Washington, pp. 135–148, 1974 (corrections are in SIGACT News 6(3):15–22, 1974).

[Corbin and Bidoit, 1983] J. Corbin and M. Bidoit. A Rehabilitation of Robinson's Unification Algorithm. In *Information Processing*, pages 909–914. North-Holland, 1983.

[Courcelle, 1983] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[Davis et al., 1962] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Eder, 1985a] E. Eder. Properties of Substitutions and Unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.

[Eder, 1985b] E. Eder. An Implementation of a Theorem Prover based on the Connection Method. In W. Bibel and B. Petkoff, editors, *AIMSA: Artificial Intelligence Methodology Systems Applications*, pages 121–128. North–Holland, 1985.

[Eder, 1991] E. Eder. Consolution and its Relation with Resolution. *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, pages 132–136, Morgan Kaufmann, 1991.

[Eder, 1992] E. Eder. *Relative Complexities of First-Order Calculi.* Vieweg, 1992.

[Frege, 1879] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*, 1879. Reprinted 1964.

[Gallier, 1986] J. P. Gallier. *Logic for Computer Science.* Harper & Row, 1986.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[Gentzen, 1935] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1935. Engl. translation in [Szabo, 1969].

[Gilmore, 1960] P. C. Gilmore. A Proof Method for Quantification Theory: Its Justification and Realization. IBM J. Res. Develop., pages 28-35, 1960. Reprinted in J. Siekmann and G. Wrightson (editors). *Automation of Reasoning. Classical Papers on Computational Logic*, Vol. 1, pages 151–158, Springer, 1983.

[Gödel, 1930] K. Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930.

[Goerdt, 1989] A. Goerdt. Regular Resolution versus Unrestricted Resolution, Universität Duisburg, Schriftenreihe des Fachbereichs Mathematik. Technical report, 1990, to appear in SIAM Journal of Computing.

[Haken, 1985] A. Haken. The Intractability of Resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[Herbrand, 1930] J. J. Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et des Lettres de Varsovie, Cl. III, math.-phys.*, 33:33–160, 1930.

[Hilbert and Ackermann, 1928] D. Hilbert and W. Ackermann. *Grundzüge der theoretischen Logik.* Springer, 1928. Engl. translation: Mathematical Logic, Chelsea, 1950.

[Hilbert and Bernays, 1934] D. Hilbert and P. Bernays. *Grundlagen der Mathematik.* Vol. 1, Springer, 1934.

[Hintikka, 1955] K. J. J. Hintikka. Form and Content in Quantification Theory. *Acta Philosophica Fennica*, 8:7–55, 1955.

[Hopcroft and Ullman, 1969] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relations to Automata.* Reading, Mass., 1969.

[Huet, 1976] G. Huet. *Resolution d'equations dans les languages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université de Paris VII, 1976.

[Huet, 1980] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, 1980.

[Jaffar, 1984] J. Jaffar. Efficient Unification over Infinite Terms. *New Generation Computing*, 2:207–219, 1984.

[Kapur and Narendran, 1986] D. Kapur and P. Narendran. NP-Completeness of the Set Unification and Matching Problems. *Proceedings of the 8th International Conference on Automated Deduction*, pages 487–495, 1986.

[Kleene, 1967] S. C. Kleene. *Mathematical Logic*. Wiley, New York, 1967.

[Knuth, 1968] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., 1968.

[Korf, 1985] R. E. Korf. Depth-First Iterative Deepening: an Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

[Kowalski and Hayes, 1969] R. A. Kowalski and P. Hayes. Semantic Trees in Automatic Theorem Proving. *Machine Intelligence*, 4:87–101, 1969.

[Kowalski, 1975] R. A. Kowalski. A Proof Procedure based on Connection Graphs. *Journal of the Association for Computing Machinery*, 22:572–595, 1975.

[Krivine, 1971] J.-L. Krivine. *Introduction to Axiomatic Set Theory*, Reidel, Dordrecht, 1971.

[Lassez et al., 1988] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. *Foundations of Deductive Databases and Logic Programming* (ed. J. Minker), pages 587–625, Morgan Kaufmann Publishers, Los Altos, 1988.

[Letz, 1988] R. Letz. Expressing First Order Logic within Horn Clause Logic. Technical report FKI-96-c-88, Technische Universität München, 1988.

[Letz et al., 1992] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[Letz, 1993a] R. Letz. The Deductive Power of the Cut Rule. Technical report, Technische Universität München, 1993.

[Letz, 1993b] R. Letz. On the Polynomial Transparency of Resolution. *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 123–129, Chambery, France, Morgan Kaufmann, 1993.

[Li and Vitányi, 1990] M. Li and P. M. B.Vitányi. Kolmogorov Complexity and its Applications. *Handbook of Theoretical Computer Science* (ed. J. van Leeuven), Vol. A, pages 187–254, Elsevier Science Publishers, 1990.

[Lloyd, 1984] J. W. Lloyd. *Foundations of Logic Programming.* Springer, 1984. Second edition, 1987.

[Loveland, 1968] D. W. Loveland. Mechanical Theorem Proving by Model Elimination. *Journal of the Association for Computing Machinery*, 15(2):236–251, 1968.

[Loveland, 1969] D. W. Loveland. A Simplified Format for the Model Elimination Theorem-Proving Procedure. *Journal of the Association for Computing Machinery*, 16:349–363, 1969.

[Loveland, 1978] D. W. Loveland. *Automated Theorem Proving: a Logical Basis.* North-Holland, 1978.

[Luckham, 1970] D. Luckham. Refinement Theorems in Resolution Theory. *Symposium on Automatic Demonstration*, Lecture Notes on Mathematics 125, pages 163–190, Springer, Berlin, 1970.

[Łukasiewicz and Tarski, 1930] J. Łukasiewicz and A. Tarski. Untersuchungen über den Aussagenkalkül. *Comptes rendus des Séances de la Société des Sciences et des Lettres de Varsovie*, 23:30–50, 1930.

[Martelli and Montanari, 1976] A. Martelli and U. Montanari. Unification in Linear Time and Space: a Structured Presentation. Technical report. Internal Rep. No. B76-16, 1976.

[Martelli and Montanari, 1982] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pages 258–282, 1982.

[Mayr, 1991] K. Mayr. Personal communication, Technische Universität München, 1991.

[McCarthy et al., 1962] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *The Lisp 1.5 Programmers Manual.* MIT Press, Cambridge, 1962.

[McCune, 1988] W. McCune. OTTER users' guide. Technical report, Mathematics and Computer Sci. Division, Argonne National Laboratory, Argonne, Illinois, USA, May 1988.

[Moret, 1982] B. M. E. Moret. Decision Trees and Diagrams. *ACM Computing Surveys*, 14(4):593–623, 1982.

[Ohlbach, 1991] H.-J. Ohlbach. Semantics Based Translation Methods for Modal Logics. *Journal of Logic and Computation*, 1(5):691–746, 1991.

[Ohlbach and Siekmann, 1991] H.-J. Ohlbach and J. H. Siekmann. The Markgraf Karl Refutation Proof Procedure. In *Computational Logic, Essays in Honour of John Alan Robinson*, pages 41–112, MIT press, 1991.

[Paterson and Wegman, 1978] M. S. Paterson and M. N. Wegman. Linear Unification. *Journal of Computer and Systems Sciences*, 16:158–167, 1978.

[Plaisted, 1990] D. A. Plaisted. A Sequent-Style Model Elimination Strategy and a Positive Refinement. *Journal of Automated Reasoning*, 6(4):389–402, 1990.

[Prawitz, 1960] D. Prawitz. An Improved Proof Procedure. *Theoria*, 26:102–139, 1960.

[Prawitz, 1969] D. Prawitz. Advances and Problems in Mechanical Proof Procedures. In J. Siekmann and G. Wrightson (editors). *Automation of Reasoning. Classical Papers on Computational Logic*, Vol. 2, pages 285–297, Springer, 1983.

[Reckhow, 1976] R. A. Reckhow. *On the Lenghts of Proofs in the Propositional Calculus.* PhD thesis, University of Toronto, 1976.

[Robinson, 1965a] J. A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[Robinson, 1965b] J. A. Robinson. Automatic Deduction with Hyper-Resolution. *International Journal Comp. Math.*, 1:227–234, 1965.

[Robinson, 1968] J. A. Robinson. The Generalized Resolution Principle. *Machine Intelligence*, 3:77–94, 1968.

[Shannon, 1938] C. E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Transactions of AIEE*, 57:713–723, 1938.

[Shostak, 1976] R. E. Shostak. Refutation Graphs. *Artificial Intelligence*, 7:51–64, 1976.

[Siekmann and Wrightson, 1983] J. Siekmann and G. Wrightson (editors). *Automation of Reasoning. Classical Papers on Computational Logic*, Vol. 1 and 2, Springer, 1983.

[Slagle, 1967] J. R. Slagle. Automatic Theorem Proving with Renamable and Semantic Resolution. *Journal of the Association for Computing Machinery*, 14:687–697, 1967.

[Smullyan, 1968] R. M. Smullyan. *First Order Logic.* Springer, 1968.

[Statman, 1979] R. Statman. Lower Bounds on Herbrand's Theorem. In *Proceedings American Math. Soc.*, 75:104–107, 1979.

[Stickel, 1988] M. A. Stickel. A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[Szabo, 1969] M. E. Szabo. *The Collected Papers of Gerhard Gentzen.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1969.

[Tarski, 1936] A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1, 1936.

[Tseitin, 1970] G. S. Tseitin. On the Complexity of Derivations in the Propositional Calculus. In A. O. Slisenko (ed.), *Studies in Constructive Mathematics and Mathematical Logic II*, pages 115–125, 1970.

[Turing, 1936] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[Urquhart, 1987] A. Urquhart. Hard Examples for Resolution. *Journal of the Association for Computing Machinery*, 34(1):209–219, 1987.

[van Emde Boas, 1990] P. van Emde Boas. Machine Models and Simulations. *Handbook of Theoretical Computer Science* (ed. J. van Leeuven), Vol. A, pages 1–66, Elsevier Science Publishers, 1990.

[van Leeuven, 1990] J. van Leeuven. Graph Algorithms. *Handbook of Theoretical Computer Science* (ed. J. van Leeuven), Vol. A, pages 527–631, Elsevier Science Publishers, 1990.

[van Orman Quine, 1955] W. van Orman Quine. A Way to Simplify Truth Functions. *American Mathematical Monthly*, 62, 1955.

[Venturini-Zilli, 1975] M. Venturini-Zilli. Complexity of the Unification Algorithm for First-Order Expressions. Technical report, Res. Rep. Consiglio Nazionale delle Ricerche Istituto per le Applicazioni del Calcolo, Rome, 1975.

[Wallen, 1989] L. Wallen. *Automated Deduction for Non-Classical Logic.* MIT Press, Cambridge, Mass., 1989.

[Warren, 1983] D. H. D. Warren. An Abstract PROLOG Instruction Set. Technical report, SRI, Menlo Park, California, USA, 1983.